

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Волгоградский государственный технический университет»

На правах рукописи



Егунов Виталий Алексеевич

**МОДЕЛИ И МЕТОДЫ АВТОМАТИЗАЦИИ
ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ
ДЛЯ РЕШЕНИЯ ИНЖЕНЕРНЫХ ЗАДАЧ
В ГЕТЕРОГЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СРЕДАХ**

Специальность 2.3.7. Компьютерное моделирование и автоматизация
проектирования

Диссертация
на соискание ученой степени доктора технических наук

Научный консультант:
д.т.н., профессор
Кравец Алла Григорьевна

Волгоград
2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	7
1. НАУЧНО-ТЕХНИЧЕСКИЕ ПОДХОДЫ К ПРОЕКТИРОВАНИЮ ЭФФЕКТИВНЫХ ПРОГРАММНЫХ СИСТЕМ.....	26
1.1 Объект проектирования.....	27
1.2 Жизненный цикл ПС.....	32
1.3 Средства автоматизации проектирования ПС.....	34
1.4 Существующие методологии разработки ПС для решения инженерных задач.....	35
1.5 Методы анализа ПС	47
1.6 Анализ критериев эффективности ПС.....	51
1.7 Анализ влияния архитектуры вычислительных систем на эффективность ПС	55
1.7.1 Иерархическая структура памяти.....	56
1.7.2 Параллельные вычисления.....	67
1.7.3 Распределенные вычисления.....	70
1.7.4 Векторизация вычислений.....	74
1.7.5 Использование ускорителей.....	83
1.8 Целевая вычислительная архитектура.....	86
1.9 Цели и задачи исследования.....	91
1.10 Основные результаты и выводы по главе 1.....	92
2. СОВРЕМЕННЫЕ МЕТОДЫ ПОВЫШЕНИЯ ЭФФЕКТИВНОСТИ ПРОГРАММНЫХ СИСТЕМ.....	94
2.1 Методы моделирования эффективных ПС для решения инженерных задач.....	95
2.2 Методы улучшения стратегии кеширования данных.....	111
2.3 Методы повышения эффективности векторизации вычислений.....	121

2.4	Инструментальные средства проектирования и анализа ПС.....	128
2.5	Основные результаты и выводы по главе 2.....	133
3	МЕТОДОЛОГИЯ ПРОЕКТИРОВАНИЯ ЭФФЕКТИВНЫХ ПРОГРАММНЫХ СИСТЕМ ДЛЯ РЕШЕНИЯ ИНЖЕНЕРНЫХ ЗАДАЧ В ГЕТЕРОГЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СРЕДАХ	135
3.1	Модель улучшения показателей эффективности программных систем.....	137
3.1.1	Исследование жизненного цикла программной системы.....	138
3.1.2	Мера аппаратурно-временной сложности.....	140
3.1.3	Анализ этапа разработки жизненного цикла программных систем.....	147
3.1.4	Оценка времени выполнения проектных процедур разработки ПС при традиционном подходе.....	154
3.1.5	Предлагаемый подход к проектированию ПС.....	156
3.1.6	Оценка времени выполнения проектных процедур разработки ПС при предлагаемом подходе.....	158
3.1.7	Сравнение традиционного и предлагаемого подходов к реализации проектных процедур разработки ПС.....	160
3.1.8	Этап модификации ЖЦ ПС.....	163
3.1.9	Оценка времени, затрачиваемого на этап модификации ПС.....	170
3.1.10	Сравнение традиционного и предлагаемого подходов к реализации проектных процедур модификации ПС.....	172
3.2	Математическая модель оценки эффективности проектируемых ПС.....	173
3.2.1	Метрики, используемые для оценки эффективности использования подсистемы памяти.....	177
3.2.2	Метрики, используемые для оценки эффективности векторизации вычислений.....	180

3.2.3	Метрики, используемые для оценки эффективности использования GPU.....	183
3.2.4	Прикладной протокол определения метрик ПС.....	188
3.2.4.1	Оценка вероятности «холодных» кеш-промахов.....	188
3.2.4.2	Оценка вероятности кеш-промахов при цикловом доступе к данным.....	192
3.3	Аналитический метод количественной оценки метрик использования подсистемы памяти	215
3.3.1	Экспериментальная проверка метода количественной оценки метрик использования подсистемы памяти	235
3.4	Имитационная модель исполнения программы в подсистеме памяти	241
3.5	Комплекс методов повышения эффективности ПС.....	255
3.5.1	Метод повышения эффективности использования подсистемы памяти.....	256
3.5.2	Метод повышения эффективности векторизации вычислений.....	260
3.5.3	Метод повышения эффективности использования GPU.....	267
3.6	Методология проектирования эффективных ПС.....	269
3.6.1	Получение метрик ПС методами статического анализа.....	272
3.6.2	Оценка эффективности ПС для гетерогенных кластеров.....	279
3.7	Основные результаты и выводы по главе 3.....	282
4.	ВЕРИФИКАЦИЯ МЕТОДОЛОГИИ ПРОЕКТИРОВАНИЯ ЭФФЕКТИВНЫХ ПРОГРАММНЫХ СИСТЕМ	286
4.1	Реализация методологического подхода для задачи приведения матрицы общего вида к ХФ методом Хаусхолдера.....	286
4.1.1	Реализация МУСК для базового преобразования Хаусхолдера	295
4.1.2	Энергоэффективность полученных вычислительных схем.....	309
4.1.3	Реализация MBV для базового преобразования Хаусхолдера	317

4.1.4 Результат реализации методологического подхода для задачи приведения матрицы общего вида к ХФ методом Хаусхолдера.....	336
4.2 Реализация методологического подхода для задачи матрично-векторного умножения	336
4.2.1 Реализация MBV для задачи матрично-векторного умножения.....	338
4.2.2 Результат реализации методологического подхода для задачи матрично-векторного умножения.....	345
4.3 Реализация методологического подхода для задачи обработки числовых последовательностей	346
4.3.1 Реализация MBV для задачи одномерной свертки.....	348
4.3.2 Реализация MBV для задачи двумерной свертки.....	354
4.3.3 Результат реализации методологического подхода для задачи обработки числовых последовательностей.....	357
4.4 Основные результаты и выводы по главе 4.....	358
5. АРХИТЕКТУРА И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СИСТЕМЫ ПОДДЕРЖКИ ЖИЗНЕННОГО ЦИКЛА ПРОЕКТИРУЕМЫХ ПС	360
5.1 Система поддержки жизненного цикла проектируемых ПС	360
5.1.1 Архитектура системы.....	361
5.1.2 Подсистема интерпретации и имитационного моделирования.....	363
5.1.3 Подсистема оценки эффективности кода.....	378
5.2 Апробация методологии проектирования эффективных ПС.....	381
5.2.1. Внедрение в ООО «ТЕСИС».....	383
5.2.2 Апробация в АО «Топ Системы».....	385
5.2.3 Внедрение в ОАО «Волжский абразивный завод».....	386
5.3 Реализация геометрического преобразования отражения для прототипа высокопроизводительного геометрического ядра.....	387
5.4 Основные результаты и выводы по главе 5.....	393
ЗАКЛЮЧЕНИЕ.....	395
СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ.....	398
СПИСОК ЛИТЕРАТУРЫ.....	400

ПРИЛОЖЕНИЯ.....	421
Приложение А – Свидетельства о регистрации программ для ЭВМ.....	421
Приложение Б – Документы, подтверждающие внедрение результатов.....	430

ВВЕДЕНИЕ

Актуальность темы исследования

В настоящее время проблеме разработки эффективных программных систем (ЭПС), используемых в гетерогенных вычислительных средах (ГВС) с различной архитектурой для решения вычислительно-сложных задач в научных и инженерных расчетах, системах искусственного интеллекта, уделяется большое внимание. ГВС представляет собой вычислительную среду, содержащую несколько вычислительных ресурсов, причем множества операций, выполняемых различными ресурсами, различны, каждый ресурс имеет свою функциональную специализацию. С точки зрения проектирования программных систем под гетерогенностью понимается использование в составе вычислительной среды нескольких типов устройств, каждое из которых обладает собственной системой команд. Программная система (ПС) - сложный технический объект, представляющий собой совокупность взаимодействующих программных компонентов. ПС является частью вычислительной среды (ВС), представляющей собой сочетание программной и аппаратной частей, при этом под повышением эффективности ПС понимается минимизация затрат, связанных с ПС на различных этапах жизненного цикла (ЖЦ), включая затраты на разработку, модификацию и эксплуатацию ПС. Эффективность ПС тесно связана с понятием эффективности использования ПС вычислительных ресурсов, которая определяется степенью загрузки вычислительных ресурсов (компонентов ВС), участвующих в процессе вычислений. Эффективность получаемых ПС и ряд других характеристик в значительной степени зависят от квалификации разработчика. На начальных этапах проектирования разработчики часто используют эвристический подход к повышению эффективности ПС, для дальнейшего улучшения характеристик полученных ПС используют различные инструментальные средства анализа и оптимизации. Оба указанных подхода не гарантируют получения

необходимого результата и, более того, указанные методы проектирования могут приводить к увеличению времени разработки и последующей модификации ПС.

Необходимо отметить, что современная российская действительность ориентирована на отечественные решения, вырос спрос на импортозамещение, в т.ч. в сегменте высокопроизводительных вычислений (ВПВ) для решения инженерных задач. К таким задачам относятся, в частности, вычислительные ядра для численного моделирования, статистического анализа, машинного обучения, геометрические ядра CAD/CAE-систем. Дополнительную актуальность теме исследования придают прогнозы, согласно которым доля ВПВ на рынке информационных технологий в ближайшие годы и десятилетия будет значительно расти.

В 2024 году Указом Президента РФ утверждена Стратегия научно-технологического развития Российской Федерации. Цель стратегии — «обеспечить устойчивое, динамичное и сбалансированное развитие России на долгосрочный период». Одним из направлений стратегии, позволяющих получить значимые научные и научно-технические результаты, создать отечественные наукоемкие технологии является «переход к передовым технологиям проектирования и создания высокотехнологичной продукции, основанным на применении интеллектуальных производственных решений, роботизированных и высокопроизводительных вычислительных систем, новых материалов и химических соединений, результатов обработки больших объемов данных, технологий машинного обучения и искусственного интеллекта».

Правительство РФ в 2020 г. утвердило Программу фундаментальных научных исследований в Российской Федерации на долгосрочный период (2021 - 2030 годы), в которой подчеркивается необходимость «создания методов, алгоритмов, инструментальных средств и пакетов прикладных программ для ВС сверхвысокой производительности», разработки «теоретических основ вычислительных методов и алгоритмов для

компьютерных систем высокой производительности». Подчеркивается важность применения ВПВ, которые окажут большое влияние на развитие фундаментальных наук.

Таким образом, тематика исследования является актуальной.

Степень разработанности темы исследования.

В процессе разработки программных средств в соответствии с ГОСТ Р ИСО/МЭК 12207-2010 различают два типа работ (видов деятельности): системные и программные. При выполнении работ программного типа широко используются средства автоматизации проектирования ПС, представляющие собой набор методов и инструментальных средств, известный как Computer-Aided Software Engineering (CASE) (ГОСТ Р ИСО/МЭК 14764-2002). В состав CASE входят средства анализа, проектирования и программирования ПС, CASE-средства определяются как программные средства для поддержки жизненного цикла ПС. Технологии CASE призваны повышать эффективность процессов проектирования и модификации, способствовать повышению надёжности и упрощению сопровождения созданных при помощи CASE-технологий ПС.

Разработке методов, предназначенных для повышения эффективности ПС для ГВС с различной архитектурой, а также реализация этих методов в виде CASE-технологий посвящены работы ряда исследователей. Так, например, коллективами под руководством Воеводина В.В., Воеводина Вл.В., Гергеля В.П., Каляева И.А., Котенко И.В., Саенко И.Б., Самарского А.А., Топоркова В.В. и др. проведено множество исследований по распараллеливанию вычислений, разработке методов и алгоритмов параллельной обработки данных, исследований в области высокопроизводительных вычислений. Большое количество исследований посвящено использованию специализированных ускорителей, ВС с реконфигурируемой архитектурой (работы Каляева И.А., Заборовского В.С., Левина И.И., Ронжина А.Л., Якубовского М.В., и др.). Значительное внимание исследователей уделяется проблемам повышения эффективности

алгоритмов при решении конкретных прикладных задач. Данные вопросы освещены в работах Воеводина В.В., Воеводина Вл.В., Гергеля В.П., Ортега Дж., Деммеля Дж., Голуба Дж., Ван Лоуна Ч., Донгарра Дж., Глинского Б.М., и др.

В то же время стоит отметить гораздо меньшее число исследований по кеш-оптимизации и векторизации вычислений, разработке моделей, предназначенных для повышения эффективности проектируемых ПС, и созданию на основе этих моделей CASE-средств. Методы, используемые для повышения эффективности векторизации вычислений, освещены в работах: С.Ларсен, Дж.Лиу, (Superword Level Parallelism, SLP), Д.Нузман, И.Розен, А.Закс (векторизация обработки чередующихся данных с промежутками), А. Эйхенбергер, П.Бу, К.О'Брайен (исследование вопросов выравнивания данных), Й.Чен, Ч.Мендис и др. (использование Single Instruction Multiple Operation Multiple Data, SIMOMD), Д.Нузман, А.Закс, С.Багсорхи (построение оптимизирующих компиляторов с блоками автовекторизации, Intel C++ Compiler, GCC) и др. В области разработки моделей проектирования ПС и методов, основанных на этих моделях известны исследования, связанные с разработкой моделей ВС с иерархической структурой памяти (С.Уильямс, Дж.Трейбиг, Х.Сентегель, К.Йотов, Т.Лоу, М.Фриго и др.). Методы, используемые для улучшения стратегии кэширования данных, описываются в работах: Ф.Густавсон, Дж.Маккомб (множественная реализация алгоритмов и эмпирический поиск параметров), Р.Уэйли, Дж.Доггараа, Д.Билмс, Дж.Деммель (автонастройка с использованием генерации кода), Д.Спампинато, Г.Белтер, К.Ванг (автонастройка с использованием аналитических техник), К.Йотов, Т.Лоу (аналитический поиск параметров) и др. Абсолютное большинство предложенных моделей ориентировано на решение задачи умножения плотных матриц. При этом аналитические подходы, описанные в ряде работ, не предполагают автоматизации, для каждого нового преобразования требуется проводить отдельное исследование. Отмечается также сложность

проведения подобного анализа, требующая наличия высокой квалификации и ряда компетенций у исследователя-разработчика. Для большинства подходов необходим непосредственный доступ к ВС, на которой планируется использование исследуемых ПС. Кроме того, предложенные методы получения необходимых характеристик проектируемых алгоритмов опираются на аналитическое моделирование, которое, как правило, выполняется для конкретного семейства микропроцессоров. При этом авторы методик не предлагают информационных систем, автоматизирующих данный процесс.

Таким образом, сложилась **проблемная ситуация**, сущность которой заключается в том, что:

- отсутствуют или являются неполными существующие подходы к разработке ЭПС, которые могут использоваться на начальном этапе проектирования и верификации алгоритмов
- исследования и разработки в сфере автоматизации проектирования ПС часто имеют частный характер;
- подходы к повышению эффективности ПС, основанные на кеш-оптимизации и векторизации вычислений, являются неполными и имеют существенные несовершенства
- существует необходимость разработки отечественных программных решений, направленных на проектирование ЭПС.

В проектировании ПС для ГВС выявлена **фундаментальная проблема**: научно-технические подходы к проектированию ЭПС обладают существенной неполнотой и имеют частный характер, что оказывает существенное влияние на выбор стратегии повышения эффективности ПС.

Научно-техническая проблема исследования заключается в разработке нового методологического подхода к автоматизации проектирования ПС, связанного с созданием моделей, методов и алгоритмов, предназначенных для повышения эффективности ПС для решения инженерных задач в ГВС на различных этапах ЖЦ.

Цели и задачи.

Целью работы является разработка научных основ поддержки жизненного цикла проектируемых программных систем для решения инженерных задач в гетерогенных вычислительных средах.

Для достижения цели исследования в работе поставлены и решены следующие **задачи**:

1. Исследовать существующие научно-технические подходы, применяемые для проектирования и повышения эффективности ПС.
2. Разработать методологию проектирования ПС для решения инженерных задач в ГВС, повышающую эффективность ПС на всех этапах ЖЦ, включающую основные проектные процедуры, модели и методы:
 - компьютерные модели оптимального проектирования ПС для ГВС;
 - методы повышения эффективности ПС для ГВС.
3. Верифицировать созданную методологию проектирования ЭПС на примере практически значимых вычислительно-сложных задач.
4. Разработать архитектурные и программные решения системы поддержки жизненного цикла проектируемых ПС.

В работе в качестве **объекта исследования** выступает жизненный цикл ПС для решения инженерных задач в ГВС, а **предметом исследования** являются модели, методы и алгоритмы автоматизации проектирования для повышения эффективности ПС в ГВС на различных этапах жизненного цикла ПС.

Научная новизна. Впервые предложена совокупность моделей и методов поддержки жизненного цикла программных систем для решения инженерных задач в гетерогенных вычислительных средах, реализующая комплексный подход к повышению эффективности программных систем, включающая в себя:

1. Модель улучшения показателей эффективности программных систем, основанная на методах статического анализа исходного кода и оценки аппаратурно-временной сложности ПС, отличающаяся от известных

переносом этапа анализа эффективности ПС с динамического анализа готового приложения на статический анализ исходного кода, позволяющая снизить количество итераций проектных процедур на этапах разработки и модификации жизненного цикла ПС (пп. 3, 10 паспорта специальности).

2. Математическая модель оценки эффективности проектируемых программных систем, основанная на классификации подсистем целевой вычислительной архитектуры, отличающаяся от известных совокупностью новых метрик ПС и обоснованием их применимости, что позволяет на различных этапах ЖЦ комплексно оценить ПС при эксплуатации в гетерогенных вычислительных средах. (п. 6 паспорта специальности).

3. Аналитический метод определения максимального размера блока при регулярном доступе к данным, основанный на отображении множества кешируемых данных на множества кеш-памяти, отличающийся от известных прямым способом расчета параметров регулярных шаблонов доступа к данным, позволяющий выбирать параметры алгоритмов, минимизирующие количество кеш-промахов при доступе к данным проектируемой ПС. (п. 6 паспорта специальности).

4. Имитационная модель исполнения программы в гетерогенных вычислительных средах с иерархической структурой памяти, основанная на множествах системных и пользовательских операций, где впервые использован полный набор параметров, характеризующих различные уровни подсистемы памяти, что позволяет определять характеристики программных систем на конкретной целевой вычислительной архитектуре, а также характеристики подсистемы памяти для проектируемой ПС. (п. 8 паспорта специальности).

5. Комплекс методов повышения эффективности программных систем, основанный на предложенных моделях (п. 6 паспорта специальности):

– метод использования подсистемы памяти, в котором впервые реализован аналитический подход к оценке эффективности проектируемых ПС на базе совокупности метрик ПС, позволяющий осуществить улучшение

стратегии кеширования данных в гетерогенных вычислительных средах с иерархической структурой памяти;

– метод векторизации вычислений, в котором впервые реализован новый способ векторной обработки данных, что позволяет избежать потерь производительности программных систем в гетерогенных вычислительных средах при доступе к данным;

– метод использования GPU, в котором впервые реализован аналитический подход к оценке эффективности проектируемых программных систем на базе совокупности метрик ПС, позволяющий повысить уровень загрузки и степень сбалансированности загрузки потоковых процессоров.

6. Методология проектирования эффективных программных систем для решения инженерных задач в гетерогенных вычислительных средах, основанная на исследовании жизненного цикла ПС, впервые реализующая подход непрерывного улучшения показателей проектируемых ПС, отличающаяся от известных реализацией предложенных моделей и методов, которая позволяет принимать проектные решения по эффективному использованию вычислительных ресурсов на начальных этапах жизненного цикла ПС (пп. 1,4 паспорта специальности).

Теоретическая и практическая значимость работы.

Теоретическая значимость диссертационного исследования состоит в разработке и научном обосновании нового методологического подхода к проектированию эффективных программных систем для решения инженерных задач в гетерогенных вычислительных средах. В отличие от существующих подходов, предложенная методология обеспечивает перенос анализа эффективности с завершающих этапов жизненного цикла на ранние стадии проектирования за счёт использования комплекса новых моделей и методов: математической и имитационной моделей оценки эффективности с новой совокупностью метрик, а также аналитических методов анализа и повышения производительности. Тем самым работа вносит вклад в развитие

теории автоматизации проектирования и компьютерного моделирования, создавая формальный аппарат для системного, количественно обоснованного повышения эффективности проектируемых программных систем.

Практическая значимость диссертационного исследования определяется созданием комплексного подхода, позволяющего решать задачи автоматизации проектирования и повышения эффективности программных систем для инженерных и научных расчетов в гетерогенных вычислительных средах. На основе предложенных моделей и методов разработаны архитектура и программная реализация системы поддержки жизненного цикла, включающая статический анализатор кода с функциями частично-динамического анализа. В системе реализован комплекс алгоритмических решений по оптимизации стратегий кеширования, векторизации вычислений и использованию GPU. Эти решения позволяют снизить затраты и трудоемкость проектирования, уменьшить требования к уровню компетенций разработчиков за счет автоматизации анализа эффективности ПС на ранних этапах жизненного цикла.

Практическая ценность результатов подтверждается регистрацией 9 программ для ЭВМ и внедрением разработанных решений в промышленную практику, что отражено в актах внедрения. В ходе верификации на реальных задачах достигнуты значительные количественные улучшения: ускорение вычислений до 4 раз при одновременном повышении энергоэффективности. Совокупный годовой экономический эффект от внедрения результатов диссертационного исследования составил 2,8 млн руб.

Методология и методы исследования. Для решения поставленных задач в работе используются методы объектно-ориентированного программирования, вычислительной математики и теории алгоритмов, методы математического и имитационного моделирования, методы автоматизированного проектирования и CASE-технологий.

Положения, выносимые на защиту.

1. Модель улучшения показателей эффективности программных систем, позволяющая снизить количество итераций проектных процедур на этапах разработки и модификации ЖЦ ПС.

2. Математическая модель оценки эффективности проектируемых ПС, позволяющая на различных этапах ЖЦ комплексно оценить ПС при эксплуатации в гетерогенных вычислительных средах.

3. Аналитический метод определения максимального размера блока при регулярном доступе к данным, позволяющий производить исследование показателей эффективности использования подсистемы памяти проектируемых ПС.

4. Имитационная модель исполнения программы в ГВС с иерархической структурой памяти, позволяющая определять технические характеристики ПС на конкретной целевой вычислительной архитектуре, а также характеристики подсистемы памяти для проектируемой ПС.

5. Комплекс методов повышения эффективности ПС:

- метод повышения эффективности использования подсистемы памяти, позволяющий принимать проектные решения по улучшению стратегии кеширования данных в ГВС с иерархической структурой памяти;

- метод повышения эффективности векторизации вычислений, позволяющий избежать потерь производительности ПС при доступе к данным;

- метод повышения эффективности использования GPU, позволяющий повысить уровень загрузки и степень сбалансированности загрузки потоковых процессоров.

6. Методология проектирования ЭПС для решения инженерных задач в ГВС, позволяющая на начальных этапах ЖЦ ПС принимать проектные решения по эффективному использованию вычислительных ресурсов.

7. Концепция и архитектура системы поддержки жизненного цикла проектируемых ПС на основе предложенных моделей и методов, которая позволяет обеспечить повышение эффективности ПС.

Степень достоверности и апробация результатов.

Достоверность результатов диссертации обеспечивается корректным использованием методов исследований и математического аппарата, а также подтверждается результатами проведенных экспериментов и успешного внедрения разработанных ПС. Реализуемые модели, методы и алгоритмы строго аргументированы и критически оценены по сравнению с другими известными результатами. Полученные результаты подтверждены серией вычислительных экспериментов на наборе задач, характерном для инженерных и научных вычислений, где продемонстрировано значительное улучшение показателей производительности и энергоэффективности по сравнению с базовыми реализациями и возможностями современных оптимизирующих компиляторов. Дополнительным подтверждением достоверности служит практическое использование разработанного программного обеспечения и архитектурных решений, успешное внедрение которых зафиксировано актами внедрения и сопровождается документированным экономическим эффектом

Диссертационная работа выполнена в рамках одного из научных направлений Волгоградского государственного технического университета, а также проектов РНФ № 25-21-20073 – «Модели и методы проектирования эффективного программного обеспечения для высокопроизводительных вычислений» (руководитель), РФФИ № 18-47-340010 - "Разработка, адаптация и анализ эффективности алгоритмов реализации базовых операций линейной алгебры на неоднородных многопроцессорных вычислительных архитектурах с потоковыми FPGA – ускорителями" (руководитель), № 16-47-340385 – «Адаптация алгоритмического обеспечения ядра системы междисциплинарного моделирования с многотельным представлением

технических объектов для неоднородных многопроцессорных вычислительных архитектур» (исполнитель),

Диссертационная работа выполнена в рамках ряда проектов, выполненных по заказу АО «Топ Системы» (№ 44/227-19 – «Разработка специализированных объектно-ориентированных библиотек классов, предназначенных для решения геометрических задач прототипа САД системы, связанных с генерацией, модификацией и анализом трёхмерных математических моделей»); ООО «ТЕСИС» (№ 44/776-15 – «Модификация алгоритмов реализации матричных операций с использованием векторных расширений систем команд, дополненных совмещенными сложениями–умножениями (FMA3), и архитектур процессоров с длинным командным словом»); ООО "ОКБ АНТ", ОАО «Волжский абразивный завод» и др., в т.ч. ряда проектов, связанных с разработкой ПС по теме диссертации, в которых автор являлся научным руководителем или исполнителем. Получены 3 акта о внедрении результатов исследования.

Основные положения диссертационной работы докладывались и обсуждались на квалификационной секции международной научной конференции «Математические методы в технике и технологиях» ММТТ-37 (2024, Казань), международных конференциях ММТТ (2026, Санкт-Петербург, 2025, Самара, 2024, Казань, 2023; Нижний Новгород, 2022; Ярославль), «Суперкомпьютерные дни в России (Russian Supercomputing Days)» (2019, 2016, Москва), «Параллельные вычислительные технологии (ПаВТ)» (2021, Волгоград; 2020, Пермь; 2016, Архангельск), «Национальный Суперкомпьютерный Форум (НСКФ)» (2015-2019, Переславль-Залесский), «Creativity in Intelligent Technologies and Data Science (CIT&DS)» (2015 – 2025, Волгоград), «International Scientific Conference Artificial intelligence and digital technologies in technical systems (AIDTTS)» (2020, Волгоград), «Перспективные информационные технологии (Advanced Information Technologies and Scientific Computing)» (2020, Самара), «International Conference on Artificial Life and Robotics (ICAROB)» (2018, Япония),

«Инновационные, информационные и коммуникационные технологии (ИНФО)» (2024, Сочи, 2023, Махачкала; 2022, Сочи).

Материалы диссертационного исследования легли в основу курсов «Программная инженерия», «Параллельные вычислительные технологии», «Применение ускорителей и оптимизация приложений в системах искусственного интеллекта».

Публикации.

Основные результаты диссертации опубликованы в 175 научных работах, в том числе 2 монографиях, 45 – в изданиях, рекомендованных ВАК России, 23 публикациях в международных изданиях, проиндексированных в Scopus и Web of Science и 3 отчетах о НИР. Получены 9 свидетельств о регистрации программ для ЭВМ.

Структура и объем работы.

Диссертация состоит из введения, пяти глав, заключения, библиографического списка из 195 наименований и 2 приложений. Общий объем диссертации 433 страницы, основная часть содержит 126 рисунков и 43 таблицы.

Соответствие паспорту научной специальности.

Полученные в диссертационной работе результаты соответствуют следующим пунктам паспорта специальности 2.3.7. Компьютерное моделирование и автоматизация проектирования, а именно пункту 1 - методология компьютерного моделирования и автоматизированного проектирования в технике и технологиях, включая постановку, формализацию и типизацию проектных и технологических процедур, алгоритмов и процессов проектирования; пункту 3 - разработка научных основ построения комплекса средств САПР, включающего информационное, математическое, лингвистическое, методическое, техническое, программное обеспечение непрерывной информационной поддержки жизненного цикла проектируемых объектов пункту; 4 - разработка принципиально новых и повышение эффективности существующих методов и средств

взаимодействия проектировщик – система, включая компьютерные модели и технологии искусственного интеллекта; пункту 6 - разработка компьютерных моделей, алгоритмов, программных комплексов оптимального проектирования технических изделий и процессов; пункту 8 - разработка имитационных компьютерных моделей для оценки и тестирования технических, экономических, экологических характеристик технических объектов проектирования; пункту 10 - разработка научных основ реализации жизненного цикла проектирование – производство – эксплуатация – утилизация, построения интегрированных средств управления проектными работами и унификации прикладных протоколов информационной поддержки.

Сведения о личном вкладе автора.

Личный вклад автора заключается в разработке основных теоретических положений, выносимых на защиту; в разработке моделей, методов и методологии, вошедших в структуру концепции; в разработке концепции и архитектуры системы поддержки жизненного цикла ПС. Вклад автора в основные опубликованные работы был определяющим. Все представленные в диссертации положения, выносимые на защиту, получены лично автором.

В первой главе выполнен анализ научно-технических подходов к проектированию ЭПС. Рассмотрены модели жизненного цикла ПС, основные методологии проектирования ПС, CASE-технологии как средства автоматизации проектирования ПС. Исследованы вопросы, связанные с поддержкой жизненного цикла ПС в области ВПВ, влиянием методологий проектирования на качество получаемых ПС. Исследованы вопросы, связанные с методиками проектирования ПС которые используются для разработки инженерных и научных приложений в области ВПВ и их влиянием на качество получаемых ПС. Исследованы методы анализа программных систем, выделены две группы подобных методов: статический и динамический анализ ПС. Показано, что потенциально динамический

анализ позволяет выявить большее количество ошибок в программе, проанализировать ее эффективность, однако он применяется к уже готовому коду, в который заложены возможные неэффективные проектные решения. Избежать выполнения неудачных итераций в процессе проектирования ПС можно, если включить анализ эффективности ПС в ранние этапы проектирования, например, в проведение статического анализа исходного кода на этапе разработки ПС. Проанализированы критерии эффективности ПС и влияние архитектуры ВС на эффективность ПС, Рассмотрены группы критериев эффективности, зависящих от типа ПС, критерии эффективности, характеризующие ПС как технический объект, подлежащий разработке и модификации. Приведены основные понятия и определения предметной области. Определено понятие ПС как технического объекта, определены объект проектирования и целевая вычислительная архитектура. Проведен анализ влияния архитектуры вычислительных систем на эффективность программных систем, в т.ч. анализ влияния иерархической структуры памяти, распараллеливания вычислений, реализации распределенной модели вычислений, векторизации вычислений.

Сформулированы выводы о необходимости разработки новых подходов к проектированию ЭПС для решения инженерных задач.

Во второй главе проведен анализ современных методов повышения эффективности ПС в области ВПВ. Выполнен обзор моделей в области ВПВ, используемых для повышения эффективности ПС, обзор методов, используемых для улучшения стратегии кеширования данных, векторизации вычислений, использования GPU, а также инструментальных средств, применяемых для анализа и улучшения характеристик ПС.

Основное внимание уделяется моделям исполнения, которые исследуют поведение ВС с целью изучения, проверки и валидации. Рассматриваются модели ВС с иерархической структурой памяти, акцент делается на том, что некоторые подобные подходы основаны на моделировании производительности целевой ВС. Рассматриваются модели

двух категорий: моделирование “черного ящика”, которое опирается на статистические методы и машинное обучение для описания и прогнозирования производительности ВС на основе наблюдаемых данных и моделирование “белого ящика”, которое использует упрощенные математические модели для описания взаимодействия кода с оборудованием.

Среди методов улучшения стратегии кеширования данных выделены использование пространственной и временной локальности данных (в т.ч. оптимизация доступа и расположения данных в памяти), блочная реализация алгоритмов. Проанализированы параметризованные модели алгоритмов, в т.ч. эмпирический поиск параметров, автоматическая настройка с генерацией кода, использование аналитических моделей. Исследованы вопросы применения кеш-симуляторов. Исследованы вопросы автоматической векторизации кода современными оптимизирующими компиляторами, ограничения блоков автоматической векторизации. Проанализированы методы автоматической векторизации циклов, векторизации зависимостей и чередующихся данных с промежутками, вопросы влияния выравнивания данных на эффективность векторизации кода.

Рассмотрены инструментальные средства, которые используются для анализа и улучшения характеристик ПС для ГВС, в т.ч. оптимизирующие компиляторы, высокопроизводительные библиотеки, инструменты статического и динамического анализа.

Сформулированы выводы о необходимости разработки комплекса средств поддержки ЖЦ ПС для решения инженерных задач в ГВС.

В третьей главе проводится разработка основных положений методологии проектирования ПС для решения инженерных задач в ГВС, повышающей эффективность ПС на всех этапах жизненного цикла. Под повышением эффективности ПС понимается минимизация затрат, связанных с ПС на различных этапах ЖЦ, включая затраты (например, затраты на оборудование, разработчиков и т.д.) на разработку, модификацию и эксплуатацию ПС.

Проведено исследование ЖЦ ПС в аспекте создания и эксплуатации ПС в области ВПВ для решения научных и инженерных задач, для которых важными являются вопросы выбора алгоритма решения задачи и повышения эффективности ПС. ЖЦ состоит из трех этапов: разработки, модификации и эксплуатации.

Представлена модель улучшения показателей эффективности ПС, представляющая собой модель итеративного процесса разработки и модификации ПС, которая используется для повышения эффективности ПС на этапах разработки и модификации. Модель отличается от известных подходов переносом этапа анализа эффективности ПС с динамического анализа готового приложения на статический анализ исходного кода, что позволяет снизить количество итераций проектных процедур на этапах разработки и модификации жизненного цикла программных систем.

Для принятия проектных решений по повышению эффективности ПС в рамках модели улучшения показателей эффективности была предложена математическая модель оценки эффективности проектируемых ПС в ГВС с различными характеристиками. В рамках данной модели предложены метрики ПС, позволяющие на различных этапах ЖЦ комплексно оценить ПС при эксплуатации в ГВС.

Была исследована проблема оценки значений вероятности «холодных» или обязательных кеш-промахов. Важным в данном случае является вопрос определения максимального размера блока, при котором генерируются только «холодные» промахи и не происходит вытеснения строк из кеш-памяти, т.к. данный вопрос исследуется в процессе разработки блочных алгоритмов. В случаях, когда кеш-память заполняется равномерно, максимальный размер такого блока совпадает с размером кеш-памяти. Однако, в случае регулярного доступа к данным, которые не расположены в памяти непрерывно, это не так, потому что кеш-память заполняется неравномерно. Для оценки размера блока в данном случае был предложен аналитический метод определения максимального размера блока при

регулярном доступе к данным, позволяющий выбирать параметры алгоритмов, минимизирующие количество кеш-промахов при доступе к данным проектируемой ПС.

В главе 3 представлена имитационная модель исполнения программы в ГВС с иерархической структурой памяти, позволяющая определять характеристики ПС на заданной целевой вычислительной архитектуре, а также конфигурацию подсистемы памяти для проектируемой ПС и параметры алгоритмов. Она представляет собой детерминированную модель, описывающую ГВС с иерархической структурой памяти, содержащей N уровней, каждый уровень реализован в виде кеш-памяти, которая описывается набором параметров. На вход модели подается исходный код проектируемой ПС, который преобразуется в последовательность пользовательских операций, выполнение которых моделируется в ГВС с заданными характеристиками. В результате формируются выходные параметры, представляющие собой характеристики ПС на целевой вычислительной архитектуре, параметры ВС для ПС или параметры алгоритмов. Модель применяется в случае невозможности получения значений метрик ПС аналитическим способом

На основе предложенных моделей был разработан комплекс методов повышения эффективности ПС:

- использования ПП, позволяющий принимать проектные решения по улучшению стратегии кеширования данных в ГВС с иерархической структурой памяти;
- векторизации вычислений, позволяющий избежать потерь производительности ПС при доступе к данным;
- использования GPU, позволяющий повысить уровень загрузки и степень сбалансированности загрузки потоковых процессоров.

На базе разработанных моделей и методов была предложена методология проектирования эффективных ПС для решения инженерных задач в ГВС. Методология основана на взаимодействии проектировщика ПС

с СП ЖЦ, основанном на реализации предложенных моделей и методов. СП ЖЦ позволяет:

- выполнять аналитическую оценку метрик ПС;
- проводить имитационное моделирование исполнения ПС на ГВС;
- выбирать проектные решения по повышению эффективности ПС.

В четвертой главе представлена верификация созданной методологии проектирования ЭПС на примере практически значимых вычислительно-сложных задач на примере операций BLAS 2 и 3 уровня и алгоритмов обработки числовых последовательностей. Проведены вычислительные эксперименты, доказана эффективность предложенной методологии

В пятой главе представлены концепция и архитектура системы поддержки ЖЦ проектируемых ПС, реализующей предложенные модели и методы. Система позволяет выбирать проектные решения по повышению эффективности ПС и состоит из трех основных подсистем: подсистемы формирования характеристик кода, подсистемы оценки эффективности кода, подсистемы поддержки проектирования ПС. Описаны внедрение и апробация предложенных программных решений.

В Заключении приводятся основные результаты, полученные автором в диссертационном исследовании.

В Приложениях приведены документы, подтверждающие апробацию и внедрение результатов работы.

Автор выражает **благодарность** научному консультанту Кравец А.Г. за помощь в подготовке диссертации; Славину О.А., Чистяковой Т.Б., Щербакову М.В., Андрееву А.Е., Кравчене П.Д. за обсуждение результатов исследований.

1. НАУЧНО-ТЕХНИЧЕСКИЕ ПОДХОДЫ К ПРОЕКТИРОВАНИЮ ЭФФЕКТИВНЫХ ПРОГРАММНЫХ СИСТЕМ

Глава содержит анализ методологий проектирования и анализа ПС для ГВС с различной архитектурой.

В разделе 1.1 рассмотрены стандарты в области проектирования ПС, определен объект проектирования

В разделе 1.2 рассмотрен жизненный цикл ПС, процессы, выполняемые в течение ЖЦ ПС. Описаны процессы ЖЦ, анализируемые в диссертационной работе и их соответствие категориям процессов ЖЦ в соответствии со стандартом.

В разделе 1.3 рассмотрены средства автоматизации проектирования ПС (специализированные инструментальные средства или CASE-средства), приводятся различные типы CASE-средств.

В разделе 1.4 рассмотрены основные методологии проектирования ПС. Были выделены две группы методологий: методологии со структурой «шаг за шагом» (Waterfall) и итеративные (инкрементальные) методологии. Показано, что в области ВПВ используются итеративные методологии проектирования ПС. Проведено исследование вопросов, связанных с методиками проектирования ПС которые используются для разработки инженерных и научных приложений в области ВПВ и их влиянием на качество получаемых ПС.

В разделе 1.5 рассмотрены методы анализа программных систем. Были выделены две группы подобных методов: статический и динамический анализ ПС. Показано, что потенциально динамический анализ позволяет выявить большее количество ошибок в программе, проанализировать ее эффективность, однако он применяется к уже готовому коду, в который заложены возможные неэффективные проектные решения. Избежать выполнения неудачных итераций в процессе проектирования ПС можно, если включить анализ эффективности ПС в ранние этапы проектирования,

например, в проведение статического анализа исходного кода на этапе разработки ПС

В разделе 1.6 .проведен анализ критериев эффективности ПС. Рассмотрены группы критериев эффективности, зависящих от типа ПС, критерии эффективности, характеризующие ПС как технический объект, подлежащий разработке и модификации.

В разделе 1.7 проведен анализ влияния архитектуры ВС на эффективность ПС, в т.ч. анализ влияния иерархической структуры памяти, распараллеливания вычислений, реализации распределенной модели вычислений, векторизации вычислений, использования ускорителей.

В разделе 1.8 приведена целевая вычислительная архитектура, под которой понимается архитектура ВС, в которой реализуются предлагаемые модели и методы проектирования ЭПС. Совокупность новых моделей и методов предлагается реализовать в ГВС, имеющей в своем составе следующие вычислительные ресурсы: вычислительная система с общей памятью, построенная по иерархической схеме, векторный процессор или векторное расширение команд, графический процессор.

В разделе 1.9 формулируются цели и задачи исследования, в разделе 1.10 приводятся основные результаты и выводы по главе 1.

1.1 Объект проектирования

Под ВПВ понимают применение вычислительных систем высокой производительности для решения вычислительно-сложных задач. Чаще всего ВПВ связаны с решением сложных технических задач, вычислениями в рамках инженерных задач, научных исследований, системах ИИ. В последние годы ВПВ стали активно применяться в бизнесе, хранилищах данных и связанных с ними технологиях обработки больших данных и в ряде других областей.

Технологии ВПВ в последнее время реализуются с помощью вычислительных кластеров, представляющих собой группу компьютеров, объединенных каналами связи. Компьютеры кластера совместно решают вычислительно-сложные задачи, поэтому с точки зрения пользователя представляют собой единый аппаратный ресурс.

ГОСТ Р ИСО/МЭК 12207 [157] устанавливает связь между системой в целом и программным средством. На практике интерпретация термина «система» уточняется с помощью ассоциативного существительного, может заменяться контекстно-зависимым синонимом, например, «вычислительный кластер»

В соответствии с ГОСТ Р ИСО/МЭК ТО 15271-2002 [160] «система является комбинацией технических средств, компьютеров, программных средств, материалов, персонала и возможностей». Стандарт описывает, как система становится комбинацией технических и программных средств. Это означает, что при создании работоспособной системы учитываются как аппаратные, так и программные компоненты.

ВС, включая вычислительные кластеры, являясь системой, представляет собой комбинацию аппаратных и программных средств. Для автономных систем аппаратные средства представляют собой совокупность вычислительных ресурсов ВС. Для вычислительных кластеров под данным термином понимается совокупность отдельных компьютеров, узлов кластера, и высокоскоростных каналов связи, объединяющих отдельные узлы в единую вычислительную среду.

Программные средства вычислительного кластера (рисунок 1.1) включают в себя:

- специализированные ПС, представляющие собой инструменты и системы, используемые для реализации и создания высокопроизводительных вычислительных систем;
- ПС, проектируемые для решения вычислительно-сложных задач.



Рисунок 1.1 – Программные средства вычислительного кластера

Для достижения требуемой производительности в процессе проектирования ПС для вычислительных кластеров необходимо решить несколько технических задач:

- максимально эффективно использовать вычислительные ресурсы отдельных узлов вычислительного кластера для повышения эффективности проектируемых ПС в рамках отдельных узлов;
- распределить вычисления между узлами вычислительного кластера за счет распараллеливания вычислений.

Диссертационное исследование направлено на разработку методологии проектирования ПС, которые обрабатывают большие массивы данных и осуществляют регулярный доступ к данным в соответствии с определенными шаблонами. К подобным ПС относятся:

- ПС для решения научных и инженерных задач, требующих большого времени вычислений, включая вычислительные ядра для численного моделирования, статистического анализа, машинного обучения, обработки больших данных и т.д..
- ПС для решения инженерных задач, требующие минимального времени отклика ВС, включая CAD/CAE-системы, а также реализацию геометрических ядер, ПС обработки мультимедиа-контента и т.д.

Данные ПС обладают высокой степенью локальности данных и проектируются для использования как в автономных ВС (компьютерах), так

и в вычислительных кластерах в зависимости от назначения и условий эксплуатации.

Объектом проектирования являются программные системы для решения инженерных задач, повышающие эффективность использования вычислительных ресурсов ВС, в т.ч. узлов вычислительного кластера. Данные ПС относятся к прикладным ПС, разрабатываются на компилируемых языках программирования, относятся к проблемно-ориентированным ПС, предназначенным для решения научных и инженерных задач.

Эффективность использования вычислительных ресурсов определяется степенью загрузки вычислительных ресурсов (компонентов ВС), участвующих в процессе вычислений (1.1).

$$E = \frac{O_R * K}{O_T} \quad (1.1)$$

где E – эффективность использования вычислительного ресурса;

O_T – общее количество операций, формирующих множество $S_O = \{O_T\}$, выполняемых в процессе исполнения ПС, потенциально использующих вычислительный ресурс;

O_R – количество операций из множества S_O , использующих данный ресурс, формируют множество $S_R = \{O_R\}$;

K - коэффициент, принимающий значения от 0 до 1 и характеризующий способ выполнения операций из множества S_R .

Максимальное значение показателя эффективности использования вычислительного ресурса равно 1. Значение коэффициента K зависит от типа анализируемой подсистемы ВС.

В случае анализа эффективности использования подсистемы памяти, оценка в соответствии с (1.1) показывает отношение количества обращений к кеш-памяти (возможно, определенного уровня) к общему количеству операций обращения к памяти. Максимальная эффективность 1 в данном случае означает, что все обращения осуществляются к кеш-памяти (или

анализируемому уровню кеш-памяти). Достичь данного результата на практике невозможно, поэтому необходима максимизация данного параметра. Коэффициент K в данном случае всегда равен 1, т.к. обращение к кеш-памяти либо происходит, либо нет.

В случае анализа эффективности использования подсистемы векторизации вычислений оценка в соответствии с (1.1) показывает отношение количества векторных операций обработки данных к общему количеству подобных операций. Максимальная эффективность 1 в данном случае означает, что все операции обработки данных были векторизованы. Коэффициент K в данном случае характеризует данные операции с точки зрения достижения максимально возможной производительности, например, учитывая долю невыровненных обращений к векторным данным среди общего количества подобных обращений.

В случае анализа эффективности использования GPU множества S_O и S_R совпадают, т.к. анализируемые операции полностью выполняются на GPU. В данном случае эффективность зависит от коэффициента K , значение которого определяется рядом факторов, таких как плотность вычислений, локальность данных и других факторов.

Средневзвешенный параметр эффективности можно определить следующим образом (1.2)

$$E_{av} = \frac{\sum_{i=1}^N (k_i * E_i)}{N} \quad (1.2)$$

где k_i - весовые коэффициенты;

E_i – эффективность i -ой подсистемы, N – количество подсистем.

В соответствии с ГОСТ 23501.108-85 [154] установлен ряд признаков классификации САПР, в т.ч.:

- тип объекта проектирования;
- разновидность объекта проектирования;
- сложность объекта проектирования;
- уровень автоматизации проектирования;

- комплексность автоматизации проектирования.

По типу объекта проектирования ГОСТ выделяет ряд классификационных группировок, в т.ч. САПР программных изделий. Характеристика данной классификационной группировки в соответствии с ГОСТ: «Проектирует программы для электронных вычислительных машин, станков с ЧПУ, роботов и т.д.».

На рисунке 1.2 показаны компоненты системы, используемые в процессе автоматизации проектирования.



Рисунок 1.2 – Автоматизация проектирования ПС

Проектируемые ПС взаимодействуют с системным программным обеспечением и проектируются с использованием инструментальных средств. Для автоматизации проектирования необходимо разработать средства автоматизации проектирования. В свою очередь для решения этой задачи необходимо разработать новую методологию проектирования ПС.

1.2 Жизненный цикл ПС

ГОСТ Р ИСО/МЭК 12207 [157], идентичный международному стандарту ISO/IEC 12207:2008 «System and software engineering — Software life cycle processes», принят Федеральным агентством по техническому регулированию и метрологии РФ 01.03.2012 г. ЖЦ ПС в соответствии с [157] определяется как развитие системы, продукта, услуги, проекта или других

изготовленных человеком объектов, начиная со стадии разработки концепции и заканчивая прекращением применения. Для ПС жизненный цикл определяется как период времени, который начинается с момента принятия решения о необходимости создания ПС и заканчивается в момент ее полного изъятия из эксплуатации. Стандарт определяют понятие процесса как совокупность взаимосвязанных или взаимодействующих видов деятельности, преобразующих входы в выходы. Стандарт группирует различные виды деятельности, которые могут выполняться в течение жизненного цикла программных систем, в семь групп процессов:

- процессы соглашения (два процесса);
- процессы организационного обеспечения проекта (пять процессов);
- процессы проекта (семь процессов);
- технические процессы (одиннадцать процессов);
- процессы реализации программных средств (семь процессов);
- процессы поддержки программных средств (восемь процессов);
- процессы повторного применения программных средств (три процесса).

На рисунке 1.3 приведены процессы ЖЦ, анализируемые в диссертационной работе и их соответствие категориям процессов ЖЦ в соответствии с [157]. Соответствие анализируемых процессов ЖЦ ПС и процессов ЖЦ в соответствии со стандартом:

- процесс разработки включает в себя все процессы ЖЦ, связанные с проектированием ПС, включая процессы проекта, технические процессы и процессы реализации ПС;
- процесс эксплуатации включает в себя процесс функционирования ПС (технические процессы) и процессы верификации и валидации ПС (процессы поддержки);
- процесс модификации включает в себя процесс решения проблем в ПС (процессы поддержки).

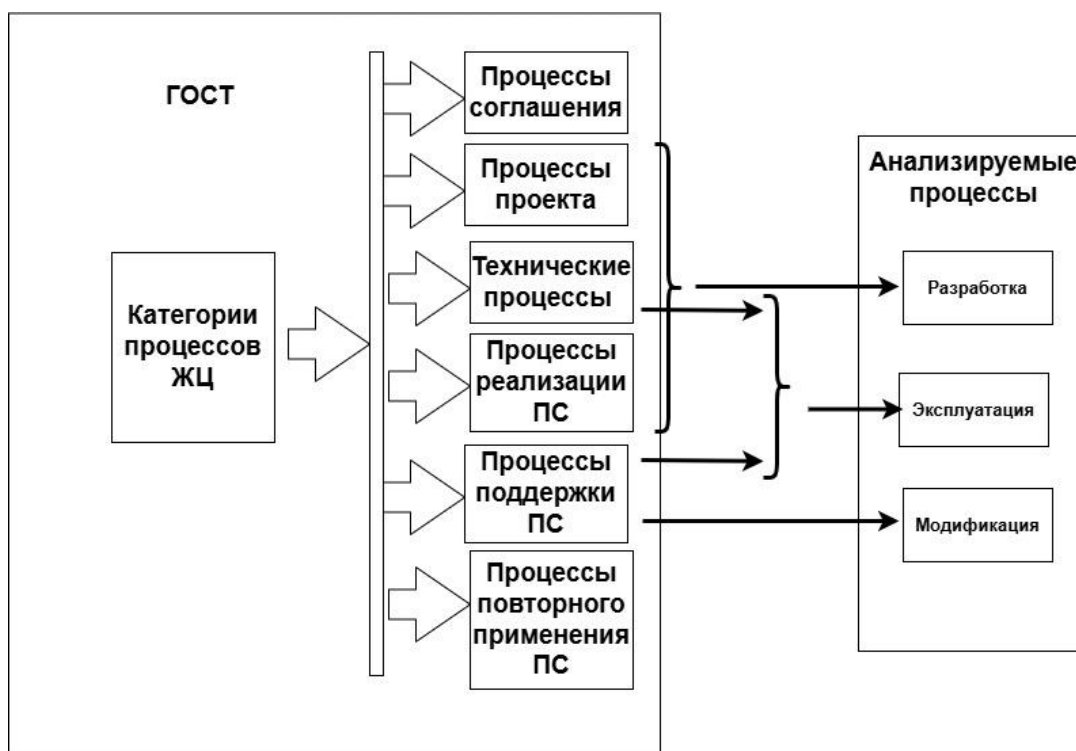


Рисунок 1.3 - Процессы ЖЦ, анализируемые в диссертационной работе и их соответствие категориям процессов ЖЦ в соответствии с ГОСТ

Стандартом определяется модель ЖЦ ПС как структура процессов и действий, связанных с ЖЦ, при этом стандарт не требует использования какой-либо конкретной модели ЖЦ.

1.3 Средства автоматизации проектирования ПС

Для автоматизации проектирования ПС применяются специализированные инструментальные средства (CASE-средства) (рисунок 1.4), поддерживающие процессы создания и сопровождения информационных систем: анализ и формулировка требований, проектирование прикладных ПС и баз данных, генерация кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

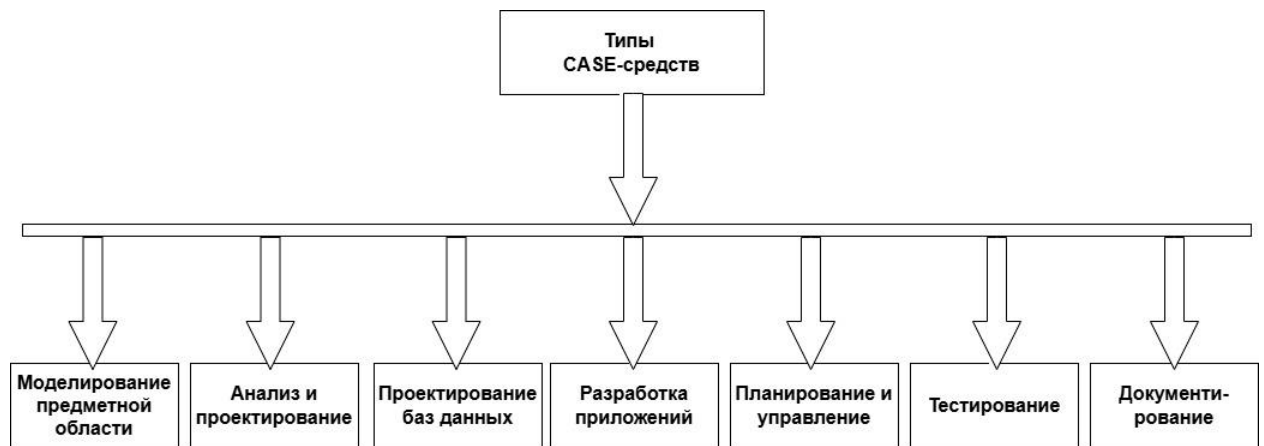


Рисунок 1.4 – Средства автоматизации проектирования ПС

ГОСТ Р ИСО/МЭК 14764-2002 [158] описывает CASE-средства как инструментальные средства сопровождения ПС, определяющее стоимость сопровождения, а также обеспечивающие все аспекты разработки и сопровождения ПС [174]. Полная среда разработки ПС представляет собой совокупность используемых CASE-средств, системного программного обеспечения и технических средств. Конкретная CASE-технология включает в себя методологию проектирования информационных систем и инструментальные средства анализа и моделирования.

1.4 Существующие методологии разработки ПС для решения инженерных задач

Проблема разработки эффективных ПС, а также влияния архитектуры на их эффективность, формирования системы критериев эффективности ПС привлекает внимание широкого круга исследователей. В качестве примера можно привести монографии Д.Паттерсона [187], работы других исследователей. Данной проблеме посвящено большое число публикаций, которые можно разделить на две группы: повышение эффективности ПС путем анализа готовых приложений (эффективность использования различных подсистем компьютера, скорость работы и т.д.) и разработка теоретических подходов к разработке эффективных программ.

В процессе проектирования ПС для ВПВ в области решения инженерных и научных задач крайне важным является выбор лучшего алгоритма решения задачи и максимально глубокая оптимизация кода. Выполнение данных условий позволяет повышать эффективность ПС. При этом методы оптимизации кода зависят от архитектуры целевой ГВС, а лучший алгоритм решения задачи может зависеть от данной архитектуры. Процесс повышения эффективности ПС, как это будет показано далее, носит итеративный характер, поэтому для проектирования ПС для ВПВ используются итеративные методологии проектирования ПС. Проанализируем их с точки зрения проектирования ПС для ВПВ, результат анализа приведен в таблице 1.1.

Таблица 1.1. Итеративные методологии проектирования ПС.

Методология	Свойства, используемые в процессе проектирования ПС для ВПВ
Agile	итеративность
Scrum, RUP	инкрементальность
Lean	эффективность,
Prototype	прототип на базе лучшего алгоритма
RAD	разработка нескольких прототипов на базе нескольких алгоритмов

Agile (гибкая методология разработки) представляет собой совокупность подходов к проектированию ПС, которые ориентированы на использование итеративной разработки, т.е. разработки, в рамках которой осуществляется повторение итераций для доработки результатов, полученных на предыдущем этапе. В процессе проектирования ПС для решения инженерных задач используется именно такой подход, т.к. за одну итерацию в рамках, например, Waterfall, практически невозможно в полной

мере учесть все особенности целевой ГВС и выполнить эффективную оптимизацию кода.

В то же время проектирование подобных ПС часто носит инкрементальный характер, т.к. на каждой следующей итерации осуществляется попытка повысить эффективность проектируемой ПС за счет модификаций алгоритма решения задачи и адаптации его под архитектуру целевой ГВС, а также за счет выполняемых оптимизаций. Таким образом, в процессе разработки ПС для решения инженерных задач используются подходы, реализуемые в таких Agile-методологиях, как Scrum и RUP (Rational Unified Process).

В последние годы актуальным стал вопрос применения концепции бережливого производства в различных областях: от производства до руководства различными трудовыми коллективами. Еще одной методологией проектирования ПС в рамках семейства Agile-подходов является бережливая разработка, Lean, характеризуется принципами бережливого производства. В частности, она ориентирована на исключение потерь, т.е. исключение всего, что не добавляет ценности проектируемой ПС. Нельзя сказать, что в настоящее время разработчики ПС используют подходы, связанные с бережливой разработкой, однако, реализация данных принципов может повысить эффективность проектируемой ПС.

Еще одним итеративным подходом к разработке, который реализуется в задачах проектирования ПС для ВПВ в области решения инженерных и научных задач, является создание прототипа (Prototype). В самом деле, задача выбора или разработки лучшего алгоритма решения задачи, который учитывает особенности архитектуры целевой ГВС и проектируемой ПС, и задача оптимизации полученного кода с применением технологий, учитывающих все подсистемы целевой ГВС, часто разделяются. В том смысле, что один из подходов к проектированию подобных систем заключается в создании прототипа на начальном этапе, в котором

отлаживается реализация выбранного алгоритма, после чего выполняется оптимизация полученного кода.

В случае если для решения задачи могут быть использованы различные алгоритмы, эффективность которых на начальном этапе сложно оценить, могут использоваться элементы методологии быстрой разработки приложений RAD в части быстрого создания нескольких прототипов на базе различных базовых алгоритмов. Кроме того, могут создаваться отдельные прототипы, реализующие различные подходы к оптимизации ПС с дальнейшей оценкой их эффективности и дальнейшим развитием одного или нескольких прототипов.

Таким образом, можно сделать вывод о том, что данные методологии применяются в процессе проектирования ПС для решения инженерных задач. Однако, как это будет показано далее, инструменты, используемые в процессе реализации данных методологий (моделей проектирования), часто не могут быть эффективно использованы для проектирования ПС. Поэтому существует необходимость в разработке нового методологического подхода к проектированию данных ПС.

Анализ готовых приложений осуществляется инструментальными средствами, такими, как, например, Intel Parallel Studio. Данные приложения являются мощными инструментами анализа приложений, однако, они не могут быть применены на начальных этапах разработки программ [100, 123]. В работе [36] проведено систематическое исследование (systematic mapping study) вопросов, связанных с методиками проектирования ПС которые используются для разработки инженерных и научных приложений в области ВПВ и их влиянием на качество получаемых ПС. В работе [133] говорится о том, что «многие ученые и инженеры тратят большую часть своей жизни на написание, отладку и сопровождение программного обеспечения. В результате большинство тратит слишком много времени на борьбу с программным обеспечением, вместо того чтобы заниматься исследованиями, но понятия не имеют, насколько надежны или эффективны их программы». В

работе [54] утверждается, что хотя ученые тратят значительную часть своего времени (более 40%) на создание ПС, они часто не используют в полной мере преимущества достижений в области разработки ПС. Дело в том, что многие ученые, чья специфика работы связана с использованием вычислительной техники, с одной стороны, не могут найти необходимые ПС, и вынуждены разрабатывать их самостоятельно, с другой стороны, ученые не являются высококлассными квалифицированными специалистами в области передовых технологий разработки ПС. Данные факторы входят в противоречие друг с другом и очевидно говорят о необходимости разработки новых подходов к проектированию ПС в области ВПВ, которые, с одной стороны, уменьшают время, затрачиваемое на разработку и модификацию ПС, с другой стороны, снижают требования к квалификации и компетенции разработчиков. Специфика разработки инженерных и научных ПС связана с частыми доработками, модификациями, адаптациями под конкретные архитектуры. При этом инженерные и научные ПС для ВПВ обычно пишутся на С, С++ или FORTRAN, которые обеспечивают высокую производительность, но сложны в разработке и обслуживании, на которое может уходить от 50% до 75% стоимости проекта [60].

В научных работах, посвященных данной тематике, исследуются вопросы, связанные с компромиссами между эффективностью и сложностью в доработке или модернизации, эффективностью и использованием таких практик программирования, как шаблоны проектирования, рефакторинг (в т.ч. ООП) и т.д. В работе [36] проведен подробный анализ упоминания существующих практик программирования на различных этапах ЖЦ ПС для научных исследований, используемого в области ВПВ. Результаты проведенного анализа приведены на рисунках 1.5 и 1.6. Термин «практики программирования» здесь вводится как обобщающее определение для различных методов, технологий, стратегий и приемов проектирования ПС.

На этапе реализации в качестве одной из наиболее упоминаемых практик разработки ПС является «Техники программирования». К данной

категории относятся многочисленные методы разработки ПС, она была выделена исследователями для того, чтобы не анализировать отдельно все многочисленные существующие в настоящее время методы и подходы к проектированию ПС, в частности выполнение различных оптимизаций, векторизация вычислений, и т.д.

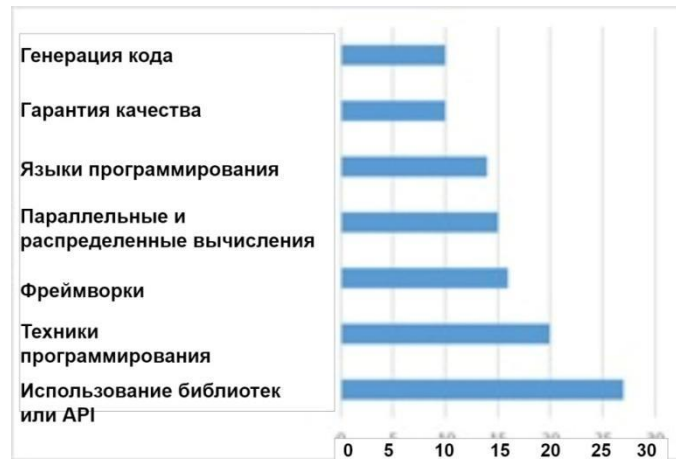


Рисунок 1.5 - Частота упоминания практик разработки ПС на этапе реализации



Рисунок 1.6 - Частота упоминания практик разработки ПС на этапе проектирования архитектуры

На этапе проектирования архитектуры наибольшее внимание исследователи уделяют Component-Based Development (компонентно-ориентированная разработка), которая представляет собой практику или стратегию разработки ПС, в рамках которой для проектирования ПС разрабатываются несвязанные или слабо связанные программные компоненты с акцентом на их повторное использование. Ожидаемо встречается большое упоминание архитектуры ПС в качестве ключевого

фактора, влияющего на характеристики проектируемых ПС. Под архитектурой ПС в данном случае понимается использование различных шаблонов проектирования, техник ООП и т.д. Однако на этапе проектирования архитектуры исследователи также уделяют большое внимание разработке моделей. Причем речь здесь идет не только и не столько о программных моделях, моделях объектной области, проектируемых в процессе разработки ПС, сколько о моделях проектирования самих ПС. При этом стоит отметить, что частота упоминания разработки моделей превышает частоту упоминания архитектуры ПС.

Исследователи выделяют ряд критериев качества ПС, такие как производительность, эффективность, масштабируемость, переносимость, ремонтпригодность и ряд других. На рисунках 1.7 и 1.8 приведена частота упоминания критериев качества ПС на этапах реализации и проектирования архитектуры.

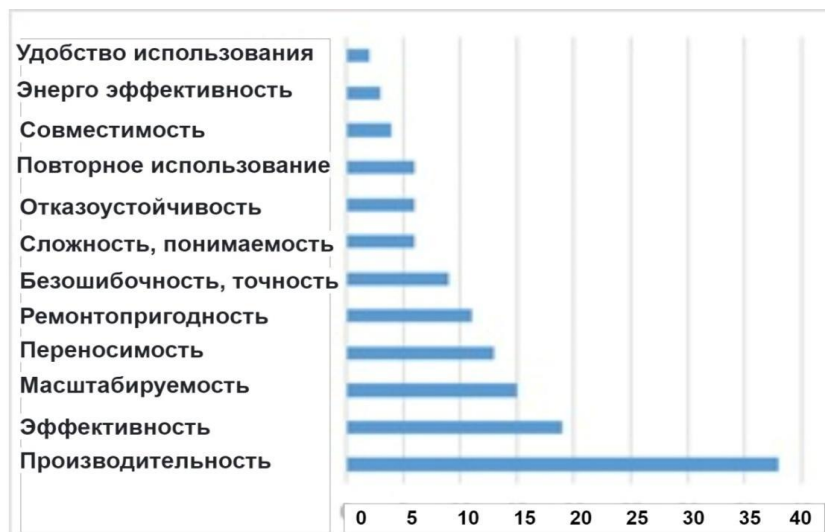


Рисунок 1.7 - Частота упоминания критериев качества ПС на этапе реализации

Видно, что на обоих этапах ведущую роль занимает производительность. Это вполне очевидно, т.к. исследование проводилось для ПС, предназначенных для выполнения инженерных и научных расчетов, ПС, спроектированных для ВПВ. В соответствии с этим, производительность

проектируемых ПС исследователи ставят на первое место. На этапе реализации одну из ведущих ролей играет эффективность, под которой здесь понимается комплексный критерий, включающий в себя степень сложности разработки ПС, а также ремонтпригодность, т.е. приспособленность ПС в т.ч. к внесению изменений и корректировке (изменение алгоритма, оптимизация под конкретную вычислительную архитектуру).

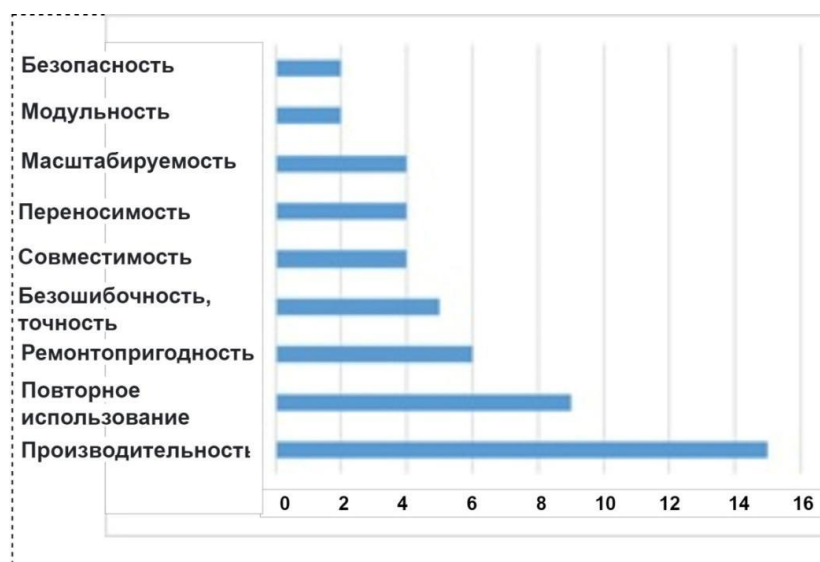


Рисунок 1.8 - Частота упоминания критериев качества ПС на этапе проектирования архитектуры

В таблице 1.2 приводится частота использования различных языков программирования для разработки научных ПС, В таблице 1.3 приводится частота упоминаний различных этапов и аспектов разработки ПС,.

Таблица 1.2. Частота использования различных языков программирования для разработки научных ПС

Язык программирования	%
C++	36,5
C	23
Fortran	15,8
Python	9,9

Таблица 1.3. Частота упоминаний различных этапов и аспектов разработки ПС.

Этапы (аспекты) разработки ПС	%	Этапы (аспекты) разработки ПС	%
Реализация	51,9	Управление проектом	4,6
Проектирование архитектуры	23,1	Поддержка	2,3
Тестирование	16,3	Развертывание	1,8

Видно, что чаще всего исследователи упоминают реализацию и проектирование архитектуры, наиболее популярными языками для разработки являются С, С++ и Fortran. Python в данном случае используется с С-библиотеками. Примерами таких библиотек являются NumPy, SciPy, TensorFlow, PyTorch и т.д.

Подводя предварительный итог, можно сделать следующие выводы.

Исследователи в качестве основных критериев качества выделяют производительность и ремонтпригодность. Под производительностью здесь понимается число задач, решаемых в единицу времени, другими словами, как быстро ПС справляется со сложными инженерными и научными расчетами. Под ремонтпригодностью понимается степень сложности доработки ПС, внесения в нее правок и адаптации под другую вычислительную архитектуры. Также исследователи выделяют важность создания различных моделей как на этапе разработки ПС, так и собственно новых моделей и методов, предназначенных для повышения эффективности процесса проектирования и модификации ПС

В процессе анализа исследователи выделили четыре основных модели процессов проектирования ПС - RUP, OpenUP, ICONIX и Scrum. Все этапы и артефакты, описывающие процесс проектирования (критерии качества,

техники программирования) приводились в соответствии с терминами, используемыми в данных моделях.

Рассмотрим подробнее особенности основных моделей проектирования ПС.

Известен целый ряд различных методологий проектирования ПС для ВС с различной архитектурой, включая ГВС. Выбор конкретной методологии проектирования зависит от ряда факторов, среди которых можно выделить следующие:

- тип ПС;
- архитектура целевой ГВС;
- субъективные предпочтения, включая опыт и уровень компетенций руководителя проекта и членов команды разработчиков.

Сначала рассмотрим наиболее часто используемые методологии проектирования ПС без привязки к типу ПС, далее проведем анализ методологий, реализуемых в процессе проектирования ПС для ВПВ.

Методология проектирования ПС представляет собой совокупность правил, основных задач и ролей, которыми руководствуются в процессе проектирования ПС. Основные цели реализации любой методологии проектирования ПС заключаются в структурировании процесса проектирования, выделения отдельных задач, распределения их по исполнителям, распределение ролей в команде. Методология проектирования предполагает планирование этапа разработки ЖЦ ПС.

Методологии проектирования ПС можно разделить на две группы: методологии, имеющие структуру «шаг за шагом» и итеративные (инкрементальные) методологии/

Классическим примером методологии со структурой «шаг за шагом» является методология Waterfall (водопадная модель разработки) [107]. Водопадная модель разработки — это линейный подход к разработке ПС. Она характеризуется строгим, структурированным процессом. Основной

принцип — каждый этап проекта должен быть завершен до начала следующего. К плюсам данной методологии можно отнести:

- хорошо подходит для проектов со строгими и понятными требованиями;
- может применяться для проектов любого масштаба — от небольших до крупных;
- много внимания уделяется тщательному документированию каждого этапа разработки;
- является простой, четкой и понятной, легко внедряется в рабочий процесс благодаря простым фазам, которые идут строго по порядку.

К минусам методологии можно отнести:

- недостаточная гибкость, после завершения какого-либо этапа сложно вернуться назад и внести изменения, не нарушая последующие этапы;
- сложно применяется в проектах, где нужны итерации и частая обратная связь от владельца продукта;
- предполагает ограниченную вовлеченность владельца продукта, поскольку он участвует в проекте в основном в начале и в конце его реализации;
- фаза тестирования наступает в конце проекта, поэтому проблемы и ошибки могут быть выявлены не сразу, устранять их будет дорого и сложно.

Данная модель не подходит для проектирования ПС, предназначенных для ВПВ, ПС, реализующих научно-технические и инженерные расчеты:

- методология является «однопроходной» отсутствуют итерации, в рамках которых можно повышать эффективность ПС;
- данная методология плохо подходит для проектов, предполагающих частые изменения, доработки и модификации.

В итеративной модели полные требования к системе делятся на различные сборки. Терминология часто используется для описания поэтапной сборки ПС. Имеют место несколько циклов разработки, и вместе они составляют ЖЦ «мульти-водопад». Цикл разделен на более мелкие легко

создаваемые модули. Каждый модуль проходит через фазы определения требований, проектирования, кодирования, внедрения и тестирования. Процедура разработки по инкрементной модели предполагает выпуск на первом большом этапе продукта в базовой функциональности, а затем уже последовательное добавление новых функций, так называемых «инкрементов». Процесс продолжается до тех пор, пока не будет создана полная система. Данная группа методологий больше подходит для проектирования ПС для решения инженерных и научных задач.

Существует большое количество разновидностей итеративной разработки, в рамках которой выполнение работ осуществляется параллельно с анализом полученных результатов и корректировкой дальнейших этапов разработки. Первые идеи итеративной модели разработки были заложены в «спиральной» модели [48], классическими примерами итеративной разработки являются методологии RUP [30], Prototype [109]. В методологии Agile, которая появилась как противовес крайне формальным и тяжеловесным технологиям, после каждой итерации заказчик может наблюдать результат и понимать, удовлетворяет он его или нет [120]. Это одно из преимуществ гибкой модели. К ее недостаткам относят то, что из-за отсутствия конкретных формулировок результатов сложно оценить трудозатраты и стоимость, требуемые на разработку. В рамках Agile проект делится на небольшие забеги («спринты»), продолжительность которых обычно составляет 2–4 недели, но может меняться в зависимости от типа и сложности проекта. Agile-разработка подходит для проектов, требующих гибкости и постоянных изменений и доработок, в частности для научных и инженерных разработок. Она может стать лучшим выбором для тех случаев, когда важна способность к быстрой адаптации из-за меняющихся требований. Она также идеально подходит для проектов, предполагающих инновации и эксперименты. Известен целый ряд «гибких» методологий разработки ПС, самыми известными из которых являются Extreme Programming (XP), Scrum, Lean.

1.5 Методы анализа ПС

Анализ ПС необходим для выполнения оптимизации ПС, выявления ошибок, дефектов и уязвимостей ПС, связанных с ее надежностью, безопасностью, работоспособностью, корректностью выдаваемых результатов. Различают статический и динамический анализ ПС. Статический анализ выполняется без компиляции и выполнения программы путем анализа исходного кода, динамический анализ проводится во время выполнения программы.

Статический анализ позволяет находить ошибки, дефекты и уязвимости программы на этапе ее разработке. Под ошибкой обычно понимают место в программе, которое может привести к ее аварийному завершению, выдаче некорректных результатов на определенном наборе входных данных. Под дефектом понимают место в программе, которое может ухудшить ее эксплуатационные характеристики. Классический пример дефектов программы – утечки памяти. Под уязвимостью понимают место в программе, которое потенциально может быть использовано для атаки на ПС с целью кражи данных, краха программы и т.д. Все эти проблемы ПС можно объединить в один класс – ошибочные ситуации [147], именно для поиска таких ошибочных ситуаций в исходном коде программы и используется подавляющее большинство современных статических анализаторов.

В процессе проведения статического анализа выполняются три этапа:

- получение модели кода (промежуточное или внутреннее представление);
- дополнение модели кода дополнительной информацией на основе методов статического анализа;
- достижение целей анализа (поиск ошибочных ситуаций).

В качестве модели кода используются абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) [143], граф потока управления (Control

Flow Graph, CFG) [58], используются и другие подходы, например, поток лексем, байт-код и т.д.

Выявление проблем до начала выполнения программы позволяет предотвратить дорогостоящие ошибки и уязвимости в конечном продукте.

К основным методам выполнения статического анализа относятся:

- анализ абстрактного синтаксического дерева (AST);
- использование единого внутреннего представления для анализа различных языков программирования;
- использование специализированных языков запросов к построенному AST [190];
- анализ потока данных (внутрипроцедурный, межпроцедурный, межмодульный);
- построение графа потока управления;
- аннотирование функций;
- символьное выполнение;
- использование SMT-решателей (Satisfiability Modulo Theories).

Анализаторы, основанные на обходе AST, чаще всего основаны на открытых компиляторных инфраструктурах для соответствующих языков программирования. В качестве примера здесь можно привести Clang [178]. Хотя известны анализаторы, основанные на собственных разработках [144]

Стоит отметить, что в общем случае в соответствии с теоремой Райса задача поиска ошибочных ситуаций является неразрешимой. В силу этого анализаторы могут как пропускать ошибочные ситуации, так и генерировать ложные срабатывания.

Также необходимо отметить, что в ходе проведения статического анализа оптимизирующими компиляторами анализируется ряд аспектов эффективности ПС. Данные компиляторы выполняют оптимизирующие преобразования, например, перестановочные оптимизации, получают эквивалентные варианты кода с точки зрения семантики с целью улучшения

показателей эффективности ПС с использованием набора определенных критериев и целей оптимизации.

Динамический анализ ПС проводится путем выполнения программы с использованием специальных инструментальных средств. Потенциально динамический анализ позволяет выявить большее количество ошибок в программе, проанализировать ее эффективность, однако он применяется к уже готовому коду, в который заложены возможные неэффективные проектные решения. К основным методам динамического анализа относятся тестирование и профилирование ПС. Тестирование представляет собой проверку ПС на соответствие заявленным требованиям по результатам ее работы. Профилирование заключается в сборе статистики об анализируемой ПС: время выполнения отдельных функций и блоков кода, подсчет количества кеш-промахов, удачно предсказанных переходов и т.д.

Помимо поиска ошибочных ситуаций, динамический анализ может использоваться для повышения эффективности ПС. Последовательность действий в данном случае следующая.

1. Получить исполняемый код для анализа.
2. Сбор статистики о выполняемом коде, получение необходимых метрик эффективности.
3. Если полученные метрики соответствуют требованиям, предъявляемым к ПС, переход к п.5, иначе к п.4
4. Модификация исходного кода, получение исполняемого модуля.
Переход к п.2
5. Конец.

Очевидно, что переход к следующей итерации цикла улучшения показателей эффективности ПС не гарантирует повышения эффективности ПС, т.к. могут быть сделаны неудачные проектные решения. Однако для оценки качества сделанной итерации необходимо выполнить всю итерацию полностью, т.е. модифицировать код, получить исполняемый модуль, проведение профилирования. Данный подход может снижать эффективность

процесса разработки в связи с возможной необходимостью выполнения неудачных итераций разработки.

Избежать выполнения неудачных итераций можно, если включить анализ эффективности ПС в ранние этапы проектирования, например, в этап разработки исходного кода. Однако, как показано ранее, статический анализ не используется для анализа эффективности ПС, для этих целей используется динамический анализ, который не может быть применен на начальных этапах, т.к. требуется запуск написанной и скомпилированной программы.

При этом анализ эффективности ПС на этапе написания кода может значительно сократить время, затрачиваемое на разработку, т.к. позволит не выполнять компиляцию, отладку и анализ неэффективных версий ПС, в которых на начальном этапе заложены неудачные проектные решения. Нет никаких препятствий для проведения статического анализа эффективности проектируемой ПС путем вычисления соответствующих метрик ПС. Для этих целей необходимо внедрить в системы статического анализа подсистемы поддержки проектирования ПС, позволяющие выбирать проектные решения по повышению эффективности ПС.

Стоит отметить, что статический анализ исходного кода в настоящее время широко используется, существует большое количество инструментальных средств, предназначенных для выполнения статического анализа. Задачей такого анализа часто является поиск ошибок в коде, правильного использования методов, классов, поиск утечек памяти и т.д. Однако в научной литературе практически отсутствуют описания подходов к проектированию ПС, связанных с использованием статического анализа в целях определения показателей эффективности ПС, в частности эффективности использования различных подсистем ВС. Для этих целей используется динамический анализ, который позволяет выявить неэффективное использование различных подсистем ВС. Однако, проведение подобного анализа возможно только для готового приложения, на начальных этапах проектирования проведение такого анализа невозможно.

1.6 Анализ критериев эффективности ПС

Одним из основных показателей качества ПС является их эффективность. Под эффективностью ПС в данном случае понимается совокупность свойств ПС, характеризующая те аспекты ее уровня пригодности, которые связаны с характером и временем использования ресурсов, необходимых для заданных условий функционирования [155]. Другими словами, эффективность ПС определяет отношение уровня услуг, предоставляемых ПС конечному пользователю к объему используемых ресурсов. Показатель эффективности демонстрирует, насколько рационально ПС использует вычислительные ресурсы, имеющиеся в наличии, при этом решая поставленную задачу за приемлемое время.

Критерии эффективности могут отличаться в зависимости от типа ПС и от целей оценки [155, 156, 159]. Существует также большое количество критериев эффективности, не вошедших в официальные стандарты. Например, для мобильных приложений могут быть сформулированы следующие критерии:

- количество скачиваний;
- процент пользователей, которые после скачиваний зашли в приложение снова;
- количество покупок, сделанных в приложении;
- индекс лояльности клиента, т.е. готовность клиента порекомендовать приложение друзьям и знакомым;
- ценность клиента, т.е. доход, который он принесет в процессе использования приложения.

Большинство из приведенных критериев являются чисто экономическими, однако, на значение данных критериев оказывают влияние технические характеристики приложения:

- требования к вычислительным ресурсам;

- производительность приложения, которая влияет на скорость отрисовки графических элементов, например, маршрутов, «зависания» приложения и т.д.

Для игровых приложений могут быть введены дополнительные критерии эффективности:

- динамичность игры, порог вхождения;
- художественная составляющая игры, сюжет, персонажи.

Также широко используются критерии экономической эффективности ПС:

- экономический эффект от внедрения ПС;
- расчетный коэффициент эффективности капитальных затрат на разработку и внедрение ПС;
- срок окупаемости капитальных затрат.

Для ПС, используемых на производстве возможно использование следующих критериев:

- снижение издержек производственно-хозяйственной деятельности на объекте управления в результате разработки и внедрения ПС;
- повышение производительности труда;
- экономия по видам ресурсов;
- высвобождение работников за счет внедрения ПС;
- повышение качества выпускаемой продукции.

Приведенные выше критерии являются критериями эффективности ПС как технического изделия, однако, они не характеризуют ПС как технический объект, подлежащий разработке и модификации.

Особое значение повышение эффективности ПС имеет для ВПВ, в т.ч. в области решения инженерных и научных задач. С данным термином обычно связывают решение задач, связанных с обработкой большого количества данных и выполнением большого числа операций над этими данными. Решение данных задач требует значительных затрат времени, для получения решения за приемлемое время требуется наличие ВС с высокой

производительностью. Здесь также существует ряд критериев, которые используются для оценки эффективности ПС:

- производительность, т.е. способность ПС решать большее количество задач за меньшее количество времени;
- надежность, т.е. способность ПС решать задачи на разных наборах входных данных без отказов и аварийных завершений;
- кроссплатформенность, т.е. способность ПС эффективно решать задачи на ВС с различными характеристиками.

Разработка ПС для ВПВ связана с разработкой программного кода, способного решить исходную задачу и с оптимизацией полученного кода с целью уменьшения времени решения задачи, снижения требований к производительности используемой ВС, что в итоге приводит к уменьшению затрат на эксплуатацию полученной ПС. При разработке ПС других классов (веб-приложений, мобильных приложений и т.д.) на первый план выходят другие техники программирования. Вопросы оптимизации и повышения эффективности здесь связаны с использованием объектно-ориентированного программирования, применением различных паттернов проектирования, использованием фреймворков, библиотек и т.д.

Эффективность любого изделия, в т.ч. ПС, определяется тем, насколько хорошо изделие помогает решать задачи, для которых оно предназначено. Таким образом, эффективность ПС часто определяется возможностью получения результата за приемлемое время. Однако не менее важным критерием эффективности ПС являются ресурсы, затраченные на получение данного результата. Следовательно, эффективность ПС необходимо оценивать комплексно, включая анализ затрат на этапе проектирования ПС и возможных затрат на модификацию ПС.

Под повышением эффективности будем понимать минимизацию любых затрат, связанных с ПС на различных этапах ЖЦ, включая затраты на разработку, модификацию и эксплуатацию ПС а под эффективными ПС те системы, которые среди прочих систем, спроектированных для достижения

аналогичного результата, связаны с минимальными затратами на всех этапах ЖЦ. Эффективность ПС тесно связана с понятием эффективности использования вычислительных ресурсов, имеющихся в наличии, которая определяется степенью загрузки вычислительных ресурсов, участвующих в процессе вычислений. Можно сказать, что под минимизацией затрат понимается либо уменьшение времени решения на сопоставимом по вычислительной мощности оборудовании, либо решение задачи за то же время, но на оборудовании, обладающем меньшей производительностью. Очевидно, что ПС, способная быстрее получить требуемый результат на сопоставимом оборудовании, является более эффективной, ПС, которая требует меньших вычислительных ресурсов при решении задачи за сопоставимое время, является более эффективной, более того, ПС, на проектирование которой затрачено меньше ресурсов (различных затрат), является более эффективной.

В целом под затратами, связанными с ПС понимаются все прямые и косвенные экономические затраты, в т.ч. затраты на покупку и аренду оборудования на всех этапах ЖЦ, затраты на разработку и модификацию на соответствующих этапах ЖЦ, затраты, связанные с временем решения задач на этапе эксплуатации (оборудование, электроэнергия, увеличение сроков реализации проектов).

Базовыми критериями эффективности ПС в данном случае выступают:

- время T_O , необходимое для получения результата в процессе эксплуатации ПС;
- вычислительные ресурсы R_O , необходимые для получения результата за время T_O ;
- время, T_D необходимое для разработки ПС;
- вычислительные ресурсы R_D , необходимые для разработки ПС за время T_D ;
- время, T_M необходимое для модификации ПС;

- вычислительные ресурсы R_M , необходимые для модификации ПС за время T_M ;

На основе базовых критериев эффективности можно выделить несколько дополнительных критериев:

- энергопотребление ВС с запущенной ПС;
- стоимость ПС;
- стоимость поддержки ПС.

Энергопотребление прямо зависит от пары критериев (T_O , R_O), пара критериев (T_D , R_D) является одним из показателей, от которого зависит стоимость ПС, пара критериев (T_M , R_M) является одним из показателей, от которого зависит стоимость поддержки ПС.

1.7 Анализ влияния архитектуры вычислительных систем на эффективность ПС

В последние десятилетия наблюдается неуклонный рост производительности ВС. Данная тенденция традиционно обусловлена двумя факторами.

«Электрический» или «технологический фактор», заключающийся в постоянном улучшении технологии изготовления электронных устройств, в том числе, микропроцессоров и других устройств, выступающих в качестве вычислительных узлов, способных обрабатывать данные под управлением программы. В рамках данного фактора повышения производительности постоянно наблюдаются следующие улучшения электронных изделий:

- повышение рабочей тактовой частоты, приводящее к повышению производительности МП даже при использовании той же архитектуры за счет того, что сокращается время выполнения команд;
- уменьшение геометрических размеров базовых элементов, таких как транзисторы, за счет чего достигается размещение большего числа исполнительных узлов на кристалле; в данном случае также без

значительных изменений архитектуры устройства можно повысить его производительность вследствие размещения на кристалле решающих блоков, которые отсутствовали в предыдущем поколении устройств.

Второй фактор – изменение архитектуры микропроцессоров. Под архитектурой ВС понимают ее абстрактное описание, модель ВС, состоящая из взаимодействующих компонентов. В более широком смысле архитектура понимается как концепция взаимосвязи элементов структуры ВС, которая включает в себя компоненты логической, физической и программной части ВС.

Развернутое описание архитектуры включает в себя целый ряд компонентов:

- представление программ в памяти ВС и правила их интерпретации;
- форматы представления данных, способы их адресации в программе;
- состав аппаратных средств и их характеристики;
- взаимодействие программных и аппаратных средств ВС

В свое время значительное влияние на рост производительности МП оказали следующие архитектурные решения:

- организация очередей и конвейеров команд;
- реализация кеш-памяти конструктивно вместе с МП и разделение кеш-памяти на кеш-память инструкций и данных;
- появление суперскалярной архитектуры;
- появление архитектуры VLIW (Very Large Instruction Word);
- появление SIMD-расширений (Single Instruction Multiply Data);
- появление многоядерных микропроцессоров;
- ряд других технологий.

1.7.1 Иерархическая структура памяти

Кеширование – техника, используемая для повышения производительности ВС за счет хранения копий данных или результатов

вычислений во временной области хранения, известной как кеш-память. Это позволяет быстрее получать доступ к часто используемой информации, сокращая время и ресурсы, необходимые для извлечения данных из более медленных слоев хранения [146]. Данная технология, во-первых, повышает скорость доступа к данным, так как извлечение информации из кеш-памяти занимает значительно меньше времени по сравнению с основными местами хранения, такими как основная память, жесткие диски или сетевые соединения. Во-вторых, использование кеш-памяти повышает эффективность системы, поскольку снижает нагрузку на основные хранилища и сеть, что в конечном итоге улучшает общую производительность. Наконец, использование кеш-памяти также обеспечивает экономическую выгоду, т.к. позволяет оптимизировать использование ресурсов и обрабатывать больше задач с меньшим количеством оборудования, что приводит к снижению затрат.

Кеш-память МП используется для уменьшения времени доступа к основной памяти. Данный результат достигается за счет хранения копий часто используемых данных из основной памяти в небольшой, быстрой памяти (типа Static Random Access Memory, SRAM или Synchronous Dynamic Random Access Memory, SDRAM). Если большая часть запросов в память будет обрабатываться кеш-памятью, средняя задержка обращения к памяти будет приближаться к задержкам работы кеш-памяти. Самая маленькая единица данных, которая может быть передана между кеш-памятью и основной памятью, называется кеш-линия или строка кеш-памяти. [124].

Использование кеш-памяти для повышения эффективности ПС связано с понятиями пространственной и временной локальности [17]. Пространственная локальность предполагает, что обращение к определенным местам хранения часто сопровождается обращениями к близлежащим местам. Например, последовательная обработка элементов массива приводит к тому, что следующим обрабатываемым элементом массива будет являться элемент, расположенный в памяти сразу за текущим

элементом. Если он расположен в той же кеш-линии, то он уже был ранее загружен в кеш-память, и обращаться в основную память необходимости нет. Другими словами, если произошло обращение к данным, расположенным по адресу i , велика вероятность, что следующим обращение будет осуществлено по адресу $(i + 1)$. Временная локальность, в свою очередь, указывает на то, что если к определенным данным был доступ недавно, вероятно, к ним снова будет осуществлен доступ в ближайшем будущем. Событие обращения к данным, которые уже были ранее загружены в кеш-память, называется кеш-попаданием, обращение к данным, отсутствующим в кеш-памяти, называется кеш-промахом.

В настоящее время используются различные технологии, связанные с кешированием данных, применяемые с целью повышения эффективности ВС. К основным технологиям, используемым в подсистемах памяти современных МП, можно отнести следующие.

1. Многоуровневая кеш-память. Современные МП часто используют многоуровневую кеш-память, включающую уровни L1, L2 и L3. В некоторых системах также может присутствовать кеш-память более высоких уровней, например L4, но такие архитектуры используются в настоящий момент редко [37].

Основной принцип работы многоуровневой кеш-памяти – минимизировать задержку при обменах между МП и памятью, получая при этом больший объем буферной памяти для хранения данных.

Кеш первого уровня является самым маленьким и быстрым. Его размер небольшой, несколько десятков килобайт, при этом в серверных решениях размер данного устройства может быть достаточно большим. Кеш L1 делится на две части: одна для хранения инструкций L1i и другая для хранения данных L1d. Данное разделение поддерживает различную полосу пропускания выборки, используемую МП, поскольку большинство программ обычно требует большего объема кеш-памяти для данных, чем для инструкций. Кеш-память L2 больше по размеру и немного медленнее по

скорости по сравнению с L1. L2 может совместно использоваться несколькими ядрами или только одним, в зависимости от архитектуры ЦП. Кеш-память L3 больше, чем L1 и L2, но гораздо более медленная, чаще всего разделяется всеми ядрами. Кеш L3 играет важную роль в совместном использовании данных и в организации связи между ядрами МП.

Многоуровневая кеш-память является ключевым компонентом архитектуры современных МП, позволяющим значительно уменьшить задержки при обращении к основной памяти и повысить общую производительность ВС.

Возможна инклюзивная и эксклюзивная организация многоуровневой кеш-памяти. При инклюзивной организации данные, хранящиеся на уровнях памяти, расположенных ближе к МП, дублируются на более старших уровнях. Например, при считывании данных из ОЗУ, они дублируются во все уровни кеш-памяти. При вытеснении данных из L1 они останутся в L2, при вытеснении из L2 – останутся в L3. Данная политика используется в том случае, если разница в объемах различных уровней кеш-памяти велика. При эксклюзивной организации данные не дублируются в различных уровнях, совокупный объем буферной памяти составляет сумму объемов кеш-памяти всех уровней.

2. Ассоциативная организация кеш-памяти. Задаёт способ отображения строк основной памяти на строки кеш-памяти, который определяет способ хранения и поиска данных в кеш-памяти. Данная характеристика кеш-памяти оказывает большое влияние на скорость доступа к данным и эффективность кеширования [170]. Выделяют три основных типа ассоциативности в кеш-памяти.

Кеш-память с прямым отображением. Представляет собой самый простой подход к отображению памяти, который используется для копирования блока основной памяти в доступную строку кеш-памяти. Данный алгоритм назначает строку памяти определенной строке кеш-памяти напрямую, без дополнительных промежуточных процессов [68]. Если строка

кеш-памяти уже занята, то предыдущая строка удаляется для загрузки новой. При этом адрес ячейки в памяти компьютера разбивается на три составляющих:

- смещение в строке кеш-памяти;
- номер строки кеш-памяти, в которую помещается строка основной памяти, содержащая требуемые данные;
- тег, используемый кеш-памятью для идентификации строки.

Таким образом, по номеру строки в основной памяти кеш-контроллер однозначно определяет строку кеша, в которую надо поместить загружаемые данные. Т.к. строка основной памяти может быть помещена только в одну строку кеш-памяти, данный вариант является самым простым, но и обладает худшей эффективностью среди всех методов отображения, т.к. приводит к генерации большого числа кеш-промахов.

Полностью ассоциативная кеш-память. При таком варианте отображения любая строка основной памяти может быть размещена в любой строке кеш-памяти [43]. Данный подход характеризуется высокой эффективностью, гибкостью и более высокой скоростью по сравнению с прямым отображением.

Кеш-память с множественно-ассоциативным отображением. Данный тип кеш-памяти сочетает в себе достоинства двух предыдущих вариантов, представляя собой компромиссный вариант. В данном случае кеш-память делится на множества, при этом отображения на множества осуществляются в «прямом» режиме, размещение внутри множества – полностью ассоциативное. Название кеш-памяти такого типа определяется числом строк в каждом множестве. Так, например, если множество содержит 2 строки, кеш-память называется 2-входовой (2-way), 4 строки – 4входовой (4-way) и т.д.

В таблице 1.4 приведены оценки вероятности промахов кеш-памяти в различных конфигурациях. В таблице вероятность промаха классифицируется как «Низкая», «Средняя», «Высокая» на основе общих

принципов, поскольку конкретные задачи могут значительно различаться в зависимости от рабочей нагрузки, размера кеш-памяти и архитектуры системы.

Таблица 1.4 – Оценка вероятности кеш-промахов

Тип кеша	Обязательные промахи	Прوماхи по емкости	Конфликтные промахи	Оценка общей вероятности промахов
Прямое отображение	Тип кеш-памяти не оказывает влияния	Тип кеш-памяти не оказывает влияния	Высокая	Высокая
2-way Set Associative			Средняя	Средне-высокая
4-way Set Associative			Низкая	Средняя
8-way Set Associative			Очень низкая	Низко-средняя
Полностью ассоциативный			Отсутствует	Низкая

Таблица отображает, как ассоциативность кеш-памяти влияет на различные типы и общую вероятность промахов. Обязательные или «холодные» промахи остаются неизменными во всех вариантах, поскольку они формируются во время первого доступа к данным. Стоит отметить, что на вероятность обязательных промахов оказывает влияние размер кеш-линии и предвыборка данных. Прوماхи по емкости также остаются довольно стабильными, так как они в большей степени зависят от размера кеш-памяти, чем от его ассоциативности. Ассоциативность в данном случае может оказать влияние только в случае использования данных, не представляющих собой непрерывный массив. Конфликтные промахи уменьшаются с увеличением ассоциативности, ведущей к снижению вероятности промахов в кеш-памяти.

Таким образом, выбор типа ассоциативности кеша зависит от конкретных требований к производительности и стоимости ВС. Баланс между скоростью, стоимостью и сложностью реализации определяет, какой тип ассоциативности будет выбран для конкретной архитектуры. Стоит

отметить, что в подавляющем большинстве современных МП используется кеш-память с множественно-ассоциативной организацией.

3. Предвыборка данных (cache prefetching). Данная техника повышения производительности заключается в асинхронной загрузке данных из основной памяти в кеш-память заранее, до того, как эти данные будут запрошены МП [47]. Это позволяет снизить задержки при обращении к данным, которые не находятся в кеш-памяти, увеличивая общую производительность системы и эффективность ПС.

В большинстве современных МП реализована автоматическая аппаратная предвыборка кода и данных. Следует отметить, что предвыборка кода тесно связана с предсказанием ветвлений, что также является одним из способов повышения эффективности МП. Предвыборка данных позволяет МП заранее загружать данные, которые могут понадобиться программе на следующих этапах работы. МП анализирует, по каким адресам и в каком порядке обращается программа и пытается предсказать, какие данные понадобятся в ближайшем будущем, осуществляет их автоматическую предвыборку в кеш-память. Самым простым и наиболее часто используемым методом предсказания здесь является метод загрузки следующей кеш-линии из основной памяти. Данный метод оказывается очень эффективным при последовательном доступе к памяти. При случайном доступе в память предвыборка не оказывает должного эффекта, при этом может даже оказывать отрицательное влияние, засоряя кеш-память ненужными данными.

Существует также возможность программной предвыборки данных, для этого в МП существуют специальные команды, которые позволяют загрузить данные в необходимый уровень кеш-памяти. Часто такие команды вставляются в код программы компиляторами в процессе оптимизации кода. Команды предвыборки также могут использоваться непосредственно программистами, однако чаще используются соответствующие интринсики (встроенные функции, для которых в компиляторе реализована особая обработка) [53].

В частности, интринсик `__mm_prefetch` является частью векторного расширения SIMD, предложенного компанией Intel [86]. SIMD-инструкции позволяют выполнять операции над несколькими данными одной командой МП, улучшая тем самым производительность обработки данных. `__mm_prefetch` предназначена для предвыборки данных в кеш-память, позволяя разработчику указать, какие данные скоро будут нужны, чтобы они могли быть загружены заранее. Следует отметить, что для получения прироста производительности необходимо правильно подбирать шаг предвыборки, при правильном использовании данная методика является достаточно мощным инструментом повышения эффективности ПС.

4. Раздельные кешы данных и инструкций. Во многих МП кеш-память данных и инструкций разделяются; Это связано с тем, что в процессе обработки данных объем считываемых инструкций и данных может сильно отличаться. Кроме того, подобный подход обладает рядом преимуществ. Так, например, разделенная кеш-память может иметь вдвое большую пропускную способность по сравнению с объединенной кеш-памятью. Использование разделенной кеш-памяти оказывает сильное влияние на производительность МП, имеющих конвейер команд, т.к. доступ к инструкциям и данным можно организовать в одном и том же цикле на различных этапах конвейера. Есть и чисто технологические преимущества от такого подхода, которые заключаются в том, что кеш-память инструкций размещается рядом с блоком выборки инструкций, а кеш-память данных - рядом с блоком памяти. Такой подход сокращает задержки обоих блоков. Кроме того, т.к. объем считываемых инструкций и данных в подавляющем числе случаев сильно отличается, инструкции и данные не будут конфликтовать друг с другом в общей кеш-памяти. Стоит отметить, что разделенная технология используется в современных МП для реализации L1.

Раздельные кеш-памяти данных и инструкций широко используются в современных МП, включая такие архитектуры как x86 и ARM.

Производители, такие как Intel, AMD и Apple, в своих МП используют этот подход для повышения производительности.

5. Различными компаниями и исследовательскими центрами ведутся исследования и разработки для повышения производительности и масштабируемости в современных вычислительных системах [192]. Можно выделить несколько основных направлений: интеллектуальное кеширование (Smart Caching), 3D-стекирование памяти, оптимизация для специализированных задач, гибридные кеш-системы, уменьшение электропотребления.

Интеллектуальное кеширование использует алгоритмы машинного обучения и ИИ для предсказания того, какие данные будут запрошены ПС в ближайшем будущем, чтобы заранее загрузить эти данные в кеш-память. Это позволяет сократить время доступа к данным и увеличить общую производительность системы [134]. Примером могут служить алгоритмы, основанные на нейронных сетях, которые анализируют шаблоны (паттерны) доступа к данным и улучшают стратегии кеширования «на лету» в режиме реального времени. По сути, данная технология представляет собой «умную предвыборку» и может стать существенным фактором в дальнейшем совершенствовании технологии кеширования данных в МП.

3D-стекирование памяти позволяет увеличить плотность хранения данных и пропускную способность, стекируя несколько слоев памяти друг на друга вертикально. Данная технология уменьшает физическое расстояние между МП и памятью, сокращая задержки и повышая скорость передачи данных. Примером служит High Bandwidth Memory (HBM), которая используется в GPU и устройствах для ВПВ, обеспечивая значительно большую пропускную способность по сравнению с традиционной памятью. Стоит отметить, что данная технология разрабатывается многими ведущими ИТ-компаниями, в частности, Intel и AMD.

Гибридные кеш-системы комбинируют различные типы кеш-памяти (например, SRAM, DRAM, Flash) в одной системе, используя их

преимущества для повышения эффективности хранения и доступа к данным [142]. Примером таких систем является использование быстрой SRAM для кеширования наиболее часто используемых данных и более медленной, но емкой Flash-памяти для редко используемых данных.

Разработка энергоэффективных архитектур кеш-памяти включают в себя использование технологий, минимизирующих энергопотребление при доступе к данным и их хранении, например:

- оптимизированные алгоритмы кеширования, минимизирующие число промахов кеш-памяти, что снижает количество энергозатратных операций доступа к основной памяти;

- применение низкоэнергетических технологий памяти, например, Low-Power Double Data Rate (LPDDR) для кеш-памяти низких уровней.

Примером служит технология Intel's Speed Shift, которая позволяет МП более динамично управлять своей частотой и напряжением, оптимизируя энергопотребление в зависимости от текущей нагрузки

Повышение эффективности ПС с учётом характеристик подсистемы памяти ВС является одним из ключевых аспектов достижения максимальной производительности.

1. Локальность данных. Кеш-память работает эффективнее, когда ПС проявляет высокую степень локальности данных, как пространственной, так и временной. Чтобы воспользоваться свойствами пространственной локальности, необходимо обеспечить доступ к данным, размещенным в памяти последовательно [121]. Кроме того, из принципа пространственной локальности вытекает необходимость обработки данных в той последовательности, в которой они расположены в памяти. Например, при разработке ПС с использованием языка программирования C, доступ к массивам осуществляется построчно, в то время как при использовании Fortran доступ осуществляется вдоль столбцов.

Еще одним методом повышения эффективности является блочная реализация алгоритмов. Размеры блоков подбираются таким образом, чтобы

целиком помещаться в кеш-память определенного уровня, чаще всего оптимизация осуществляется под L1. В данном случае в соответствии с принципами пространственной и временной локальности количество кеш-промахов минимизируется.

2. Оптимизация под длину строки кеш-памяти. Необходимо учитывать длину строки кеш-памяти при проектировании структур данных. Для уменьшения числа загрузок кеш-строк и снижения вероятности кеш-промахов, необходимо выравнивать данные по границам кеш-строк. Данной цели может помочь добиться выделение памяти под массивы данных размерами, кратными длине кеш-строк, а также выравнивание статически определяемых и динамически выделяемых массивов данных по границам строк (например, функция `_mm_malloc()`).

3. Компилятор и флаги компилятора. Современные компиляторы могут оптимизировать код с целью повышения эффективности использования кеш-памяти. Для использования этой опции необходимо использовать соответствующие флаги компилятора, такие как `-O2` или `-O3` в GCC. Кроме того, некоторые компиляторы имеют возможность оптимизации кода под целевую архитектуру ВС, например, Intel C++ Compiler. Примером использования флагов могут служить флаги компиляции для GCC [125].

4. Профилирование и бенчмаркинг. Для изучения взаимодействия ПС с кеш-памятью можно воспользоваться инструментами профилирования. Инструментальные средства, такие как `perf` [39] в Linux, встроенные анализаторы производительности в средах разработки (IDE) или специализированные приложения, такие как Intel VTune Amplifier [65], помогают выявить потенциальные проблемы, связанные с кеш-памятью.

5. Привязка потоков. При работе с многопоточными программами используются методы привязки потоков к конкретным ядрам МП. Данная техника позволяет уменьшить вероятность конфликтов в кеш-памяти между потоками.

6. Кеш-оптимизация отдельных задач.

Кеш-оптимизации позволяет существенно повысить эффективность получаемых ПС. Существует большое число публикаций, посвященных оптимизации программных реализаций конкретных вычислительных задач. В качестве примера можно привести многочисленные исследования в области оптимизаций процедур BLAS [14] и LAPACK [4].

В качестве промежуточного итога можно сказать, что характеристики подсистемы памяти имеют существенное влияние на итоговую эффективность ПС, разработчики ПС должны учитывать данные характеристики при проектировании ПС. В первую очередь это касается количества уровней, объема кеш-памяти каждого уровня, а также ассоциативности кеш-памяти всех уровней. В меньшей степени это касается политики инклюзивности или эксклюзивности кеш-памяти.

Стоит также отметить, что не были рассмотрены ряд важных характеристик кеш-памяти, т.к. использовать или учитывать их в процессе разработки ПС крайне затруднительно или просто невозможно. Это касается политики свопинга (сквозная Write Through или обратная Write Back запись) [141], алгоритмов вытеснения (LRU, FIFO и т.д.) [179], протоколов обеспечения когерентности кеш-памяти (MSI, MESI, MOESI и т.д.) [183] и ряда других.

1.7.2 Параллельные вычисления

Под параллельными вычислениями понимают способ организации вычислений, при котором программа разбивается на несколько взаимодействующих единиц, представляющих собой потоки инструкций, выполняющихся одновременно (параллельно) [150]. Существуют различные способы организации параллельных вычислений: вычисления, основанные на множестве параллельных потоков в рамках одного процесса, вычисления, основанные на множестве параллельных процессов.

Основная цель параллельных вычислений – уменьшение времени решения поставленной задачи. Для достижения этой цели необходимо наличие параллельной ВС, имеющей в своем составе несколько ВУ, которые могут работать параллельно. В качестве ВУ могут выступать ядра многоядерного МП, отдельные МП в рамках ВС с архитектурой симметричной многопроцессорности Symmetric Multiprocessing (SMP) или «неравномерного доступа к памяти» Non-Uniform Memory Access (NUMA), отдельных компьютеров в рамках вычислительных кластеров.

Для разработки параллельных программ часто необходимо изменить алгоритм получения решения, который использовался для последовательных вычислений. Для анализа параллельных алгоритмов часто используется понятие идеальной параллельной ВС, которая характеризуется:

- неограниченным числом ВУ;
- неограниченными ресурсами;
- отсутствием накладных расходов.

Для исследования скрытого параллелизма в алгоритмах применяется граф алгоритма на идеальной ВС, представляющий направленный граф, в котором вершины представляют собой операции, дуги – потоки данных. В данном случае высота графа будет представлять собой минимально возможное время решения задачи, а ширина – степень параллелизма или количество операций, которые можно выполнять параллельно. Т.к. на различных этапах алгоритма степень параллелизма может отличаться, применяют характеристику средней степени параллелизма, рассчитываемую как отношение общего числа операций к числу этапов алгоритма. Еще одной важной характеристикой параллельного алгоритма является зернистость, определяемая как размер подзадач, которые могут выполняться параллельно. При этом различают крупнозернистые и мелкозернистые алгоритмы.

Кроме характеристик параллельных алгоритм существует ряд характеристик, относящихся к параллельным программам. По сути, они являются свойствами алгоритмов, реализованных в виде параллельных

программ и запущенных на конкретных ВС. К важнейшим характеристикам параллельных программ можно отнести:

- ускорение; показывает, во сколько раз параллельная программа работает быстрее последовательной программы;
- эффективность; показывает, какую долю максимально возможного ускорения удалось получить на данной параллельной реализации алгоритма;
- масштабируемость; показывает изменение эффективности в зависимости от числа используемых ВУ.

Потенциально распараллеливание может значительно повысить эффективность ПС. Причем чем больше ВУ входит в состав параллельной ВС, тем значительно можно снизить время вычислений, повысив эффективность ПС. Однако далеко не все параллельные программы обладают хорошей масштабируемостью. Существует несколько подходов к оценке зависимости производительности ВС от количества ВУ, и одним из наиболее известных является закон Амдала (1.3)

$$s \leq \frac{1}{(1 - p) + \frac{p}{n}} \quad (1.3)$$

где s – ускорение программы;

p – распараллеливаемая часть программы;

n – количество ВУ.

Основным следствием из закона Амдала является утверждение, что максимально возможное ускорение программы ограничивается ее последовательной (не подлежащей распараллеливанию) частью. В самом деле, при неограниченном росте количества числа вычислительных узлов выражение (1.3) примет вид (1.4).

$$s \leq \frac{1}{(1 - p)} \quad (1.4)$$

На рисунке 1.9 приведены зависимости получаемого ускорения в зависимости от доли распараллеливаемых вычислений в соответствии с (1.3) (f – нераспараллеливаемая часть программы).

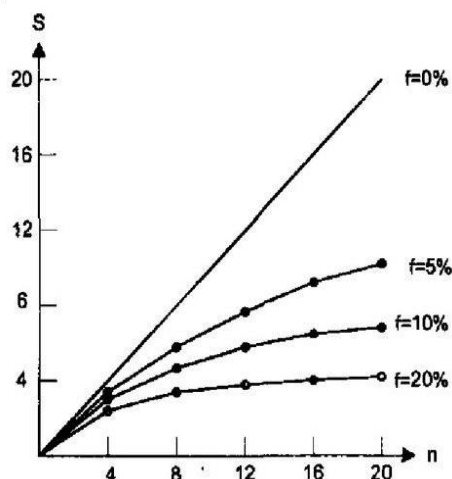


Рисунок 1.9 - Зависимости получаемого ускорения в зависимости от доли распараллеливаемых вычислений в соответствии с законом Амдала [185]

Видно, что при неограниченном росте количества вычислительных узлов ускорение программы перестает расти, при этом эффективность параллельной программы начинает снижаться, асимптотически приближаясь к нулю.

Распараллеливание вычислений является мощным средством повышения эффективности ПС, однако, при этом можно выделить несколько проблем, связанных с распараллеливанием:

- распараллеливание алгоритма решения задачи;
- создание параллельной программы;
- перенос параллельной программы на параллельную ВС.

1.7.3 Распределенные вычисления

Распределенные вычисления являются одним из способов распараллеливания, при котором для решения одной задачи объединяются несколько компьютеров, соединенных при помощи коммуникационной среды, в роли которой чаще всего выступает локальная вычислительная сеть. Хотя существуют и другие варианты, например grid-системы,

представляющие собой слабосвязанные ГВС с высокой степенью распределения, которые часто выделяют в отдельный класс ВС.

Классическим примером распределенных вычислений является вычислительный кластер, который может быть гетерогенным, т.е. включать ВУ с различной архитектурой и производительностью. Данные ВС представляют собой класс параллельных ВС с распределенной памятью в отличие от систем с общей памятью, примерами которых являются ВС на базе многоядерных МП, SMP-систем и т.д. (рисунок 1.10).

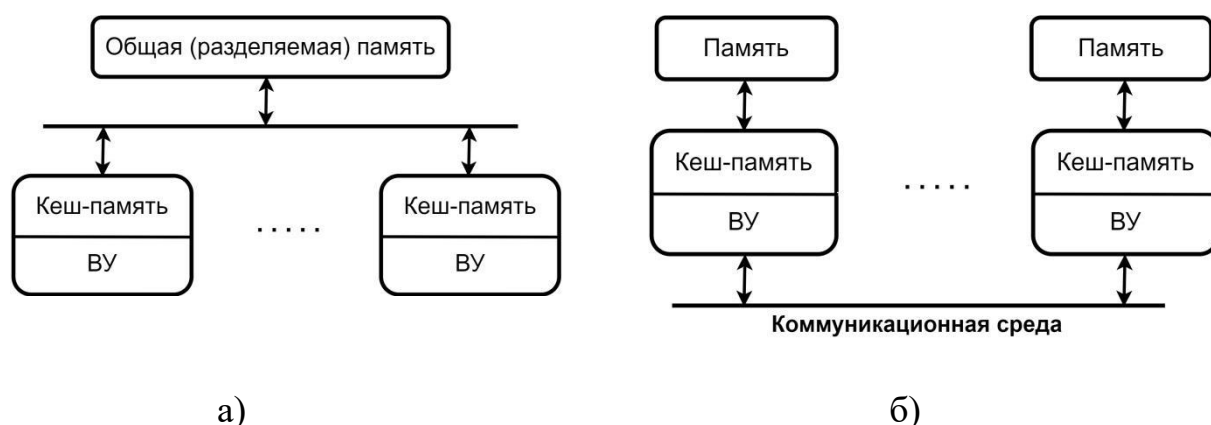


Рисунок 1.10 – Параллельные ВС с общей (а) и распределенной (б) памятью

Преимущества и недостатки параллельных ВС приведены в таблице 1.5.

Анализируя приведенные данные можно сделать выводы, что ВС с общей памятью имеют ряд преимуществ. Из-за особенностей архитектуры обмен данными осуществляется естественным образом через общую память, что значительно упрощает процесс разработки программ, снижает время, затрачиваемое на разработку. Однако подобные ВС обладают одним существенным недостатком – плохой масштабируемостью.

Несмотря на это, ВС с общей памятью являются самым многочисленным классом параллельных ВС: современные персональные компьютеры, смартфоны и множество других устройств имеют подобную архитектуру.

Системы с распределенной памятью в силу своей хорошей масштабируемости применяются при построении высокопроизводительных

систем, систем обработки больших данных, применяются для создания и обучения нейронных сетей и в целом в системах ИИ, в других областях, где производительности систем с общей памятью оказывается недостаточно.

Таблица 1.5. Параллельные ВС.

Тип ВС	Достоинства	Недостатки
Параллельная ВС с общей памятью	- не требуется обмен данными; все ВУ имеют доступ в любую точку адресного пространства, обмен с использованием системы «почтовых ящиков» в общей памяти - простота разработки программ - компактность систем	- проблема совместного доступа в память; - относительно медленное обращение к памяти; - плохая масштабируемость; - проблема когерентности кеш-памяти (необходимость синхронизации кеш-памяти ВУ)
Параллельная ВС с распределенной памятью	- практически неограниченное наращивание производительности за счет увеличения количества ВУ (хорошая масштабируемость)	- обмен данным посредством передачи сообщений; - относительная сложность разработки программ

Работы по организации распределенных вычислений ведутся с 70- годов 20 века. Стоит отметить работы Джона Шоха и Джона Хаппа (локальная сеть PARC) [111], исследования, проводимые на кафедре вычислительной техники НЭТИ (НГТУ, Новосибирск) по созданию распределённой вычислительной системы на ЭВМ «Минск-32») [194], работы В.М.Глушкова (ЕС-2701). Стоит также отметить работы по организации массовых вычислений: Дэвид Джиди (SETI@Home) [5], проект GIMPS [127] и ряд других. В настоящее время на базе ВС с распределенной

памятью созданы суперкомпьютеры с производительностью, превышающей 1 эксафлопс [140].

Помимо кластерных вычислений и grid-систем можно выделить еще ряд типов распределенных архитектур:

- клиент-сервер;
- одноранговая архитектура (peer-to-peer) ;
- облачные распределенные вычисления.

Распределенные вычисления обладают рядом преимуществ:

- высокая производительность;
- высокая масштабируемость;
- отказоустойчивость;
- гибкость.

Существует большое количество платформ для организации распределенных вычислений. Например, Apache Hadoop [131] и Spark [20] являются популярными платформами распределенных вычислений с открытым исходным кодом, используемыми для обработки больших данных.

Существует ряд библиотек, используемых при разработке ПС для вычислительных кластеров, например библиотека Message Passing Interface (MPI), первоначально разработанная для программ на языках C и Fortran, в настоящее время существуют реализации для других языков программирования, например, Java, Python. MPI реализует концепцию Single Program Multiple Data (SPMD), являющуюся частным случаем архитектуры Multiple Instruction Multiple Data (MIMD).

Несмотря на очевидные достоинства распределенных вычислений можно выделить недостатки подобного подхода:

- необходимость балансировки нагрузки, особенно в случаях использования ГВС;
- проблемы с переносимостью в распределенных ГВС;
- не все алгоритмы подходят для распределенной обработки.

Также к проблемам распределенных вычислений можно отнести так называемые «заблуждения о распределенных вычислениях», сформулированные Питером Дойчем (L Peter Deutsch) и другими из компании Sun Microsystems [105]:

- сеть надёжна;
- задержка равна нулю;
- пропускная способность бесконечна;
- сеть защищена;
- топология не меняется;
- есть один администратор;
- транспортные расходы равны нулю;
- сеть однородна.

Еще одной проблемой распределенных вычислений является доступность соответствующих ВС, в противовес ВС с общей памятью, которые в настоящее время доступны практически всем.

Несмотря на приведенные проблемы и недостатки, именно модель распределенных вычислений дала возможность создать наиболее мощные суперкомпьютеры, решать самые сложные задачи современности, дала новый импульс развитию технологий искусственного интеллекта.

1.7.4 Векторизация вычислений

Под векторными командами МП понимается применение некоторой операции к набору данных, например, секции массива. Во многих алгоритмах можно выделить некоторую повторяющуюся модель доступа к данным, а также операции, которые могут выполняться над элементами из этого набора. С математической точки зрения данные операции представляют собой операции над некоторыми векторами. Скалярные вычисления выполняют операции над отдельными числами, в то время как векторизация представляет собой один из способов распараллеливания кода, в рамках

которого одна операция осуществляется над несколькими значениями одновременно. Использование таких векторных операций позволяет существенно повысить эффективность целого ряда программ.

Векторизацией вычислений исследователи занимаются достаточно давно, и за это время было создано большое количество ВС, где так или иначе была реализована векторизация вычислений [104]. Остановимся на технологиях векторизации, реализованных в МП Intel. Хотя нельзя не сказать, что существуют технологии других компаний, например Altivec [27], реализованная в МП компаний Motorola и IBM.

SIMD-технологии компании Intel появились в результате развития встроенного математического сопроцессора FPU. Данное устройство предназначено для выполнения эффективных вычислений с использованием вещественных чисел с плавающей запятой. Первые сопроцессоры (x87) выпускались в виде отдельного чипа, который взаимодействовал с центральным процессором CPU. Сопроцессор извлекал из потока команд свои команды и выполнял их. Начиная с процессора i80486DX, данное устройство было интегрировано в состав CPU. Для хранения чисел с плавающей запятой в МП были добавлены 8 80-битных регистра.

В МП Pentium MMX, который компания Intel представила в 1997 году, появилась новая технология MMX, которая предоставила возможность векторной обработки целочисленных данных. В программно-логической модели МП появились 8 64-битных регистра MM0-MM7, представлявшие собой векторные регистры, т.е. регистры, хранящие несколько целых чисел (количество чисел зависит от их разрядности, совокупная длина вектора представляет собой 64 разряда). Кроме того, в систему команд МП было добавлено расширение MMX, которое представляет собой набор SIMD-инструкций. Но в данном МП не было возможности одновременной обработки целочисленных векторных данных и чисел с плавающей запятой. Это обусловлено тем, что регистры MM0-MM7 адресовались к регистрам FPU, физически это были одни и те же регистры. Свое название MMX

(Multimedia Extensions) технология получила из-за того, что она была разработана для выполнения операций над аудио и видео данными. Однако данная технология могла использоваться во всех областях, где требовались векторные вычисления.

В дальнейшем технология векторизации развивалась в сторону увеличения разрядности векторов, расширения количества форматов поддерживаемых данных и увеличения числа команд.

Следующая SIMD-технология компании Intel, SSE (Streaming SIMD Extensions) [115], появилась в 1999 году. Вектора в SSE имеют разрядность 128 бит и обеспечивают совместимость с MMX. Основным отличием и преимуществом SSE является поддержка чисел с плавающей запятой. Появилась возможность обрабатывать вектора, содержащие числа с плавающей запятой с одинарной (int32) и двойной точностью (int64). При этом устройства SIMD и FPU стали независимы друг от друга и физически использовали различные регистры, что дало возможность работать с командами SIMD и FPU одновременно. Далее технология SSE развивалась, что привело к существенному расширению системы команд. Начиная с SSE2, появилась возможность выполнять операции как со скалярными (scalar single (ss), scalar double (sd)), так и с упакованными (packed single (ps), packed double (pd)) данными, в SSE3 появилась возможность горизонтальной работы с регистрами.

Вместе с микроархитектурой Sandy Bridge Intel предложила новую технологию SIMD - AVX (Advanced Vector Extension) [80]. Разрядность векторов увеличилась до 256 бит, появились новые операции обработки данных. При этом осталась обратная совместимость с технологиями SSE и MMX. AVX-512 расширяет технологию AVX до векторов длиной 512 бит [81, 82].

На рисунке 1.11 приведен пример упаковки данных в SIMD-регистры для вещественных типов данных языка программирования C++.

SSE (128 бит)

<code>__mm128d</code>	double		double		64 бита
<code>__mm128</code>	float	float	float	float	32 бита
<code>__mm128i</code>	int32	int32	int32	int32	32 бита

AVX (256 бит)

<code>__mm256d</code>	double		double		double		double		64 бита
<code>__mm256</code>	float	float	float	float	float	float	float	float	32 бита

Рисунок 1.11 - Размещение данных в векторных регистрах МП Intel

В качестве примера приведены регистры SSE (128 бит) и AVX (256 бит). В регистрах SSE могут размещаться как целочисленные векторы, так и векторы, содержащие вещественные числа. Кроме целочисленного формата `int32`, указанного на рисунке 1.11, поддерживаются также форматы `int64`, `int16` и `int8`. Целочисленные форматы в AVX поддерживаются, начиная с версии AVX2. Слева на рисунке 1.11 указаны типы данных языка программирования C++, которые могут быть использованы для доступа к векторам в соответствии с использованной технологией векторизации и выбранным типом данных.

Операндами векторных инструкций могут быть как сами SIMD-регистры, так и вектора, расположенные в памяти. Для эффективной загрузки этих векторов в SIMD-регистры, их адреса в памяти должны удовлетворять определенным условиям: для 128-битных операций адрес должен быть кратен 16, для 256 – кратен 32, для 512 – кратен 64. Если адреса удовлетворяют этим условиям, говорят, что они являются выровненными. Существует возможность работы и с невыровненными данными, однако это приводит к существенным потерям производительности.

В процессе разработки ПС используются несколько способов применения технологии векторизации вычислений (рисунок 1.12).

Самым высокоуровневым способом векторизации является использование оптимизирующих компиляторов, выполняющих

автоматическую векторизацию кода самым низкоуровневым – непосредственное использование машинных инструкций.



Рисунок 1.12 – Способы векторизации

Проиллюстрируем сказанное на примере программы умножения матриц (таблица 1.6).

В таблице 1.6 представлен код внутреннего цикла программы умножения матриц в классическом варианте (скалярное произведение строки исходной матрицы *a* и столбца исходной матрицы *b*).

В первой строке таблицы 1.6 представлен код цикла, в котором нет указаний на необходимость векторизации. Современные компиляторы могут выполнять автоматическую векторизацию вычислений, т. е. преобразовывать программы из скалярного представления к векторной реализации [90]. При этом компилятор должен провести анализ кода.

Допустимость векторизации. Компилятор должен распознать цикл, выделить информационные зависимости и убедиться, что при проведении векторизации не меняется порядок вычислений, зависящих друг от друга. Другими словами, результат выполнения векторизованной программы не должен отличаться от результата выполнения последовательной программы.

Эффективность векторизации. Компилятор содержит блок эвристики, который оценивает эффективность получаемого кода. Векторизация

проводится только в том случае, если производительность приложения увеличивается.

Таблица 1.6 - Способы векторизации вычислений

Способ векторизации	Пример кода
Автоматическая векторизация компилятором	<pre>for(int k = 0; k < N; k++) c[i * N + j] = c[i * N + j] + a[i * N + k] * b[k * N + j];</pre>
Директивы компилятора	<pre>#pragma vector aligned for(int k = 0; k < N; k++) c[i * N + j] = c[i * N + j] + a[i * N + k] * b[k * N + j];</pre>
Интринсики (*)	<pre>for(int k = 0; k < MAX_EL; k += 2) { Aij = _mm_load_pd((double const *)&a[i_N + k]); int k_N_j = k * N + j; Bij = _mm_load_sd((double const *)&b[k_N_j]); temp_bij = _mm_load_sd((double const *)&b[k_N_j + N]); Bij = _mm_shuffle_pd(Bij, temp_bij, 0); Aij = _mm_mul_pd(Aij, Bij); Cij = _mm_add_pd(Cij, Aij); }</pre>
Машинные инструкции (ассемблер)	<pre>loop_k: movapd xmm0, xmmword ptr [ESI] add ESI, FIRST_MATR_INSIDE_VECTOR movsd xmm1, [EBX] add EBX, SECOND_MATR_INSIDE_VECTOR movsd xmm2, [EBX] add EBX, SECOND_MATR_INSIDE_VECTOR shufpd xmm1, xmm2, 0 mulpd xmm0, xmm1 haddpd xmm7, xmm0 sub EAX, 2 ja loop_k</pre>
Примечание: *интринсики – специальные функции, для которых в компиляторе реализована особая обработка	

Основными факторами, влияющими на эффективность, векторизации являются:

- способ доступа к данным в цикле; наиболее эффективно векторизуется последовательный доступ к данным, доступ с промежутками

векторизуется хуже, т.к. приходится использовать больше операций загрузки и сохранения данных или использовать операции сбора (gather) и распределения (scatter) данных;

- выравнивание данных, связанное с местоположением векторизуемых данных в памяти.

Для эффективной загрузки данных из памяти, адреса этих данных должны быть выровненными. Работа с невыровненными данными приводит к снижению производительности. При этом степень снижения производительности зависит от типа МП, однако, работа с выровненными данными в современных МП в любом случае оказывается более эффективной.

В общем случае данные в памяти расположены по невыровненным адресам. Даже если располагать массивы по адресам, удовлетворяющим требованиям, описанным выше, алгоритмы обработки данных могут использовать только часть или части массива. При этом доступ может осуществляться с невыровненного адреса, что потенциально может приводить к потере производительности [162].

В основном автоматическая векторизация применяется для циклов, используется цикловая перестановочная оптимизация. Из скалярных данных формируются вектора, объединяя несколько итераций цикла, к полученным векторам применяется векторная операция. Такой подход позволяет применять одну инструкцию к набору данных, распараллеливая вычисления.

Процессом автоматической векторизации можно управлять с помощью ключей компиляции. В качестве примера приведем несколько ключей Intel C++ Compiler [61]:

- xAVX – генерация кода с использованием технологий AVX, различных версий SSE;

- xSSE4.2 – генерация кода с использованием технологии SSE4.2 и более ранних версий SSE;

-по-вес – запретить векторизацию, выполнить генерацию кода без векторизирующих инструкций.

Компилятор не всегда может выполнить векторизацию кода или сделать это эффективно. В этих случаях разработчику необходимо предпринимать определенные шаги. Самым высокоуровневым способом здесь является использование директив компилятора [167]. Данные директивы сообщают компилятору информацию, которая неизвестна ему в общем случае и мешает эффективной векторизации. Пример использования одной из таких директив показан во второй строке таблице 1.6. Приведенная в качестве примера директива (`#pragma vector aligned`) сообщает компилятору, что в процессе генерации кода можно безопасно использовать инструкции обращения к выровненным данным. Каждый компилятор может иметь свой набор поддерживаемых директив, с их перечнем можно ознакомиться в документации к компилятору. С целью выравнивания данных разработчику необходимо использовать соответствующие средства языка программирования C++. Для выравнивания динамически выделяемых блоков данных используются специальные функции, например `_mm_malloc()`. Для выравнивания статически определяемых переменных используются специальные описатели, например, `__declspec(align)` или `alignas()`. Перечень функций и описателей зависит от используемого компилятора, а также может зависеть от операционной системы в случае применения переносимых программных средств.

Использование директив компилятора не во всех случаях может помочь создать эффективный векторизованный код. Например, наличие зависимостей по данным приводит к тому, что компилятор не векторизует исходный код. Мощным инструментом анализа в этом случае могут стать отчеты компилятора, в которых содержится информация об удачной векторизации или об отказе от векторизации и причинах, вызвавших этот отказ [33].

В любом случае, в таких ситуациях разработчик должен использовать низкоуровневые способы векторизации, выполнять векторизацию «вручную».

Здесь возможны два способа:

- непосредственное использование машинных инструкций (в последней строке таблице 1.6 приведен пример использования таких инструкций для 32-разрядной архитектуры процессоров Intel);
- использование интринсиков [91].

Применительно к векторным командам МП интринсики являются С-обертками над этими командами. Если компилятор встречает интринсик, не генерируется никаких команд вызова подпрограммы. Вместо этого в программу вставляется фрагмент кода, характерный для конкретной функции-интринсика. Поэтому с точки зрения «низкоуровневого» программирования использование интринсиков является более предпочтительным по отношению к использованию машинных инструкций, так как программа пишется на языке С++ без использования языка ассемблера.

Однако здесь есть нюансы. Интринсики работают со специальными типами данных, которые отображаются на векторные регистры (используются различные типы данных и интринсики для различных технологий векторизации). Но то, где будут расположены переменные указанных типов – в регистрах или в памяти – остается на усмотрения компилятора, при использовании машинных команд разработчик сам контролирует этот процесс.

Векторизация является мощным инструментом, позволяющим значительно повысить эффективность ПС. Оптимизирующие компиляторы с блоком автоматической векторизации часто позволяют получить эффективную ПС, в то же время в ряде случаев необходимо вмешательство разработчика, что делает процесс проектирования ПС сложным и трудоемким в связи:

- с необходимостью модернизации алгоритма решения задачи, изменения способа (паттерна) доступа к данным;
- с необходимостью использования низкоуровневых способов векторизации.

Необходимо также отметить специализированные векторные сопроцессоры, предназначенные для ВПВ в области научно-технических и инженерных расчетов. Так, в НИИСИ РАН разрабатывается семейство высокопроизводительных микропроцессоров КОМДИВ [148], в которых в качестве расширения базовой архитектуры предлагаются специализированные сопроцессоры, в частности векторный сопроцессор. Он содержит 64 128-битных регистра, в котором можно хранить числа в одном из следующих форматов:

- 1 комплексное число двойной точности;
- 2 комплексных числа одинарной точности;
- 2 вещественных числа двойной точности;
- 4 вещественных числа одинарной точности.

Данный сопроцессор разработан для функционирования в составе высокопроизводительных микропроцессоров (1890BM8) семейства КОМДИВ [149]

1.7.5 Использование ускорителей

Одним из способов повышения производительности ВС является специализация вычислений и аппаратная реализация сложных функций. Например, МП Intel8080 не имел команд умножения и деления, реализация данных операций осуществлялась программным способом путем выполнения последовательности сложений и сдвигов. Данное обстоятельство оказывало негативное влияние на эффективность программ, интенсивно использующих подобные арифметические операции. В МП Intel8086 операции умножения и деления появились, однако, умножение 16-битного регистра на число,

хранящееся в ячейке памяти, занимало более 100 тактов. В последующих поколениях МП число тактов, необходимое для выполнения операции умножения неуклонно снижалось. Это происходило за счет переноса логики умножения с программной и микропрограммной реализации на аппаратную реализацию.

Другой пример – выполнение операций над числами с плавающей запятой. Intel8086 не имел таких возможностей, однако, было спроектировано специальное устройство, математический сопроцессор Intel8087, которое выполняло данные операции достаточно эффективно. Впоследствии данное устройство было интегрировано в МП в виде FPU.

Приведенные выше примеры иллюстрируют реализацию принципа аппаратного ускорения, под которым понимают применение специальных устройств, отдельных или в составе МП, которые выполняют определенные операции быстрее, чем МП общего назначения в процессе выполнения программы. С тех пор было спроектировано большое количество различных ускорителей. Повышение производительности в аппаратных ускорителях осуществляется в основном за счет распараллеливания вычислений, снижения накладных расходов, аппаратной реализации сложных функций. К основным типам ускорителей можно отнести GPU, FPGA и ASIC ускорители, ускорители вычислений, построенные на многоядерных архитектурах, такие как ClearSpeed (компания ClearSpeed Technology) [76], сопроцессоры с архитектурой Intel Many Integrated Core (MIC), примером которых является Intel Xeon Phi [67] и ряд других. В последние годы развитие получили ускорители, предназначенные для решения задач ИИ.

На определенном этапе развития компьютеров для обработки вычислительных задач, связанных с компьютерной графикой, были разработаны специальные устройства – графические карты, которые впоследствии получили название GPU. Развитие данных устройств привело к тому, что по производительности они обогнали центральный МП CPU. Процессор, расположенный в GPU выполняет большее количество операций

с плавающей точкой, пропускная способность подсистемы памяти GPU значительно превосходит пропускную способность подсистемы памяти CPU. Это стало возможным благодаря архитектуре массового параллелизма, который применяется в GPU. В результате появилась технология GPGPU, в рамках которой GPU используются для математических вычислений. Для задач, которым присущ внутренний массовый параллелизм, использование GPU дает большой прирост производительности.

К проблемам использования GPU в качестве ускорителя вычислений можно отнести:

- прирост производительности только на определенном классе задач;
- сложность программирования.

Также распространенным типом аппаратных ускорителей являются FPGA-ускорители [145]. Хотя FPGA можно успешно использовать в качестве базового элемента вычислительной системы, часто они используются как сопроцессоры для ускорения вычислений путём переноса их с CPU. Процесс отладки и реализации алгоритмов для архитектуры FPGA более трудоемкий, требует наличия знаний в области схемотехники и дополнительных временных затрат по сравнению с другими архитектурами, например, CPU и GPU.

Среди отечественных решений такого класса выделяются разработки НИИ МВС ЮФУ [176, 180]. Работы по созданию масштабируемых реконфигурируемых вычислительных решений, комплексному обеспечению их применения и собственно внедрению в промышленность выполняются в НИИ МВС в течение уже нескольких десятков лет. За эти годы разработано и внедрено большое количество реконфигурируемых систем на базе различных поколений FPGA, создано развитое специализированное программное обеспечение реконфигурируемых ВС и методика его применения при решении прикладных задач. FPGA являются реконфигурируемыми устройствами. Это означает, что с помощью специализированных

программных и аппаратных средств на базе FPGA можно спроектировать устройства, предназначенные для решения различных задач.

К проблемам использования FPGA-ускорителей можно отнести:

- доступность подобных устройств;
- сложность сопряжения с вычислительными системами;
- сложность создания специализированных схем для решения конкретных задач.

ASIC-ускорители представляют собой специализированные аппаратные ускорители [88]. Хотя существуют ASIC-устройства, позволяющие выполнять определенные настройки, по гибкости они значительно уступают FPGA-ускорителям. По сравнению с FPGA ASIC обладают следующими преимуществами:

- более высокая производительность;
- энергоэффективность;
- компактностью;
- возможность крупносерийного производства;

К недостаткам ASIC по сравнению с FPGA кроме ограниченной гибкости можно отнести:

- высокие затраты на проектирование и производство мелких серий или единичных устройств;
- сложный процесс разработки;
- риск устаревания.

Общей проблемой применения ускорителей является балансировка нагрузки между CPU и ускорителями различных типов при их одновременном использовании.

1.8 Целевая вычислительная архитектура

Под целевой вычислительной архитектурой понимается архитектура ВС, в которой реализуются предлагаемые модели и методы проектирования

ЭПС. В качестве целевой вычислительной архитектуры в данной работе рассматривается ГВС, где под ГВС понимается ВС, содержащая несколько вычислительных ресурсов, причем множества операций, выполняемых различными ресурсами, различны (т.е. каждый ресурс имеет свою функциональную специализацию) [175]. С точки зрения проектирования ПС под гетерогенностью будем понимать программирование различных устройств, имеющих собственные системы команд.

Методологию проектирования ПС предлагается реализовать в ГВС, имеющей в своем составе следующие вычислительные ресурсы:

- вычислительная система с общей памятью, построенной по иерархической схеме (CPU);
- векторный процессор или векторное расширение команд (SIMD);
- графический процессор (GPU).

Каждый из ресурсов обладает набором параметров

Вычислительная система с общей памятью, построенной по иерархической схеме, реализованной в виде совокупности основной памяти и многоуровневой кеш-памяти, обладающей определенными характеристиками. Каждый уровень кеш-памяти характеризуется следующими параметрами:

- тип – множественно – ассоциативный;
- V – объем;
- M – степень ассоциативности;
- L – длина кеш-линии.

Рассматривается множественно-ассоциативная кеш-память (рисунок 1.13), т.к. кеш-память прямого отображения и полностью ассоциативная кеш-память являются частными случаями множественно-ассоциативной кеш-памяти.

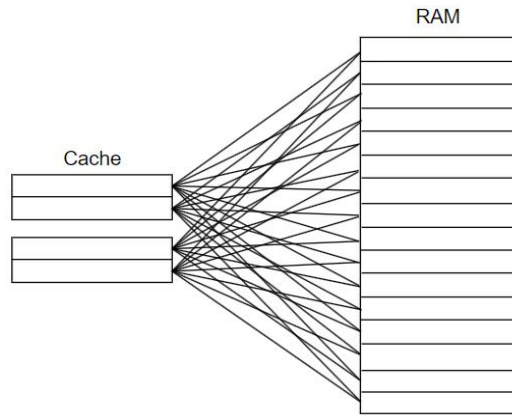


Рисунок 1.13 – Способ отображения строк основной памяти на строки множественно-ассоциативной кеш-памяти

Число множеств кеш-памяти в данном случае можно определить следующим образом:

$$N = \frac{V}{L * M} \quad (1.5)$$

. В качестве набора команд здесь используется набор команд CPU, при этом в процессе обращения к основной памяти осуществляется взаимодействие с многоуровневой кеш-памятью, в результате генерируются последовательности кеш-попаданий и кеш-промахов, оказывающие непосредственное влияние на итоговую производительность ВС.

Для векторизации вычислений необходимо наличие вычислительного устройства, способного выполнять векторные команды, где реализуется принцип SIMD, в рамках которого одна операция применяется к целому набору (вектору) данных (рисунок 1.14).

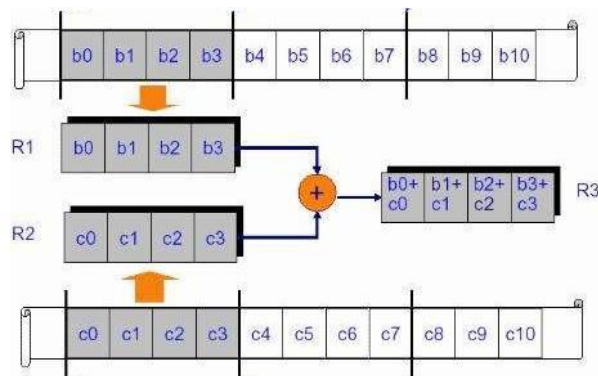


Рисунок 1.14 – Векторная обработка данных [177]

В качестве подобных устройств могут выступать векторные сопроцессоры, реализованные в виде отдельных устройств [149], однако наибольшее распространение получили блоки векторных расширений системы команд в МП общего назначения, характеризующиеся следующими параметрами:

- W - ширина SIMD-регистров; влияет на степень параллелизма при обработке векторов, оказывает влияние на доступные паттерны доступа к данным;

- влияние местоположения данных в памяти на эффективность команд; характеризует степень влияния выравнивания данных на длительность выполнения команд загрузки / сохранения:

- $A = ((address \% align\_num) == 0)$; условие выровненности адреса операнда; выражается в кратности адреса операнда $address$ определенному значению $align\_num$;

- $f: O \rightarrow T$; множество дополнительных тактов T , необходимых для выполнения операций над операндами с различным выравниванием O ; характеризует увеличение длительности выполнения команд при работе с невыровненными операндами, «штрафы» за использование команд обработки выровненных операндов по отношению к невыровненным и наоборот;

- состав команд; перечень обрабатываемых форматов данных, наличие команд горизонтальной обработки, команд перемешивания (shuffle) и др.

Для описания параметров ускорителей GPU, используемых для GPGPU в составе ГВС, используется программно-аппаратная архитектура CUDA [1] (рисунок 1.15).

На рисунке 1.15 используются следующие обозначения:

Host – центральный процессор CPU;

Device – графический ускоритель GPU;

Kernel - функция, которая будет выполняться на GPU;

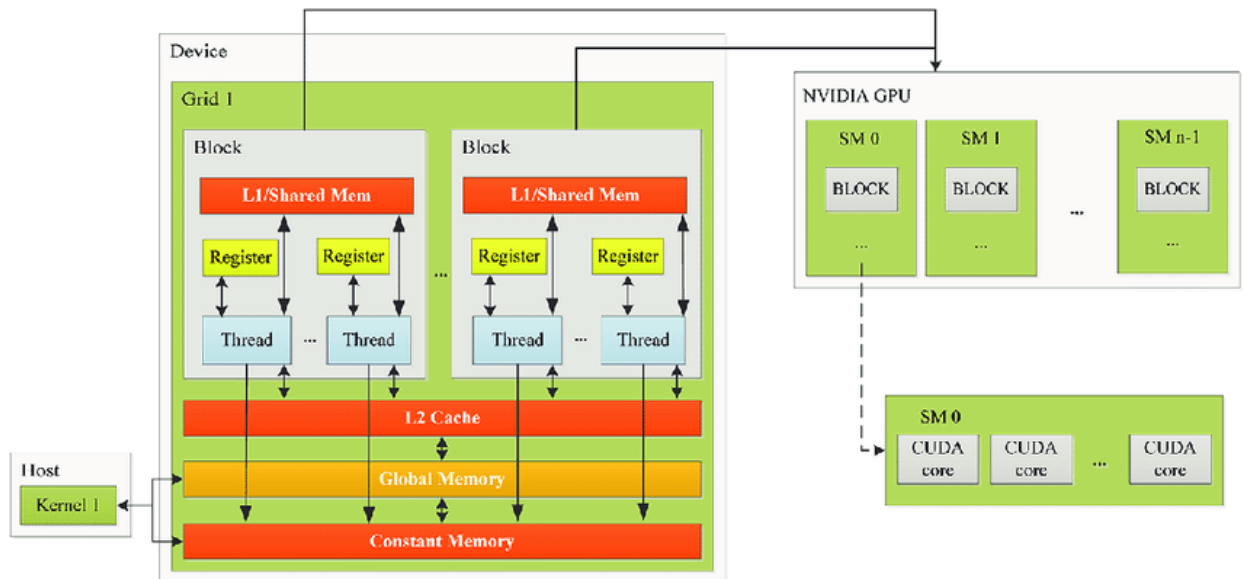


Рисунок 1.15 – Программно-аппаратная архитектура CUDA [108]

SM – потоковый мультипроцессор (Streaming Multi-processor);

CUDA Core – потоковый процессор (SP – Streaming Processors);

Grid - объединение блоков, которые выполняются на одном устройстве (device);

Block - объединение потоков (thread), которое выполняется целиком на одном SM (потоковый блок);

Thread – поток, единица исполнения потока команд;

Global, Shared, Constant Memory, L1, L2 cache – различные уровни памяти.

К параметрам GPU, влияющим на производительность в модели CUDA относятся:

- количество регистров;
- пропускная способность памяти;
- частота работы ядра, памяти;
- объем памяти различных уровней;
- размер варпа (warp) – группы потоков, выполняющихся одновременно;
- количество потоковых мультипроцессоров;
- версия вычислительных возможностей (compute capability);

- размерность конфигурации блоков в сетке и потоков в блоке.

Приведенные параметры являются аппаратными характеристиками устройства GPU, на которые разработчик повлиять не может. К параметрам, которые влияют на производительность проектируемой ПС, относятся:

- плотность вычислений – отношение числа арифметических операций к числу операций обращения к памяти;
- степень локальности данных – отношение числа потоковых обращений к памяти (пространственная локальность) к общему числу обращений к памяти;
- степень сбалансированности потоков;
- количество используемых потоков.

1.9 Цели и задачи исследования

Ранее исследованы научно-технические подходы к проектированию ЭПС для решения инженерных задач в ГВС, проанализировано влияние архитектуры вычислительных систем на эффективность программных систем. Показано, что подходы к разработке ЭПС, которые могут использоваться на начальном этапе проектирования и верификации алгоритмов, отсутствуют или являются неполными, показана применимость для этого инструментальных средств, используемых в процессе разработки. В связи с этим, **целью работы** является разработка научных основ поддержки жизненного цикла проектируемых программных систем для решения инженерных задач в гетерогенных вычислительных средах. Для достижения цели исследования в работе необходимо решить **задачи**:

1. Исследовать существующие научно-технические подходы, применяемые для проектирования и повышения эффективности ПС:
 - методы моделирования эффективных ПС для ВПВ в области решения инженерных задач;
 - методы улучшения стратегии кеширования данных;

- методы повышения эффективности векторизации вычислений;
- методы повышения эффективности использования GPU;
- инструментальные средства анализа и улучшения характеристик ПС для ВПВ.

2. Разработать методологию проектирования ПС для решения инженерных задач в ГВС, повышающую эффективность ПС на всех этапах ЖЦ, включающую основные проектные процедуры, модели и методы:

- компьютерные модели оптимального проектирования ПС для ГВС;
- методы повышения эффективности ПС для ГВС.

3. Верифицировать созданную методологию проектирования ЭПС на примере практически значимых вычислительно-сложных задач:

- реализовать методологический подход на примере операций BLAS 2 и 3 уровня и алгоритмов обработки числовых последовательностей;
- провести вычислительные эксперименты, проанализировать уменьшение времени решения задач, сокращение энергопотребления.

4. Разработать архитектурные и программные решения системы поддержки жизненного цикла проектируемых ПС.

1.10 Основные результаты и выводы по главе 1

Выполнен анализ существующих научно-технических подходов, применяемых для повышения эффективности ПС для решения инженерных задач. Приведены основные понятия и определения предметной области, определено понятие ПС как объекта. Рассмотрены стандарты в области проектирования ПС, определен объект проектирования, рассмотрен ЖЦ ПС. Проведен анализ влияния архитектуры ВС на эффективность ПС. Проанализировано влияние иерархической структуры памяти, распараллеливания вычислений, векторизации вычислений, организации распределенных вычислений, использования ускорителей на эффективность ПС. Определена целевая вычислительная архитектура. Проведен анализ

существующих методологий разработки ПС, используемых в области ВПВ для решения инженерных задач. Выявлена необходимость разработки новых подходов к проектированию подобных ПС.

Проанализированы методы анализа ПС. Показано, что, несмотря на свою эффективность, статические анализаторы не выполняют анализ эффективности ПС, например, эффективности использования кеш-памяти или других подсистем ВС. Анализ эффективности ПС осуществляется с использованием динамического анализа, однако он применяется к уже готовому коду, в который заложены возможные неэффективные проектные решения. Анализ эффективности ПС на начальных этапах проектирования ПС может значительно сократить время, затрачиваемое на разработку. Для этих целей необходимо внедрить в системы статического анализа подсистемы поддержки проектирования ПС, позволяющие выбирать проектные решения по повышению эффективности ПС.

Сформулирована цель диссертационного исследования, выполнена постановка задач поддержки ЖЦ проектируемых ПС.

2. СОВРЕМЕННЫЕ МЕТОДЫ ПОВЫШЕНИЯ ЭФФЕКТИВНОСТИ ПРОГРАММНЫХ СИСТЕМ

Глава содержит обзор современных методов и инструментальных средств, применяемых для повышения эффективности ПС в процессе их проектирования. Описываются методы моделирования эффективных ПС для решения инженерных задач, основной акцент делается на моделях, используемых для оценки производительности МП в целом и его различных подсистем, в частности подсистемы памяти. Рассматриваются методы улучшения стратегии кеширования данных, методы повышения эффективности векторизации вычислений, инструментальные средства проектирования и анализа программных систем.

В разделе 2.1 анализируются методы моделирования эффективных ПС для решения инженерных задач. Основное внимание уделяется моделям исполнения, которые исследуют поведение ВС с целью изучения, проверки и валидации. Рассматриваются модели ВС с иерархической структурой памяти, акцент делается на том, что ряд подходов основан на моделировании производительности целевой ВС. Выделяются две категории моделей: модели «черного ящика», опирающиеся на статистические методы и машинное обучение для описания и прогнозирования производительности ВС на основе наблюдаемых данных, и модели «белого ящика», использующие упрощенные математические модели для описания взаимодействия кода с оборудованием.

В разделе 2.2 рассматриваются методы улучшения стратегии кеширования данных. Показано, что данные методы основаны на повышении пространственной и временной локальности обрабатываемых данных, к наиболее эффективным методам относится разработка блочных алгоритмов решения задач. При этом одной из основных задач в реализации блочных алгоритмов решения задач, помимо разработки самого алгоритма, является задача подбора набора параметров алгоритма, которые дадут максимальную

производительность на ЦВА; здесь известны несколько подходов: множественная реализация и эмпирический поиск, генерация кода и автоматическая настройка и поиск параметров алгоритмов на основе аналитической модели.

В разделе 2.3 рассматриваются методы повышения эффективности векторизации вычислений. В данной области выделены несколько направлений исследований: способы векторизации, векторизация конкретных алгоритмов и задач, автоматическая векторизация вычислений. В области автоматической векторизации доминирующим подходом является Superword Level Parallelism (SLP), однако используются и другие подходы, например Lane Level Parallelism (LLP) и Padded SLP (PSLP). Для задачи автоматической векторизации выделяются задачи векторизации циклов с несмежными и невыровненными данными. Делается вывод о том, что задача эффективной векторизации обработки невыровненных данных на настоящий момент не имеет общего решения

В разделе 2.4 рассматриваются инструментальные средства проектирования и анализа программных систем, в т.ч. оптимизирующие компиляторы, инструменты статического и динамического анализа ПС, библиотеки функций, средства моделирования ВС, в т.ч. архитектурные симуляторы.

В 2.5 приводятся основные результаты и выводы по главе 2.

2.1 Методы моделирования эффективных ПС для решения инженерных задач

Разработка, основанная на моделях (Model-Driven Engineering), является одной из парадигм разработки ПС [59]. Моделирование представляет собой эффективный инструмент для анализа качественных характеристик ПС с целью изучения альтернативных вариантов проектирования или выявления потенциальных улучшений, что требует

проверки как функциональных, так и нефункциональных свойств путем изучения структурных и поведенческих аспектов системы.

Одним из классов таких моделей являются модели исполнения (executable models), которые исследуют поведение ВС с целью изучения, проверки и валидации (verifying and validating (V&V)) ее свойств. Часть этих моделей использует механизм трассировки для записи информации о выполнении программы (в трассы записывается последовательность изучаемых событий, например, кеш-промахов). Важной предпосылкой для большинства динамических инструментов V&V является возможность отслеживать выполнение исполняемых моделей. Примером подобного подхода являются модели, предложенные в [55, 56, 95], которые проверяют, удовлетворяют ли трассировки выполнения характеристикам ВС [10].

Моделирование на этапе проектирования ПС в основном используется для снижения сложности разработки за счет повышения уровня абстракции путем применения специальных языков моделирования, которые можно разделить на две группы:

- языки моделирования общего назначения (General-Purpose Modeling Languages, GPMLs), одним из наиболее известных представителей языков данной группы является Unified Modeling Language (UML);
- предметно – ориентированные языки моделирования Domain-Specific Modeling Languages (DSMLs)

Известно большое количество работ, посвященных поддержке реализации моделей, направленных на разработку исполняемых DSMLs (xDSMLs) [18, 25, 93], поддержку исполнения UML моделей [24], другим подходам [130]. Данные работы включают в себя как определение семантики выполнения языков моделирования, так и разработку динамических методов V&V, адаптированных к этим исполняемым языкам.

В области разработки моделей проектирования ПС и методов, основанных на этих моделях, известны исследования, связанные с разработкой моделей ВС с иерархической структурой памяти, при этом

некоторые подходы основаны на моделировании производительности целевой ВС. С некоторыми допущениями подобные модели можно разделить на две категории:

- моделирование “черного ящика”, которое опирается на статистические методы и машинное обучение для описания и прогнозирования производительности ВС на основе наблюдаемых данных, таких как показатели производительности оборудования [102];

- моделирование “белого ящика”, которое использует упрощенные математические модели для описания взаимодействия кода с оборудованием.

Представляют интерес модели, которые позволяют оценивать производительность ВС. При этом простые подходы, ориентированные на узкие места вычислений, основанные на соотношении пропускной способности памяти и пиковой производительности вычислений, используются уже давно [57]. Этот подход был популяризирован и усовершенствован Уильямсом и др. [132] и теперь известен как «Roofline Model».

Roofline Model представляет собой визуальную модель для оценки производительности ПС на конкретной ВС. Она основана на соотношении пропускной способности памяти и пиковой производительности вычислений и позволяет выявлять узкие места, связанные с производительностью ВС (рисунок 2.1). Основное предположение, используемое в данной модели, заключается в том, что передача данных по иерархии памяти перекрывается с выполнением кода в ядре (ядрах). Это подразумевает, что задержки доступа к данным, как предполагается, скрыты механизмами предварительной выборки.

Полученная кривая включает в себя два потолка производительности: пропускная способность памяти и пиковая производительность МП. На графике используются следующие обозначения:

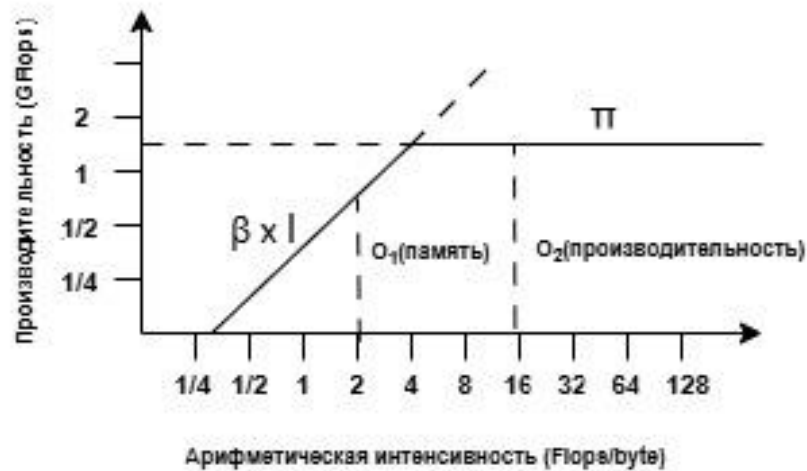


Рисунок 2.1 – Roofline Model

π – пиковая производительность МП (Floating-point Operations per Second, FLOPS);

β – пиковая пропускная способность памяти;

I – арифметическая интенсивность (FLOPS/byte).

Арифметическая интенсивность определяется как отношение общего количества операций, выполненных МП, к объему переданных из памяти данных. Другими словами, данный параметр показывает, сколько операций выполняется в МП из расчета на один байт трафика из памяти. При этом достижимая производительность определяется в соответствии с (2.1), график изображается с обеими осями в логарифмическом масштабе.

$$P = \min \left\{ \frac{\pi}{\beta * I} \right\} \quad (2.1)$$

Точка перегиба графика характеризует минимальную арифметическую интенсивность, необходимую для достижения максимальной производительности и может использоваться разработчиком для оценки необходимых усилий по оптимизации ПС. Анализ графика, приведенного на рисунке 2.1, показывает, что производительность O_1 ограничена пропускной способностью памяти, а O_2 – пиковой производительностью. При этом во втором случае используются вычисления, которые имеют гораздо более высокую арифметическую интенсивность.

Часто к модели добавляют дополнительные потолки производительности, как это показано на рисунке 2.2.

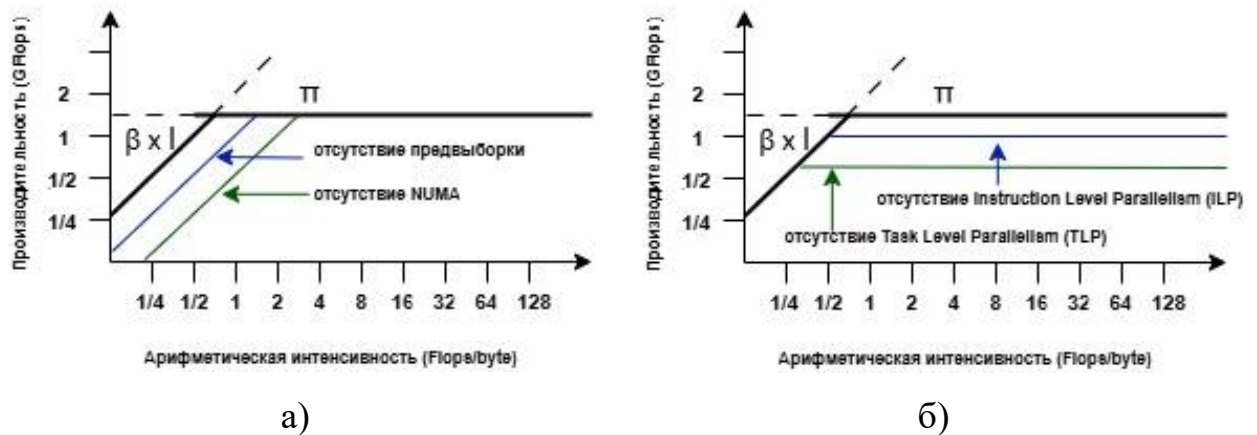


Рисунок 2.2 – Roofline Model с дополнительными потолками пропускной способности памяти а) и пиковой производительности б)

На рисунке 2.2а) добавлены дополнительные потолки пропускной способности памяти, которые представляют собой отсутствие предвыборки данных из памяти и отсутствие архитектуры Non-Uniform Memory Access (NUMA). Другими словами, в первом случае учитываются особенности иерархической структуры памяти, во втором случае – проблемы с масштабируемостью ВС с общей памятью.

На рисунке 2.2б) добавлены дополнительные потолки пиковой производительности ВС, связанные с отсутствием параллелизма на уровне инструкций и задач.

Несмотря на простоту и наглядность, данная модель имеет ряд недостатков. Во-первых, она не может быть использована в процессе проектирования ПС для повышения ее эффективности, т.к. не дает ответа на вопрос каким образом стоит повышать арифметическую интенсивность вычислений, предсказывая только ее минимально необходимый уровень для достижения пиковой производительности МП. Во-вторых, точная оценка параметра «арифметическая интенсивность» невозможна без учета особенностей архитектуры ВС, а «пропускная способность памяти» без учета структуры и параметров иерархической структуры памяти ВС.

Моделей, использующих концепцию «белого ящика» и учитывающих особенности аппаратуры ВС достаточно много. Понимание свойств и узких мест производительности ВС позволяет с помощью моделирования получить четкое представление о перспективных возможностях оптимизации.

Примером такой модели является Execution-Cache-Memory Model (ECM)|, которая обеспечивает точное предсказание производительности одного ядра МП и масштабирования вычислений. Модель можно рассматривать как расширение и доработку модели Roofline. Впервые она была представлена в [119], позже усовершенствована в [52]. Модель предлагает итеративный подход для аналитического прогнозирования производительности алгоритмов, эффективность которых ограничена полосой пропускания на всех уровнях иерархии памяти.

Основным структурным блоком исследуемых алгоритмов в данной модели является его вычислительное ядро во внутреннем цикле кода программы. Производительность данного ядра ограничена кеш-памятью L1, определяющей максимально возможное количество загрузок и сохранений данных за одну итерацию цикла, а также затратами ядра МП на выполнение программных инструкций. Таким образом, с учетом объемов данных, передаваемых по иерархии памяти и возможностей полосы пропускания подсистемы памяти, анализируется время, необходимое для каждой такой передачи и время, необходимое для выполнения инструкций с данными, поступающими из кеша L1. Результатом является время, необходимое для выполнения ядра цикла, при условии отсутствия задержки доступа или перекрытия входных данных. Дополнительно учитывается время, необходимое для обмена L1 с верхними уровнями иерархии памяти, которое называется «задержками передачи данных»

Очевидно, что для правильного применения этой модели требуется глубокое знание архитектуры кеш-памяти, МП и путей передачи данных в процессе выполнения программы. Эту информацию можно получить у

производителей процессоров, но иногда уровень детализации недостаточен для фиксации всех параметров, и соответствующую информацию необходимо получать из измерений, проводимых в рамках вычислительных экспериментов [119].

Работа [52] посвящена реализации ЕСМ на многоядерных МП, в качестве целевой платформы был выбран Intel Sandy Bridge. В данной работе учитывается длина строки и инклюзивность кеш-памяти, предвыборку данных. Кроме того, учитывается количество портов в L1. Рассматриваемая в работе ЦВА является в этом смысле «однопортовой», т.е. она может либо перезагружать/удалять строки кеша из/в L2, либо взаимодействовать с регистрами, но не работать в обоих направлениях одновременно. Авторы акцентируют внимание на том, что в сложном программном цикле часто бывает трудно найти критический путь выполнения, который определяет количество машинных циклов. При этом говорится о возможности, а часто о необходимости использования таких инструментов, как Intel Architecture Code Analyzer (IACA) [118].

Во многих современных многоядерных МП одно ядро не способно полностью загрузить интерфейс памяти, хотя простое сравнение пиковой производительности и пропускной способности памяти в рамках Roofline свидетельствует об обратном. Модель ЕСМ объясняет этот эффект, разделяя время исполнения на непересекающиеся вклады вычислений на ядре и передачи данных и тем самым развивая и уточняя модель Roofline:

В работе [116] модель ЕСМ применяется для количественной оценки «узких мест» производительности для трафаретных вычислений. Для определения времени выполнения внутри вычислительного ядра требуется простая модель для определения пропускной способности микроархитектуры. В МП Intel для этой цели можно использовать модель планировщика портов. В работе была использована модель портов для микроархитектуры Intel Sandy Bridge. Предполагалось, что все команды в цикле независимо распределяются по портам и выполняются вне

программного порядка (out-of-order) в исполнительных модулях. Время выполнения устанавливается тем модулем, который выполняет больше всего циклов для выполнения инструкций. Дополнительным ограничением, которое оказывает влияние на результат, является общая максимальная пропускная способность команд, составляющая четыре команды в рассматриваемой микроархитектуре. В качестве примера рассматривалась реализация программного цикла DAXPY

```
for( i=0; i<N; ++i)
    a[ i] = a[ i] + s * b[ i];
```

Одна итерация цикла состоит из двух команд загрузки, одной команды сохранения, одной команды сложения и одной команды умножения в случае, если не используется векторизация вычислений. Кроме того, обычно в теле есть одно целочисленное сложение, одно целочисленное сравнение и одна условная ветвь для реализации “механики цикла”. Если цикл развернут, относительная доля этих инструкций будет меньше, и ими обычно можно пренебречь. При векторизации с использованием технологии AVX один рабочий блок (восемь итераций) включает в себя четыре команды загрузки, две команды сохранения, две команды сложения и две команды умножения, для выполнения которых ядру Intel Sandy Bridge требуется четыре цикла. На рисунке 2.3 показана одноядерная модель ECM для итерации DAXPY на микроархитектуре Intel Sandy Bridge.

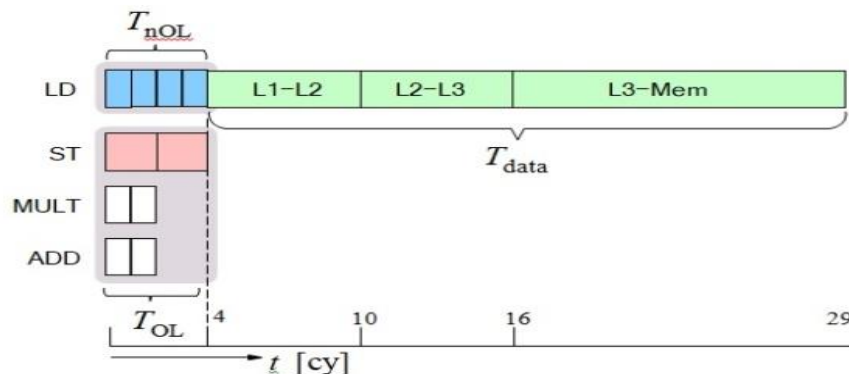


Рисунок 2.3 – Модель ECM цикла DAXPY на МП Intel Sandy Bridge

Предсказанное количество циклов на обработку одной кеш-линии составляет 29, если данные находятся в памяти, 16, если они находятся в L3 cache, 10 для L2 cache, и 4 для случая, когда все данные находятся в L1 (используется технология векторизации AVX).

Данная модель обеспечивает прогнозирование количества циклов МП, необходимых для выполнения анализируемого кода (чаще всего цикла) и время передачи данных между уровнями кеш-памяти, а также между кеш-памятью и основной памятью. Она дает четкое представление о соответствующем вкладе аппаратной модели в производительность данного кода цикла и, следовательно, позволяет четко определить возможности оптимизации. Модель ЕСМ обеспечивает прогнозирование количества циклов процессора, необходимых для выполнения определенного числа итераций данного цикла на одном ядре:

- время выполнения инструкций на ядре МП, предполагая, что промахов в кеш-памяти нет;
- время передачи данных между их начальным местоположением и кешем L1 (рассчитывается максимальная, без задержек, скорость передачи данных L1 – L2, L2 – L3, L3 - память);
- осуществляется точная количественная оценка времени выполнения, основанная на прогнозировании количества машинных циклов МП, необходимых для выполнения определенного числа итераций заданного программного цикла на одном ядре МП

Данный подход может достаточно достоверно прогнозировать производительность МП в процессе выполнения анализируемой программы, однако, данный подход обладает существенными недостатками:

- привязка к конкретному МП (для другого устройства необходимо провести новое моделирование),
- сложность проведения аналитического моделирования.

Кроме того, не представляется оптимальным выполнение сложного аналитического расчета для оценки производительности (например, для МП

Intel используется модель планировщика портов), т.к. на практике подобная точность нужна не всегда или не нужна вовсе, однако, она добавляет сложность в процесс моделирования.

Другим типом моделей ВС с иерархической структурой памяти является модель кеш-независимых алгоритмов (Cache-Oblivious Algorithm) [40, 136]. В данном случае модель системы МП – кеш-память позволяет асимптотически оптимально использовать кеш-память, игнорируя ее отдельные параметры, такие как число уровней кеша, их объем, длину кеш-линии и т.д. В отличие от других подходов к проектированию эффективных алгоритмов, в данном случае не учитывается кеширование: для достижения эффективности, не требуется настраивать переменные, зависящие от аппаратных параметров, такие как размер кеша и длина строки кеша. Тем не менее, полученные алгоритмы выполняют необходимый объем работы и эффективно перемещают данные между несколькими уровнями кеш-памяти. При этом очевидно, что алгоритмы, не обращающие внимания на ресурсы, которые, тем не менее, эффективно используют ресурсы, обладают преимуществами простоты и переносимости по сравнению с алгоритмами, учитывающими ресурсы, использование которых должно быть запрограммировано явно.

Авторы в [40] предлагают идеальную модель кеш-памяти, структура которой приведена на рисунке 2.4.

Структура памяти в данном случае состоит из двух уровней: кеш-памяти и основной памяти. Данные перемещаются кеш-линиями, которые содержат несколько слов фиксированного размера. Используется модель идеального кеша, который является одноуровневым, имеет большую высоту и полностью ассоциативен, использует оптимальную стратегию замены строк кеша, таким образом, идеально используется временная локальность.

Исторически сложилось так, что хорошая производительность была получена с использованием алгоритмов, учитывающих параметры кеш-

памяти, но авторы предложили несколько эффективных алгоритмов, не учитывающих кеш.

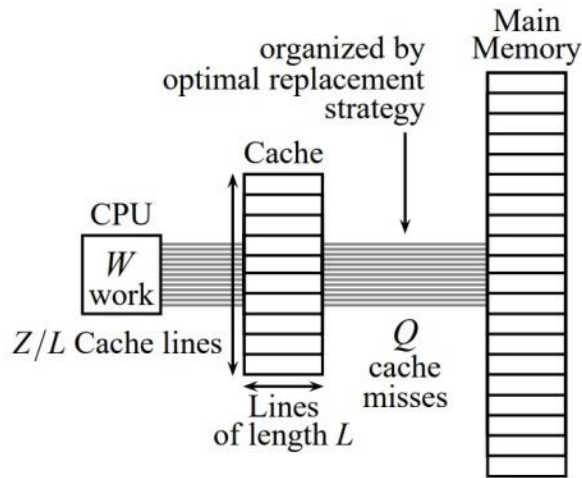


Рисунок 2.4 – Идеальная модель кеш-памяти в модели кеш-независимых алгоритмов [40]

Модель идеального кеша делает, возможно, сомнительные предположения о том, что в иерархии памяти существует только два уровня (кеш-память и основная память), что управление памятью осуществляется с помощью оптимальной стратегии замены строк кеша и что кеш является полностью ассоциативным. Однако авторам удалось добиться достаточно высокой эффективности для ряда алгоритмов. Подобный подход позволяет значительно упростить процесс оптимизации ПС с одной стороны, однако, с другой стороны, не позволяет глубоко оптимизировать код под конкретную ВС.

Разработанные в рамках данной модели алгоритмы основаны на парадигме "разделяй и властвуй" - каждый шаг разделения создает подзадачи меньшего размера, и когда рабочий набор подзадач помещается на некотором уровне иерархии памяти, вычисления в этой подзадаче могут выполняться без потери производительности на этом уровне. Таким образом, алгоритмы "разделяй и властвуй" автоматически адаптируются ко всем уровням иерархии памяти; фактически, для таких задач, как умножение матриц, транспонирование матриц и БПФ, эти рекурсивные алгоритмы являются оптимальными с точностью, сравнимой с некоторыми

теоретическими моделями иерархии памяти. Однако, в работе [136] показано, что ПС для ВПВ должны, как минимум, учитывать требования МП, то есть опираться на микроядра, тщательно оптимизированные под используемую микроархитектуру. Кроме того, полученные результаты демонстрируют то, что итеративные микроядра, разработанные для задачи умножения матриц, работают лучше, чем рекурсивные микроядра, которые были созданы на базе модели кеш-независимых алгоритмов. Это связано с тем, что итеративные микроядра имеют больше степеней свободы, что позволяет проводить независимую оптимизацию для различных ресурсов МП.

Такой подход позволяет разрабатывать ПС, которые работают эффективно и без модификаций в различных ВС, независимо от характеристик кеш-памяти. Однако, модели с «точной настройкой» параметров алгоритмов дают более эффективные решения. К таким моделям относится Target Architecture Platform (TAP) [87] (модель целевой архитектурной платформы, ЦАП).

В работе [87] показано, как фреймворк BLAS-like Library Instantiation Software (BLIS) позволяет аналитически определять параметры настройки для высокопроизводительных экземпляров умножения матриц. Идея параметризованных алгоритмов не нова. В научной литературе параметризация алгоритмов описывается как процесс, связанный с:

- компиляцией приложения с различными наборами параметров; если параметров много, много будет и вариантов полученных исполняемых модулей;

- запуском полученных модулей с замеров времени исполнения;

- выбором лучшего варианта.

В любом случае данный процесс должен осуществляться высококвалифицированным специалистом, знающим особенности архитектуры ГВС, понимающим степень влияния различных архитектурных особенностей и параметров алгоритма на итоговую эффективность получаемой ПС.

Данный подход значительно сокращает затраты на разработку, необходимые для реализации операций BLAS 3-го уровня, а также улучшает понимание того, как иерархически многоуровневая память взаимодействует с высокопроизводительным программным обеспечением. Предложенный аналитический подход по утверждению авторов также позволяет перейти от дорогостоящих инженерных решений (эмпирическая настройка и автонастройка) к научному пониманию проблемы (аналитическое решение). Работа [87] в определенной степени основана на более ранних работах [129, 136], однако использует более современную модель МП. Модель допускает реализацию таких аппаратных функций, как векторный регистровый файл и иерархия памяти с несколькими уровнями кешей данных. Идеи, заложенные в данной модели, позволяют после определенной доработки применять ее практически на любых микроархитектурах МП.

Модель ЦАП описывается следующим образом.

1. Архитектура загрузки/сохранения и векторные регистры. Данные должны быть загружены в регистры МП, прежде чем с ними можно будет выполнять вычисления. Существует в общей сложности несколько векторных регистров N_{REG} , каждый из которых может содержать элементы N_{VEC} (числа с плавающей запятой) размера S_{DATA} (в МП без векторных регистров $N_{\text{VEC}} = 1$.) Предполагается, что команды для работы с памятью могут выполняться одновременно с командами арифметическими обработки чисел с плавающей запятой.

2. Векторные инструкции. Здесь используется несколько параметров. Параметр N_{VFMA} – количество инструкций VFMA (Vector Fused Multiply-Add), выполняемых МП за один такт. Параметр L_{VFMA} , представляющий собой минимальное количество тактов, которые необходимы перед началом выполнения новой инструкции VFMA, представляет собой задержку VFMA-инструкции.

3. Кеш-память.. Кешы данных всех уровней являются множественно-ассоциативными с политикой вытеснения LRU, каждый уровень кеш-памяти L_i характеризуется следующими параметрами:

- C_{Li} : размер кеш-строки,
- W_{Li} : степень ассоциативности,
- S_{Li} : размер;
- N_{Li} : количество множеств,

При этом предполагается, что пропускная способность между кешами и регистрами не является узким местом в вычислениях. Полностью ассоциативную кеш-память можно смоделировать, установив параметр N_{Li} в значение 1. В реальных случаях значения параметров модели ЦАП определяются техническими характеристиками целевой аппаратной платформы. Если у МП отсутствуют VFMA-инструкции, L_{VFMA} может быть вычислена как сумма задержек векторного сложения и умножения. Значение N_{VFMA} может быть вычислено как минимум из пропускных способностей векторных инструкций сложения и умножения, если процессор способен выполнять обе эти инструкции одновременно [87]. В противном случае, значение N_{VFMA} равно половине минимальной из этих пропускных способностей.

Для вывода формул, позволяющих найти значения параметров исследуемого алгоритма, предлагается смоделировать выполнение алгоритма на модели ЦАП Реализации матричного умножения, полученные с использованием ЦАП вполне конкурентоспособны с реализациями, настроенными вручную. В работах [73, 136] показано, что аналитические модели позволяют создавать код, реализованный на языке программирования C, который конкурирует с реализациями, сгенерированными с помощью эмпирического поиска. Однако слабым местом данных работ является то, что описанный в них подход не создает код, конкурентоспособный с реализациями, использующими внутренние ядра с ручным кодированием. Модель ЦАП позволяет обеспечить производительность,

конкурентоспособную с реализациями, настроенными вручную, которые обеспечивают лучшую в своем классе производительность с помощью аналитически полученных значений параметров. В работе рассматривался вариант однопоточной реализации матричного умножения с двойной точностью. Вопрос определения оптимальных значений параметров для многопоточной реализации частично рассмотрен в [113].

Таким образом, модель ЦАП используется для определения параметров настройки для высокопроизводительных программных реализаций умножения матриц. Хотя ее применение может быть расширено на любые другие алгоритмы, авторы не предложили иных реализаций. Модель помогает найти значения параметров алгоритма путем моделирования выполнения алгоритма на модели ЦАП. Моделирование является аналитическим, не предлагается средств автоматизации процесса. Однако применение данной модели позволило авторам получить эффективные программные реализации задачи матричного умножения.

Несмотря на то, что модели, основанные на модели ЦАП, не являются новыми, они лежат в основе технологии кеш-оптимизации ПС и постоянно совершенствуются. При этом, несмотря на свою относительную эффективность, данные модели имеют существенные недостатки. Абсолютное большинство предложенных моделей направлено на решение задачи умножения плотных матриц. Используемые аналитические подходы, описанные в ряде работ, не предполагают автоматизации, для другого преобразования необходимо проводить похожее исследование. Отмечается также сложность проведения подобного анализа, требующая наличия высокой квалификации и ряда компетенций у исследователя-разработчика. Для большинства подходов необходим непосредственный доступ к ВС, на которой планируется использование исследуемой ПС. Кроме того, авторы методик не предложили информационных систем, автоматизирующих процесс моделирования.

Существует большое количество моделей, связанных с параллельными вычислениями, подробный обзор которых дан в [172]. В работе подчеркивается важность моделей параллельных вычислений, которая вытекает из того, что они до реализации параллельного алгоритма в виде программы позволяют понять, насколько эффективно определенный алгоритм может использовать конкретную многопроцессорную платформу, и при необходимости изменить алгоритм, либо рассмотреть вариант замены целевой аппаратной платформы. Создание простой, адекватной и универсальной модели параллельных вычислений является практически неразрешимой задачей в связи большим разнообразием многопроцессорных архитектур, появляющихся и развивающихся в настоящее время в погоне за повышением производительности. Для преодоления возникших трудностей был применен подход, в соответствии с которым многопроцессорные архитектуры были разделены на три класса: архитектуры с общей памятью, архитектуры с распределенной памятью и иерархические многопроцессорные архитектуры [138]. Подобный подход позволил построить простые и универсальные модели с высоким уровнем абстракции.

В качестве промежуточного итога можно выделить следующие группы моделей в области проектирования эффективных ПС:

- простые модели оценки производительности, примером которых является Roofline Model; к недостаткам данных моделей можно отнести невозможность непосредственного использования в процессе проектирования ПС и невозможность точной оценки параметров модели без детального анализа особенностей архитектуры ВС;

- модели, обеспечивающие точное предсказание производительности МП, примером является ЕСМ; к недостаткам можно отнести привязку к конкретному МП и сложность проведения аналитического моделирования;

- модели, делающие допущения, которые значительно упрощают анализ, примером является модель кеш-независимых алгоритмов; к

недостаткам можно отнести гораздо менее эффективные решения по сравнению с моделями с «точной настройкой»;

- моделями с «точной настройкой», примером является модель ЦАП; к недостаткам можно отнести сложность проведения аналитического моделирования, необходимость проведения нового исследования для каждого нового алгоритма; кроме того все примеры использования данной модели касаются одной задачи – матричного умножения.

К недостаткам всех перечисленных моделей можно отнести отсутствие средств автоматизации, необходимость поведения «ручного» анализа, часто достаточно сложного.

Также общим недостатком приведенных моделей является их «однопоточная» реализация, анализ проводится для одного ядра. Некоторые усилия в сторону анализа многопоточных реализаций сделаны в ЕСМ и ЦАП. При этом стоит отметить большое количество моделей, связанных с параллельными вычислениями, однако данные модели в свою очередь не позволяют оптимизировать реализации на отдельном ядре МП, что крайне важно для ВПВ.

2.2 Методы улучшения стратегии кеширования данных

Существуют общие рекомендации, выработанные сообществом разработчиков в соответствии с архитектурными особенностями ВС с иерархической структурой памяти, позволяющие улучшить стратегию кеширования данных в процессе выполнения программы (рисунок 2.5).

Под стратегией кеширования понимается организация доступа к данным, влияющая на эффективность использования кеш-памяти. Эффективность использования кеш-памяти в большой степени зависит от пространственной и временной локальности данных (глава 1), поэтому наибольший эффект достигается при регулярном доступе к данным и при повторном использовании данных, ранее попавших в кеш-память.



Рисунок 2.5 – Общие рекомендации по улучшению стратегии кеширования данных

Ранее была показана роль предварительной выборки данных в современных МП, поэтому улучшение стратегии кеширования данных может иметь простое решение в виде регулярного паттерна доступа к данным, который эффективно обрабатывается блоками предварительной выборки данных в МП.

Оптимизация доступа к данным представляет собой преобразования кода, которые изменяют порядок выполнения итераций в цикле. Целью этих преобразований является, главным образом, улучшение временной локальности. Кроме того, они также могут обеспечить параллелизм и сделать итерации цикла векторизуемыми.

В случае, если невозможно построить эффективный регулярный доступ, необходимо выполнить оптимизацию расположения данных, которая повышает эффективность иерархической структуры памяти и блока предварительной выборки. Классическим примером здесь является выбор между вариантами хранения Array of Structures и Structure of Arrays, оптимизация хранения структур данных. Подробно оптимизация расположения данных рассмотрена в [77].

Подходы, представленные на рисунке 2.5, расположены от простого к сложному, т.е. проще всего использовать пространственную и временную локальность и оптимизировать доступ к данным, сложнее всего разрабатывать блочные алгоритмы решения задач. Идея разработки блочных алгоритмов заключается в разделении вычислений на части (блоки), которые целиком помещаются в кеш-память определенного уровня (чаще всего L1) для минимизации количества кеш-промахов за счет повышения пространственной и временной локальности обрабатываемых данных. Различают рекурсивные (рассмотрены выше, модель кеш-независимых алгоритмов) и итеративные блочные алгоритмы. В итеративных алгоритмах блоки данных обрабатываются друг за другом, возможна многократная обработка тех или иных блоков. Классическим примером блочного итеративного алгоритма является алгоритм блочного умножения матриц.

Оптимальные размеры блоков зависят от характеристик ЦВА и являются параметрами подобных алгоритмов. Одной из основных задач здесь, помимо разработки самого алгоритма, является задача подбора набора параметров алгоритма, которые дадут максимальную производительность на целевой вычислительной архитектуре. Здесь стоит отметить работы Ф.Густавсона Дж.Маккомба [2] (множественная реализация алгоритмов и эмпирический поиск параметров), Р.Уэйли, Дж.Донгарра, Д.Билмса, Дж.Деммеля [13. 128] (автонастройка с использованием генерации кода), Д.Спампинато, Г.Белтера, К.Ванга [114] (автонастройка с использованием аналитических техник), К.Йотова, Т.Лоу [87] (аналитический поиск параметров), и др. (рисунок 2.6).

Самый простой «ручной» подход к настройке ПС для ЦВА заключается в написании различных версий кода ПС, оценке производительности этих версий на целевой ВС и выборе той версии, которая обеспечивает наилучшую производительность. Эти разные версии обычно реализуют один и тот же алгоритм, но отличаются значениями, которые они используют для

таких параметров, как размеры блоков и коэффициенты развертывания цикла.



Рисунок 2.6 – Методы параметризации алгоритмов

Архитектурные представления и знание предметной области проектировщиком ПС используются для ограничения количества оцениваемых версий. Реализуется трехэтапный процесс:

- 1) создание версии программы, соответствующей каждой комбинации рассматриваемых параметров;
- 2) тестирование каждой версии на целевой ВС и измерение производительности;
- 3) выбор версии, которая работает наилучшим образом.

Этот подход активно применялся Ф. Густавсоном и его коллегами из IBM, которые в течение многих лет использовали его для создания высокооптимизированных библиотек ESSL и PESSL для компьютеров IBM.

Далее в ряде проектов, таких как FFTW [41, 42], PhiPAC [13], ATLAS [129] и SPIRAL [101], была автоматизирована генерация различных версий программ, производительность которых необходимо оценить. Большая часть подобных исследований проводилась при разработке библиотек программ, реализующих операции линейной алгебры. Область плотной линейной

алгебры была одной из первых, где было понято и принято сообществом, что стандартизация программного интерфейса фундаментальных примитивов обеспечивает переносимость и высокую производительность. Таким интерфейсом стал интерфейс BLAS, который развивался в течение нескольких десятилетий [28, 29, 85]. Ожидалось и до сих пор ожидается, что сообществу будет предоставлена высокопроизводительная реализация BLAS каждый раз, когда новая архитектура выходит на рынок. Для этой цели многие производители оборудования разрабатывают и распространяют собственные высокопроизводительные библиотеки (например, библиотеки Intel MKL, AMD ACML и IBM ESSL). При этом распространение получили и альтернативные решения с открытым исходным кодом (например, GotoBLAS [44], OpenBLAS [98], ATLAS [128]). Тем не менее, все эти решения требуют значительных затрат для каждой поддерживаемой целевой архитектуры.

Чтобы снизить затраты на разработку высокопроизводительных реализаций, ATLAS и его предшественник PHiPAC внедрили автонастройку. Идеи, заложенные в этих библиотеках, заключаются в том, что оптимизация сложна и отнимает много времени у специалиста-человека и что автогенерация в сочетании с автонастройкой является решением проблемы.

Некоторые исследователи добавили аналитические техники в автонастройку. Так, в работе [114] описано создание компилятора LGen, генерирующего код для реализации базовых операций линейной алгебры небольших размерностей в противовес таким библиотекам, как ATLAS или Intel MKL, ориентированные в основном на решение задач, где размерности матриц исчисляются сотнями и более элементов. В данном компиляторе совмещается аналитический подход, реализующий разбиение вычислений, оптимизацию циклов и векторизацию вычислений на высоком уровне абстракции и эмпирический поиск для выбора среди созданных реализаций наиболее эффективной. В результате авторами были получены эффективные конкурентоспособные реализации. Похожие решения описаны в работах [12, 126].

Работа [137] продемонстрировала, что подход ATLAS к оптимизации произведения матриц общего вида может быть, по большому счету, аналитически смоделирован. На рисунке 2.7 показан оригинальный вариант ATLAS и модифицированная версия, которая использует аналитическое моделирование вместо эмпирического поиска.

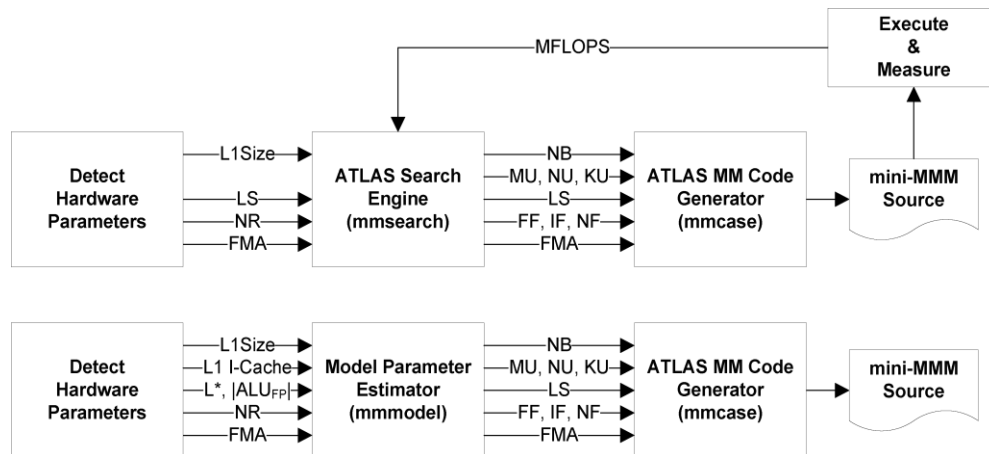


Рисунок 2.7 – Архитектура ATLAS и ATLAS, основанного на моделировании [137]

Как и любая система, использующая эмпирический поиск, ATLAS имеет:

- 1) модуль, управляющий поиском, который используется для определения оптимальных значений параметров оптимизации кода (mmsearch);
- 2) модуль, генерирующий код с учетом этих значений (mmcase).

ATLAS использует целый ряд параметров, например, размер блока, который будет использоваться при оптимизации кода для кеша данных L1. Как правило, существует неограниченное количество возможных значений для такого параметра, поэтому необходимо ограничить размер области поиска. Когда ATLAS установлен в ВС, он сначала запускает набор микробенчмарков для определения параметров оборудования, таких как емкость кеша данных L1 и количество регистров. Эти параметры оборудования используются для ограничения пространства поиска. Модуль mmsearch формирует значения параметров в пределах этого ограниченного

пространства поиска, после чего вызывает модуль `mmcase` для генерации соответствующего кода (обозначенного на рисунке символом `mini-MMM`), запускает этот код в реальной ВС и записывает время его выполнения. В конце поиска значения параметров, которые дали наилучшие результаты, используются для генерации кода библиотеки.

Авторы метода [137] модифицировали ATLAS, заменив модуль поиска модулем (`mmmodel`), который использует аналитические модели для оценки оптимальных значений параметров оптимизации. Для описания модели использовались следующие параметры, характеризующие целевую вычислительную архитектуру:

- C_1, B_1 – объем и размер строки для кеша данных L1;
- C_I – объем L1 кеша инструкций;
- L_X – время выполнения команды умножения с плавающей запятой;
- $|ALU_{FP}|$ - количество вычислительных блоков с плавающей запятой;
- N_R – количество регистров формата с плавающей запятой;
- FMA – наличие команд FMA (`fused multiply-add`)

Эмпирические оптимизаторы (ATLAS) используют значения параметров ВС только для ограничения области поиска, поэтому достаточно приблизительных значений для этих параметров. В отличие от этого, аналитические модели требуют точных значений для этих параметров.

Проведенные вычислительные эксперименты продемонстрировали, что можно использовать аналитические модели для определения близких к оптимальным значений параметров оптимизации, необходимых в системе ATLAS для получения эффективных реализаций BLAS. Использование моделей для определения значений параметров оптимизации оказалось намного быстрее использования эмпирического поиска. Однако это не означает, что поиск не играет никакой роли в генерации эффективных ПС. Например, такие системы, как FFTW и SPIRAL, используют поиск не для выбора оптимальных значений параметров преобразования, а для выбора оптимального алгоритма из целого набора алгоритмов.

Проблема данного подхода заключается в том, что ATLAS использует внутреннее ядро, оптимизированное экспертом-человеком, которое по-прежнему требует существенного ручного кодирования. Именно наличие этого внутреннего ядра "черного ящика" делает аналитическое моделирование проблематичным для ATLAS, поскольку некоторые параметры настройки скрыты внутри. Таким образом, слабым местом данного подхода является то, что он не создает код, конкурентоспособный с реализациями, использующими внутренние ядра с ручным кодированием.

Подход, описанный в [87] устраняет данный недостаток. В частности предлагаются следующие улучшения:

- более репрезентативная модель МП, включающая в себя векторные регистры и иерархию памяти с несколькими уровнями кеш-памяти, в отличие от модели mmmmodel (рисунок 2.7), где рассматривался один уровень кеш-памяти и игнорировались векторные инструкции;

- более полная аналитическая модель в том смысле, что она включает значения параметров, которые также характеризуют микроядро, которые ранее игнорировались; данный аспект является важным, так как лучшие реализации, предоставляемые ATLAS, часто включают циклы вокруг ядра с ручным кодированием (черный ящик);

- более практичная модель, которая предназначена для предоставления фактических значений, которые могут быть использованы разработчиком для настройки платформы BLIS для конкретной архитектуры, а не для использования в качестве эвристики для руководства эмпирической поисковой системой.

В результате был разработан подход, позволяющий аналитически получать значения параметров, которые обеспечивают производительность, конкурентоспособную с реализациями, настроенными вручную, которые обеспечивают лучшую в своем классе производительность.

Описанные подходы предполагают улучшение стратегии кеширования данных, которое достигается за счет минимизации числа кеш-промахов в

процессе вычислений. Можно сказать, что данные улучшения происходят в результате уменьшения количества данных, перемещаемых между уровнями иерархии памяти, взаимодействия этих уровней. В работах [9, 26] авторы подчеркивают, что современный технологический уровень производства МП сильно сократил время, необходимое для выполнения арифметических операций. Поэтому количество операций обработки данных больше не является узким местом во многих алгоритмах, хотя данный показатель считался традиционным критерием эффективности численного алгоритма, непосредственно влияющим на время решения задачи и энергопотребление ВС. Скорее узким местом является перемещение данных между уровнями иерархии памяти и вычислительными узлами в сети. В связи с этим авторы формулируют вопросы, связанные с разработкой прямых и итерационных алгоритмов численной линейной алгебры:

- существуют ли нижние границы объема перемещения данных, которые должны выполнять эти алгоритмы;
- достигают ли существующие широко используемые алгоритмы этих нижних пределов;
- если нет, то существуют ли другие алгоритмы, которые достигают этих границ.

Авторы показали, что значения таких нижних границ можно получить аналитически, при этом существующие широко используемые алгоритмы плотной линейной алгебры не достигают этих границ. Авторы проанализировали большое количество алгоритмов и выделили те, которые достигают границ объема перемещения данных, полученных аналитически. При этом только некоторые из них включены в стандартные библиотеки, такие как LAPACK.

Анализ проводился для алгоритмов плотной линейной алгебры, однако авторы утверждают, что подобные алгоритмы, оптимальные с точки зрения перемещения данных (Communication-Avoiding Algorithms), можно получить и для других классов задач. Концепцию нижних границ объемов перемещения

данных границы и оптимальных алгоритмов можно распространить на гораздо более общий класс задач, решаемых с использованием циклов, в том числе вложенных. Вложенные цикла умножения матриц и другие алгоритмы линейной алгебры являются каноническими примерами, но, конечно, многие другие алгоритмы могут быть проанализированы таким образом.

В качестве промежуточного итога можно выделить сделать следующие выводы:

- методы улучшения стратегии кэширования данных основаны на повышении пространственной и временной локальности обрабатываемых данных, к наиболее эффективным методам относится разработка блочных алгоритмов решения задач;

- одной из основных задач в реализации блочных алгоритмов решения задач, помимо разработки самого алгоритма, является задача подбора набора параметров алгоритма, которые дадут максимальную производительность на ЦВА; здесь известны несколько подходов:

- множественная реализация и эмпирический поиск;
- генерация кода и автоматическая настройка;
- поиск параметров алгоритмов на основе аналитической модели.

Несмотря на то, что данные методы не являются новыми, они лежат в основе технологии кэш-оптимизации ПС и постоянно совершенствуются. При этом, несмотря на свою относительную эффективность, данные методы имеют существенные недостатки. Абсолютное большинство предложенных методов направлено на решение задач BLAS, в основном задачи умножения плотных матриц. Апробация методов также в основном осуществлялась на решении задач BLAS. Описанные подходы часто не формализованы, для других преобразований необходимо проводить похожее исследование. Отмечается также сложность проведения подобного анализа, как и в случае описанных ранее моделей. Кроме того, предложенные методы получения характеристик проектируемых алгоритмов предполагают аналитическое моделирование, при этом моделирование часто проводится для конкретного

семейства МП, авторы методик не предложили информационных систем, автоматизирующих данный процесс.

2.3 Методы повышения эффективности векторизации вычислений

В области векторизации вычислений можно выделить несколько направлений исследований: способы векторизации, векторизация конкретных алгоритмов и задач, автоматическая векторизация вычислений. В первой группе можно выделить публикации, в которых авторы исследуют способы векторизации вычислений [92, 103], в т.ч. использование современных оптимизирующих компиляторов, использование директив компилятора, интринсиков, непосредственного использования команд МП, другие подходы к векторизации. Ко второй группе можно отнести большое число исследований, можно сказать, что основная часть работ по векторизации относится к этой группе. В качестве примера можно привести работы по векторизации реализаций математических преобразований [23], моделирования различных процессов [79], других систем [106],

Все «ручные» способы векторизации являются достаточно трудоемкими, требуют высокой квалификации разработчика, знания архитектуры ВС, используемого SIMD-расширения. Описанные в научной литературе алгоритмы векторизации в применении к конкретным задачам также используют достаточно трудоемкие способы повышения эффективности ПС, которые сложно или невозможно перенести на другие задачи,

В связи с этим появился запрос на разработку методов и программных средств, предназначенных для поддержки и автоматизации процесса векторизации кода. Существует целый ряд ПС, которые позволяют проанализировать эффективность векторизации кода, но уже на этапе анализа отлаженного скомпилированного приложения. Автоматическая векторизация поддерживается большинством современных компиляторов.

Основные методы, используемые при построении блоков автоматической векторизации и в целом для повышения эффективности векторизации вычислений: реализация параллелизма между итерациями циклов (векторизация циклов), векторизация обработки чередующихся данных (interleaved data), исследование вопросов выравнивания данных (рисунок 2.8).



Рисунок 2.8 - Методы, используемые при построении блоков автоматической векторизации

В работе [84] описывается подход к векторизации вычислений, названный авторами SLP, представляющий собой усовершенствованный метод традиционной векторизации, который облегчает параллелизм между итерациями цикла. Векторизация в данном контексте представляет собой процесс, используемый для одновременного выполнения нескольких похожих задач (или инструкций), что позволяет экономить время и ресурсы на обработку [21].

Основные шаги в рамках данного подхода (рисунок 2.9):

- поиск изоморфных операций;
- упаковка данных в векторные регистры;
- раскрутка цикла.

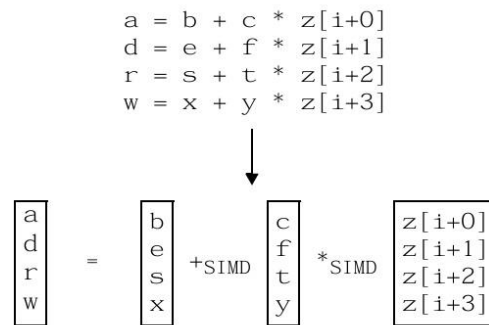


Рисунок 2.9 – Векторизация цикла с использованием SLP [84]

Данный подход является базовым для построения блоков автоматической векторизации. При этом наибольшая производительность наблюдается в случае, если упаковка данных не требуется, т.е. данные расположены в памяти непрерывно (stride-one) и естественным образом «упаковываются» при загрузке из памяти.

Предлагаются и другие модели автоматической векторизации, в частности, в работе [22] предлагается подход, названный авторами LLP, который в отличие от SLP позволяет выполнить векторизацию кода не только для групп изоморфных операций. В данном случае кроме традиционных SIMD-операций из наборов векторных инструкций авторы используют команды, которые они относят к группе команд Single Instruction Multiple Operation Multiple Data (SIMOMD), к которым они относят, например, команды горизонтального сложения/вычитания, команды преобразования данных, сложения/умножения с накоплением и ряд других команд. Данные команды объединяют данные из различных «векторных линий выполнения» (lanes), что не осуществляется в рамках SLP.

Также стоит упомянуть подход PSLP, представленный в работе [99], представляющий собой алгоритм векторизации, который также может векторизовать код, содержащий неизоморфные последовательности команд.

Алгоритм векторизации с параллелизмом на уровне SLP является основным способом автоматической генерации векторного кода, он реализован во всех современных компиляторах. При этом SLP основан на поиске последовательностей изоморфных команд для объединения в

векторы, что не всегда возможно. PSLP вводит минимальное количество избыточных команд в код для преобразования неизоморфных последовательностей в изоморфные, после чего дополненная последовательность команд может быть успешно векторизована.

Большое внимание уделяется проблеме выравнивания данных. Изначально данная проблема стояла остро при генерации кода для архитектур, позволяющих выполнять загрузки и сохранение только выровненных данных. Однако и для современных архитектур, в которых работа с невыровненными данными может приводить к временным «штрафам», данная проблема является актуальной.

Так, в работе [34] предлагается метод компиляции исходного кода с блоком автовекторизации, учитывающим работу с невыровненными данными. При этом реализуется автоматическая реорганизация данных в векторных регистрах с применением методов, позволяющих свести к минимуму количество подобных генерируемых операций по реорганизации данных.

В качестве примера рассматривается цикл

$$a[i+3]=b[i+1]+c[i+2]$$

В нем нет цикловых зависимостей по данным, поэтому он может быть векторизован. При этом очевидно, что существуют проблемы с выравниванием (рисунок 2.10).

Авторы сформулировали ограничения, которым должна удовлетворять корректная с точки зрения выравнивания векторизация кода.

1. При выполнении векторной загрузки 16-байтовое выравнивание адреса загрузки определяет байтовое смещение данных в целевом векторном регистре. Например, 16-байтовое выравнивание $b[1]$ и $c[2]$ в памяти составляет 4 и 8 байт соответственно (предполагается выравниванием первых элементов массивов), как и байтовое смещение в соответствующих векторных регистрах.

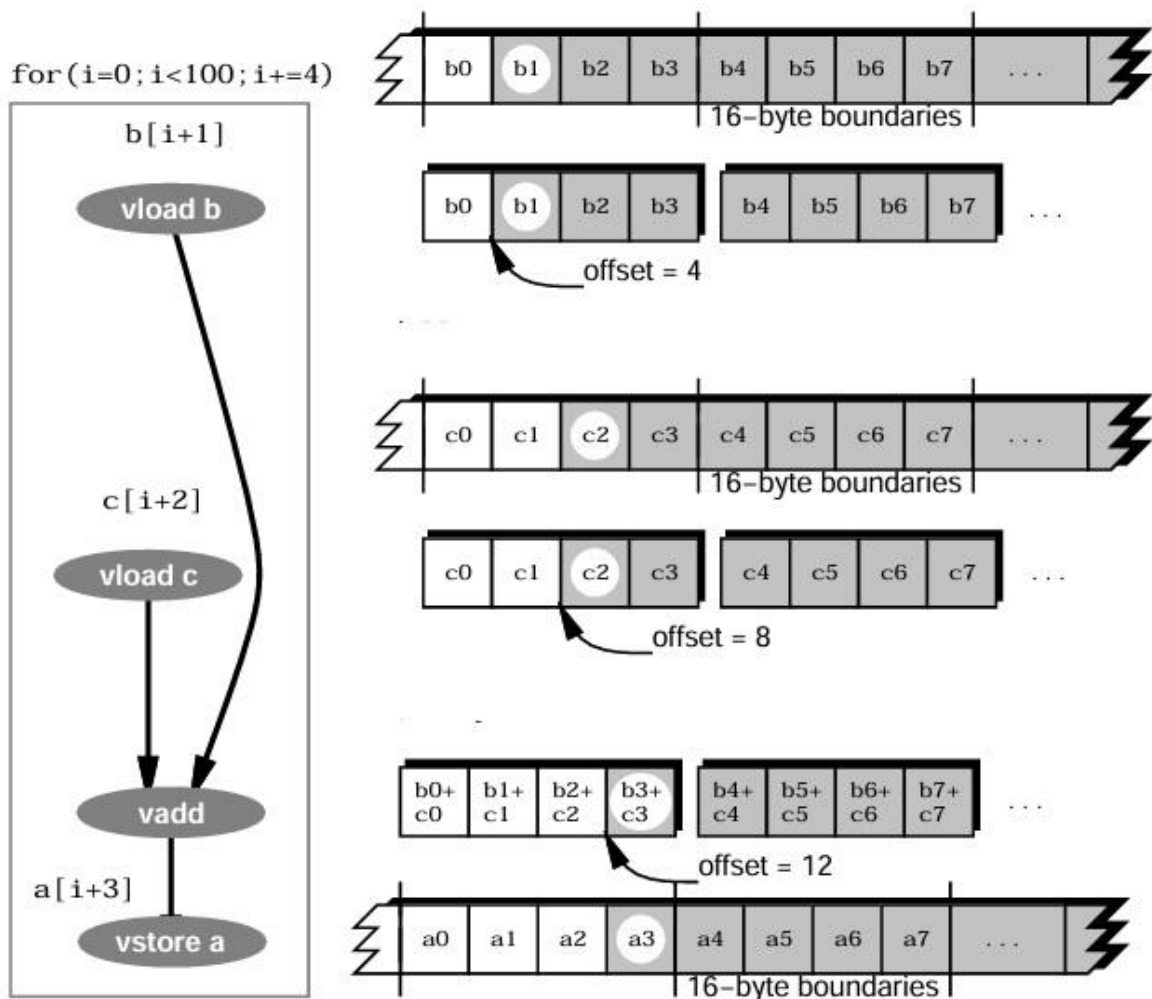


Рисунок 2.10 – Схема реализации цикла [34]

2. При вычислении векторных операций (возможно, исключая операции переупорядочивания данных) данные, задействованные в исходной операции, должны находиться с одинаковым байтовым смещением в соответствующих векторных регистрах.

3. При выполнении векторного сохранения данных байтовое смещение исходного векторного регистра должно быть одинаковым, данные в векторном регистре должны соответствовать выравниванию адреса в памяти. Например, $b[1]+c[2]$ при сохранении в $a[3]$ должны находиться со смещением в байтах 12 в своем векторном регистре, чтобы соответствовать выравниванию в памяти $a[3]$.

Другими словами, данные загружаются и сохраняются «как надо», все преобразования, связанные с выравниванием выполняются в векторных регистрах. Вводится понятие потока памяти (memory stream),

представляющего собой данные, расположенные в памяти при непрерывном доступе в цикле и регистрового потока (register stream), представляющего собой эти данные, загруженные в векторные регистры в процессе обработки в цикле. При этом появляется понятие смещения потоков (stream offset) (рисунок 2.11).

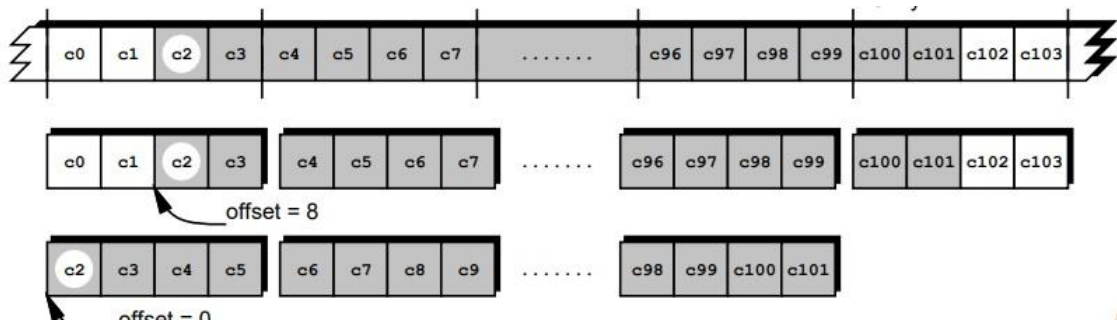


Рисунок 2.11 – Реорганизация данных в регистровом потоке [34]

Для реорганизации данных формируется граф реорганизации данных, в соответствии с которым осуществляется сдвиг данных в потоках, при этом рассматриваются различные политики сдвига (Shift Policy).

Данный подход был оправдан для архитектур, где возможны только выровненные загрузки и сохранения, однако, для архитектур, где возможны невыровненные операции с памятью, которые могут приводить к временным штрафам, подобный подход не является эффективным.

При этом описанный подход может использоваться для решения задач, где выравнивания известны на этапе компиляции. Однако это не всегда возможно, и часто выравнивание данных становится известно в процессе выполнения программы. Данный факт может являться следствием особенностей используемых алгоритмов или неспособности компилятора сделать вывод о выравнивании данных. В работе [135] предлагается новый метод векторизации циклов с выравниванием во время выполнения, который почти так же эффективен, как и циклы с невыровненными данными, известными во время компиляции.

Основной алгоритм автоматической векторизации, основанный на SLP, в качестве одного из шагов реализует упаковку данных в векторные

регистры. При этом в случае «естественной упаковки» при загрузке непрерывно расположенных данных достигается максимальная производительность. Однако данные, используемые в векторизуемых операциях, необязательно должны быть расположены непрерывно. В данном случае их необходимо реально упаковывать, тратя на это вычислительные ресурсы. Падения производительности в данном случае неизбежно даже в случае использования специальных инструкций сбора (gather) и распределения (scatter) данных.

Соответственно, существует большое количество исследований, посвященных векторизации данных, не расположенных в памяти непрерывно, а в соответствии с некоторым шаблоном доступа в память. В работе [97] рассматривается проблема автоматической векторизации обработки несмежных данных. Проанализирован частный случай обработки данных, расположенных не смежно, но регулярно с шагом, равным степени 2, для данного случая предложен эффективный алгоритм автоматической векторизации, включающий в себя динамическую реорганизацию данных.

В качестве промежуточного итога можно сделать следующие выводы:

- в области векторизации вычислений можно выделить несколько направлений исследований: способы векторизации, векторизация конкретных алгоритмов и задач, автоматическая векторизация вычислений;
- в области автоматической векторизации доминирующим подходом является SLP, однако используются и другие подходы, например LLP и PSLP;
- для задачи автоматической векторизации актуальными являются задачи векторизации циклов с несмежными и невыровненными данными;
- задача эффективной векторизации обработки невыровненных данных на настоящий момент не имеет общего решения.

2.4 Инструментальные средства проектирования и анализа ПС

Наиболее распространенным подходом к улучшению показателей эффективности ПС, в т.ч. для ВПВ, является использование специализированных инструментальных средств, основные группы которых представлены на рисунке 2.12.

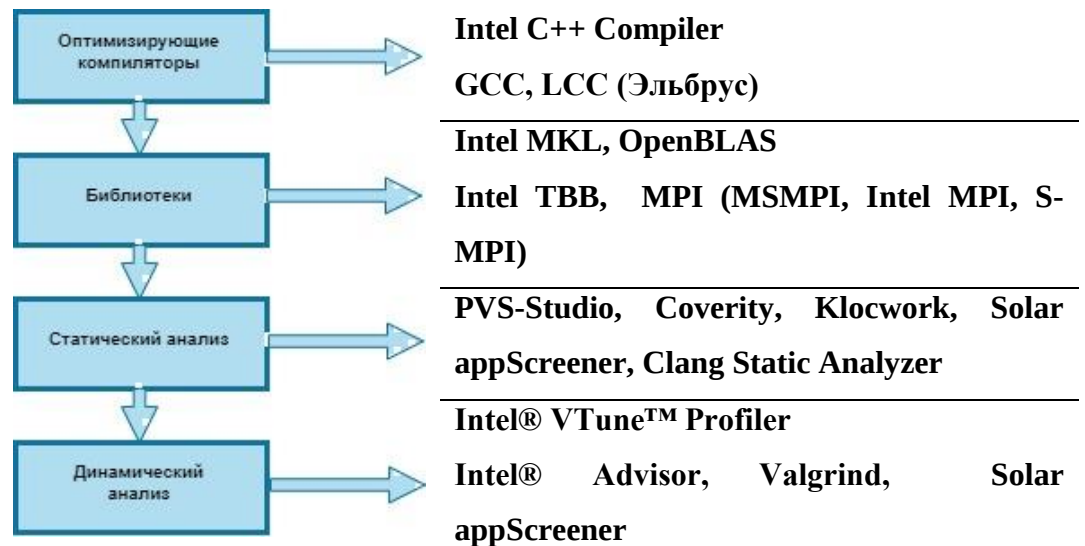


Рисунок 2.12 – Основные группы инструментальных средств, используемых для улучшения показателей эффективности ПС.

Оптимизирующие компиляторы являются мощным инструментом, предназначенным для создания эффективного программного кода. Современные оптимизирующие компиляторы представляют собой сложную ПС, управлять которой можно с помощью специальных опций, влияющих на конечный результат. Так, например, можно влиять на уровень оптимизаций, от самых простых и примитивных оптимизаций до экстремальной оптимизации, включающей в себя и новые экспериментальные методы. Можно оптимизировать скорость работы проектируемой ПС, ее размер. Существует масса различных опций, которые отличаются для разных компиляторов. В качестве примера подобных компиляторов можно привести Intel C++ Compiler [61], GCC [110], отечественную разработку LCC для платформы «Эльбрус» [186].

Современные оптимизирующие компиляторы создают эффективный программный код, однако эффективность подобного кода, особенно в области решения инженерных задач, может оказаться недостаточной. В этом случае разработчики используют такие инструментальные средства, как библиотеки и средства анализа ПС.

Программные библиотеки представляют собой набор готовых решений в виде классов, методов, функций и т.д., предназначенных для решения определенных задач. С одной стороны, использование подобных библиотек значительно сокращает время разработки ПС. Но в данном случае речь идет о библиотеках для ВПВ, код которых был оптимизирован в процессе разработки и имеет высокую эффективность. Большинство подобных библиотек имеют различные реализации одних и тех же функций для различных вычислительных архитектур или настраиваются на ЦВА. Соответственно, разработчику предоставляется программное решение, оптимизированное под конкретную вычислительную архитектуру. Использование таких библиотек позволяет создавать эффективные ПС. К подобным библиотекам можно отнести, например, BLAS-библиотеки: Intel MKL [62], OpenBLAS [98], библиотеки, предназначенные для разработки параллельных программ: Intel TBB [64], различные реализации программного интерфейса Message Passing Interface (MPI), такие как MSMPI [96], Intel MPI[63], S-MPI [151] (отечественная реализация стандарта MPI-2), а также огромное количество других библиотек.

Следующую группу инструментальных средств, используемых для повышения эффективности ПС, составляют средства анализа ПС. Анализ ПС необходим для выполнения оптимизации ПС, выявления ошибок, дефектов и уязвимостей ПС, связанных с ее надежностью, безопасностью, работоспособностью, корректностью выдаваемых результатов. Различают статический и динамический анализ ПС (глава 1). Статический анализ выполняется без компиляции и выполнения программы путем анализа

исходного кода, динамический анализ проводится путем исполнения программы.

Динамический анализ кода реализуется во время исполнения программы, поэтому для проведения подобного анализа необходимо иметь в наличии скомпилированный вариант анализируемой ПС. В качестве одного из мощных средств динамического анализа ПС можно указать Intel Parallel Studio, позволяющего детально анализировать код программы и определять ее узкие места, мешающие эффективно использовать ресурсы ВС и снижающие ее производительность [15]. В таблице 2.1 приведен набор инструментов, которые включает Intel Parallel Studio.

Таблица 2.1. Компоненты Intel Parallel Studio.

Компонент	Описание
Intel Advisor	Выполняет анализ возможности распараллеливания кода на МП Intel, осуществляет поиск подходящих участков и оценку эффекта, выявление узких мест в коде, замедляющих выполнение приложения, обнаружение ряда ошибок.
Intel Composer	Представляет собой обширный набор компиляторов и библиотек C++ и Fortran, позволяющий создавать высокопроизводительные приложения, максимально эффективно использующие аппаратные ресурсы современных МП Intel. В его состав входит ряд расширений, упрощающих разработку и отладку параллельных программ.
Intel Inspector	Инструмент для обнаружения и устранения дефектов в многопоточных параллельных программах. Выполняет анализ корректности и надёжности кода, что помогает предотвращать критические ошибки, снижать риски и повышать стабильность приложений. Позволяет выявлять взаимные блокировки, состояния гонки и ошибки работы с памятью, критичные для работоспособности программы.
Intel Amplifier VTune	Инструмент для оценки производительности программ на МП Intel. Поддерживает профилирование кода различными методами.

Инструменты динамического анализа нельзя использовать на начальных этапах проектирования ПС, для этой цели применяется статический анализ кода. Например, инструмент Clang Static Analyzer [7, 8] представляет собой статический анализатор, который находит ошибки в C, C++ и Objective-C программах. Он является частью проекта LLVM [83] и

обеспечивает глубокий анализ кода, выявляя сложные ошибки и уязвимости. Поиск потенциальных ошибок анализатором осуществляется путём обхода возможных сценариев выполнения программы. Анализатор реализован как библиотека C++, которую можно использовать другими инструментами и приложениями.

Для анализа поведения ПС в различных ВС используют специальные инструментальные средства, моделирующие ЦВА. Данные инструментальные средства получили название симуляторов или эмуляторов. Существуют различные типы симуляторов: от инструментов, моделирующих поведение всего МП, включая иерархию кеш-памяти, ядра и внутрикристальные соединения, другие могут поддерживать только иерархию кеш-памяти. Данные симуляторы используются как для исследования эффективности ПС на ЦВА, так и для оценки эффективности ВС или отдельной ее части, например, иерархической подсистемы памяти. Создание прототипа новой системы является дорогостоящим способом, реализация новой подсистемы в симуляторе является более дешевым и доступным способом оценки.

Архитектурные симуляторы, в т.ч. кеш-симуляторы, используемые для изучения поведения ПС в подсистеме памяти с иерархической структурой, делят на два класса: функциональные симуляторы и симуляторы производительности [16].

Функциональные симуляторы моделируют операции, выполняемые ЦВА в процессе исполнения ПС, при этом никак не учитываются нефункциональные характеристики, такие как время выполнения программы, потребляемая электроэнергия. Функциональное моделирование используется для выполнения статической оценки ПС путем подсчета определенных событий, например, кеш-промахов. Функциональное моделирование используется также для формирования трасс (последовательностей) выполняемых операций и для проверки функциональной корректности моделируемого проекта.

Симуляторы производительности (timing simulator) отслеживают время, необходимое для выполнения каждой операции. Подобные симуляторы представляют собой сложные инструменты, моделирующие машинные циклы МП и включают в себя подробную микроархитектурную модель. Такой уровень детализации значительно увеличивает время моделирования [66], тем самым ограничивая количество инструкций и конфигураций моделируемой платформы [19.]. Компании, разрабатывающие коммерческие архитектуры, такие как ARM, имеют внутренние симуляторы, моделирующие процесс выполнения программы с точностью до цикла. Данные симуляторы необходимы для проверки проектов аппаратного обеспечения перед их изготовлением и для точной оценки новых идей перед их использованием в производственных системах [31]. Такие симуляторы требуют гораздо больше усилий для разработки и недоступны в академических кругах.

Выделяют следующие режимы симуляции:

-trace-driven; симуляторы, управляемые трассами. Симулятор принимает в качестве входных данных файл трассировки, созданный в результате выполнения или эмуляции программы. Управляемое трассировкой моделирование полностью устраняет издержки, связанные с выполнением входной программы; однако создание, поддержание и использование больших файлов трассировки приводит к дополнительным затратам. Файл трассировки может содержать либо динамические инструкции, либо описания операций доступа к памяти, в зависимости от назначения симулятора. Симуляторы, управляемые трассировкой, как правило, проще в разработке и реализации. Они позволяют моделировать ПС на несовместимых системах, поскольку форматы файлов трассировки, как правило, не зависят от платформы, на которой программа была запущена. После создания файла трассировки при запуске программы моделирование можно выполнять много раз, ориентируясь на различные конфигурации, без необходимости повторного запуска программы.

- execution-driven. Симулятор запускает входную программу, генерируя некоторым образом файл трассы «на лету». При этом в анализируемую программу может вставляться дополнительный код, необходимый для генерации трасс. Команды программы могут прямо выполняться на той же ВС, на которой запущен симулятор, при этом ряд команд может эмулироваться. Эмуляция необходимо в том случае, если целевая платформа не может выполнять все команды анализируемой ПС. Кроме того, ЦВА может эмулироваться полностью. В целом такой подход имеет ряд преимуществ, в частности нет необходимости в генерации больших файлов трасс. Однако подобные симуляторы является гораздо более сложными для реализации.

2.5 Основные результаты и выводы по главе 2

Выполнен обзор современных методов и инструментальных средств, применяемых для повышения эффективности ПС в процессе их проектирования.

Рассмотрены методики разработки ПС, основанные на моделях (Model-Driven Engineering). Рассмотрены подходы, основанные на моделировании “черного ящика”, которое опирается на статистические методы и машинное обучение, моделировании “белого ящика”, которое использует упрощенные математические модели для описания взаимодействия кода с оборудованием. В качестве примеров рассмотрены модели, обеспечивающие точное предсказание производительности одного ядра МП и масштабирования вычислений, модели ВС с иерархической структурой памяти. К недостаткам большинства подобных моделей можно отнести привязку к конкретному МП и сложность проведения аналитического моделирования, а также необходимость проведения отдельного исследования для каждого нового алгоритма.

Рассмотрены методы улучшения стратегии кеширования данных. Помимо общих рекомендаций, выработанных сообществом разработчиков в соответствии с архитектурными особенностями ВС и основанных на повышении пространственной и временной локальности обрабатываемых данных, особый акцент делается на разработке блочных алгоритмов решения задач. Показано, что одной из основных задач в реализации блочных алгоритмов решения задач, помимо разработки самого алгоритма, является задача подбора набора параметров алгоритма, которые дадут максимальную производительность на ЦВА. Отмечается сложность данной задачи, предложенные методы получения оптимальных характеристик проектируемых алгоритмов предполагают аналитическое моделирование, при этом моделирование часто проводится для конкретного семейства МП, авторы методик не предложили информационных систем, автоматизирующих данный процесс.

Рассмотрены методы повышения эффективности векторизации вычислений. В данной области можно выделить несколько направлений исследований: способы векторизации, векторизация конкретных алгоритмов и задач, автоматическая векторизация вычислений. В области автоматической векторизации доминирующим подходом является SLP, однако используются и другие подходы, например LLP и PSLP. Для задачи автоматической векторизации актуальными являются задачи векторизации циклов с несмежными и невыровненными данными, при этом задача эффективной векторизации обработки невыровненных данных на настоящий момент не имеет общего решения.

Рассмотрены инструментальные средства проектирования и анализа ПС, выделены основные классы данных инструментальных средств, описаны области их применения.

3 МЕТОДОЛОГИЯ ПРОЕКТИРОВАНИЯ ЭФФЕКТИВНЫХ ПРОГРАММНЫХ СИСТЕМ ДЛЯ РЕШЕНИЯ ИНЖЕНЕРНЫХ ЗАДАЧ В ГЕТЕРОГЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СРЕДАХ

Глава содержит описание совокупности моделей и методов поддержки ЖЦ проектируемых ПС для решения инженерных задач в ГВС, реализующая комплексный подход к повышению эффективности программных систем

В разделе 3.1 представлена модель улучшения показателей эффективности программных систем, основанная на методах статического анализа исходного кода и оценки аппаратурно-временной сложности ПС, отличающаяся от известных переносом этапа анализа эффективности ПС с динамического анализа готового приложения на статический анализ исходного кода, позволяющая снизить количество итераций проектных процедур на этапах разработки и модификации жизненного цикла ПС (п.1 научной новизны, пп.3,10 паспорта специальности). Представляет собой модель итеративного процесса разработки и модификации программных систем и используется для повышения эффективности ПС на этапах разработки и модификации.

В разделе 3.2 представлена математическая модель оценки эффективности проектируемых ПС в ГВС с различными характеристиками, позволяющая на различных этапах ЖЦ в рамках модели улучшения показателей эффективности ПС комплексно оценить ПС при эксплуатации в гетерогенных вычислительных средах. Модель основана на классификации подсистем целевой вычислительной архитектуры и отличается от известных совокупностью новых метрик ПС и обоснованием их применимости, (п.2 научной новизны, п.6 паспорта специальности).

В разделе 3.3 представлен аналитический метод определения максимального размера блока при регулярном доступе к данным. Метод основан на отображении множества кешируемых данных на множества кеш-памяти и отличается от известных прямым способом расчета параметров

регулярных шаблонов доступа к данным, что позволяет выбирать параметры алгоритмов, минимизирующие количество кеш-промахов при доступе к данным проектируемой ПС. (п.3 научной новизны, п.6 паспорта специальности).

В разделе 3.4. представлена имитационная модель исполнения программы в ГВС с иерархической структурой памяти, используемая для определения характеристик ПС на конкретной ЦВА, в т.ч. метрик ПС в рамках модели оценки эффективности проектируемых ПС. Модель основана на множествах системных и пользовательских операций, впервые использован полный набор параметров, характеризующих различные уровни подсистемы памяти, что позволяет определять характеристики ПС на конкретной ЦВА, а также характеристики подсистемы памяти для проектируемой ПС (п.4 научной новизны, п.8 паспорта специальности)

В разделе 3.5 представлен комплекс методов повышения эффективности ПС (п.5 научной новизны, п.6 паспорта специальности): использования подсистемы памяти, в котором впервые реализован аналитический подход к оценке эффективности проектируемых ПС на базе совокупности метрик ПС, векторизации вычислений, в котором впервые реализован новый способ векторной обработки данных, что позволяет избежать потерь производительности ПС в ГВС при доступе к данным, использования GPU, в котором впервые реализован аналитический подход к оценке эффективности проектируемых программных систем на базе совокупности метрик ПС, позволяющий повысить уровень загрузки и степень сбалансированности загрузки потоковых процессоров.

В разделе 3.6 представлена методология проектирования эффективных ПС для решения инженерных задач в ГВС, основанная на исследовании жизненного цикла ПС, впервые реализующая подход непрерывного улучшения показателей проектируемых ПС. Методология отличается от известных реализацией предложенных моделей и методов и позволяет принимать проектные решения по эффективному использованию

вычислительных ресурсов на начальных этапах жизненного цикла ПС (п.6 научной новизны, пп.1,4 паспорта специальности).

В 3.7 приводятся основные результаты и выводы по главе 3.

3.1 Модель улучшения показателей эффективности программных систем

Предлагаемая модель улучшения показателей эффективности ПС, представляющая собой модель итеративного процесса разработки (рисунок 3.1) и модификации (рисунок 3.2) ПС, является частью предлагаемой методологии проектирования эффективных ПС. Выбор конкретной методологии проектирования ПС зависит от ряда факторов, таких как тип проектируемой ПС, требования, предъявляемые к ней, тип ВС, в которой предполагается эксплуатация проектируемой ПС. В данной работе рассматривается проблема проектирования ПС, предназначенных для решения научных и инженерных задач, для которых важны:

- выбор алгоритма решения задачи;
- повышение эффективности ПС.

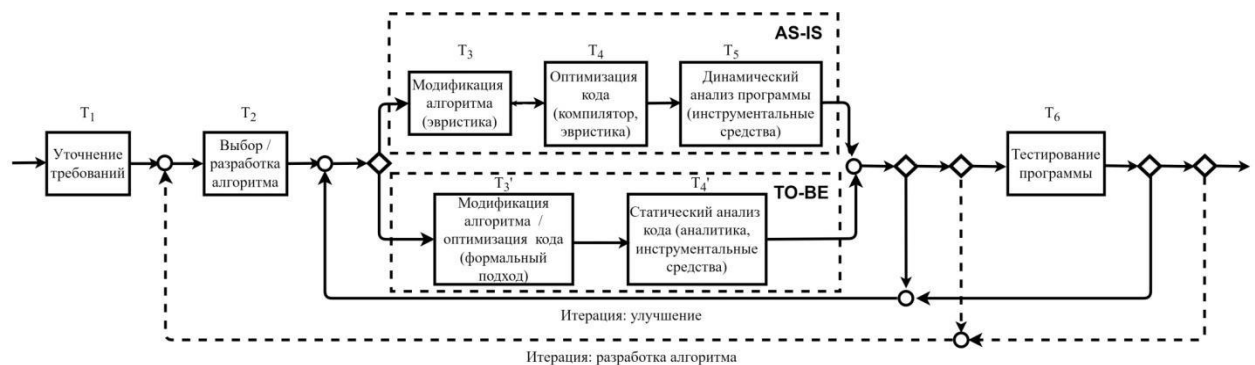


Рисунок 3.1 – Этап разработки ПС в рамках модели улучшения показателей эффективности ПС

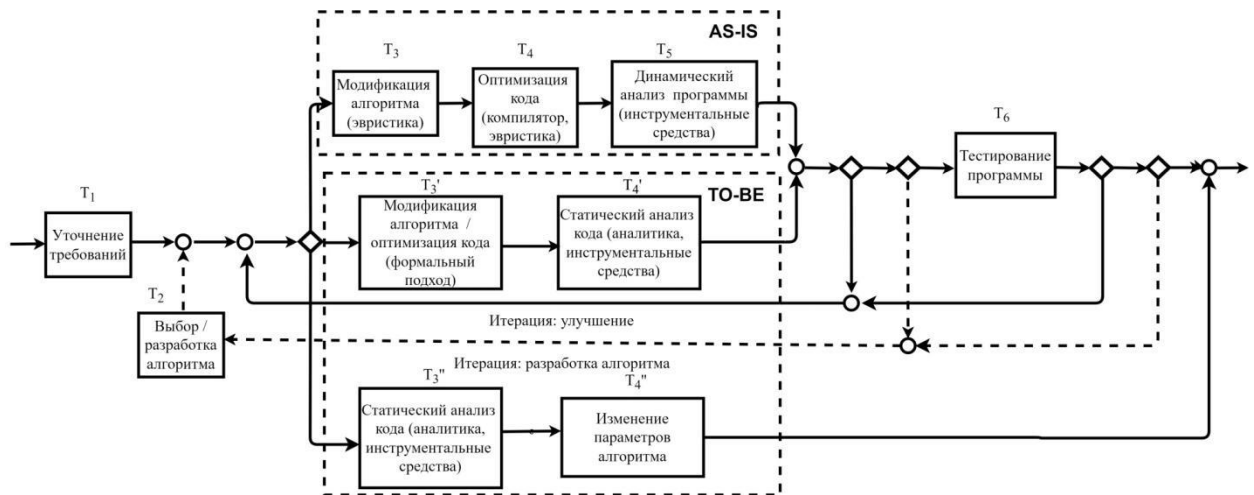


Рисунок 3.2 – Этап модификации ПС в рамках модели улучшения показателей эффективности ПС

Предлагаемую модель можно описать следующим образом (3.1).

$$E = \langle T, R \rangle \quad (3.1)$$

$$E = f(a), a = (a_1, a_2, \dots, a_n) \in A$$

$$E_{i+1} \geq E_i$$

Здесь E – эффективность;

T – затрачиваемое время;

R – затрачиваемые ресурсы.

Эффективность ПС на этапах разработки и модификации зависит от времени и ресурсов, затраченных на реализацию этапа, в то же время эффективность разработки зависит от программной реализации решаемой задачи из множества A программных реализаций. Модель гарантирует непрерывное улучшение показателей эффективности проектируемой ПС, другими словами эффективность ПС на итерации разработки ($i+1$) будет не хуже эффективности на итерации i .

3.1.1 Исследование жизненного цикла программной системы

Для решения научно-технической проблемы исследования был проведен анализ ЖЦ ПС в аспекте создания и эксплуатации ПС в области

ВПВ для решения инженерных и научных задач, для которых важными являются вопросы создания эффективных ПС.

ЖЦ, представленный на рисунке 3.3, состоит из трех этапов: разработки, модификации и эксплуатации.



Рисунок 3.3 — Жизненный цикл ПС

После окончания этапа разработки ПС переходит на этап эксплуатации. В процессе эксплуатации может возникнуть необходимость в модификации ПС по следующим причинам:

- недостаточная эффективность ПС;
- необходимость адаптации ПС под новые вычислительные архитектуры;
- выявленные ошибки.

Целью разработки модели улучшения показателей ПС является повышение их эффективности на каждом этапе ЖЦ, снижение затрат на разработку, модификацию и эксплуатацию ПС. В связи с этим существует необходимость в количественной оценке эффективности ПС.

Эффективность ПС на каждом этапе ЖЦ характеризуется кортежем (3.1). Для этапов разработки и модификации время T представляет собой время, затрачиваемое командой разработчиков на выполнение всех работ, необходимых для завершения текущего этапа, R – затрачиваемые ресурсы, в т.ч.:

- затраты на разработчиков (зарботная плата, число занятых разработчиков);
- затраты на оборудование, используемое на этапах разработки или модификации ПС.

На этапе эксплуатации T представляет собой время работы ПС для решения конкретной задачи, R – используемые при этом ресурсы:

- состав аппаратных средств;
- потребляемые энергоресурсы.

Целевыми являются ПС, предназначенные для решения инженерных вычислительно-сложных задач, приоритетами для которых являются:

- снижение времени решения задачи (3.2), т.е. поиск среди множества A программных реализаций такой реализации, время решения для которой не будет превышать времени решения для любой другой реализации;

$$t = f(a), a = (a_1, a_2, \dots, a_n) \in A \quad (3.2)$$

$$\operatorname{argmin} f(a) \in \{a | \forall b : t_b \geq t_a\}$$

- уменьшение количества используемых ресурсов.

3.1.2 Мера аппаратурно-временной сложности

Важной характеристикой современных ВС, в т.ч. ГВС, является соотношение производительности системы в целом к ее стоимости. В настоящее время высокопроизводительные системы используются во многих отраслях экономики, в т.ч. научных и инженерных расчетах, в процессе моделирования различных процессов, например, аэродинамических, космологических, экономических и т.д. Отдельно стоит отметить бурно развивающиеся в настоящее время технологии ИИ. При этом системы ИИ являются ресурсоемкими и вопрос повышения производительности ВС в данной отрасли информационных технологий также является крайне актуальным.

Возможны различные пути повышения производительности. Возьмем идеальную ВС, которая характеризуется:

- наличием неограниченного количества ВУ (процессоров, ядер многоядерных процессоров и т.д.);
- наличием неограниченного количества ресурсов (памяти и т.д.);

- отсутствием накладных расходов.

В данной системе уменьшить время выполнения программы можно за счет пропорционального увеличения числа используемых ресурсов:

- использования более производительного вычислительного узла;
- распараллеливания вычислений путём одновременного использования нескольких вычислительных узлов для решения задачи.

При этом стоит отметить, что ни один из данных способов не позволит неограниченно уменьшать время выполнения программы. В первом случае это связано с невозможностью неограниченно увеличивать производительность ВУ. Во втором случае неограниченно уменьшать время выполнения программы невозможно из-за ограниченного числа параллельных ветвей алгоритма и ростом доли накладных расходов [189]. Основными параметрами, характеризующими параллельную программу, являются ускорение (3.3) и эффективность (3.4).

$$S_p = \frac{t_1}{t_p} \quad (3.3)$$

где S_p – ускорение программы, запущенной на p вычислительных узлах;

t_1 – время работы самого быстрого доступного варианта программы на одном вычислительном узле;

t_p – время работы на p вычислительных узлах.

Следует отметить, что ускорение является свойством не абстрактного алгоритма решения задачи, а его реализации в виде конкретной программы, выполняемой в конкретной ВС.

$$E_p = \frac{S_p}{p} \quad (3.4)$$

При неограниченном росте числа узлов типичные зависимости эффективности и ускорения параллельных программ представлены на рисунке 3.4.

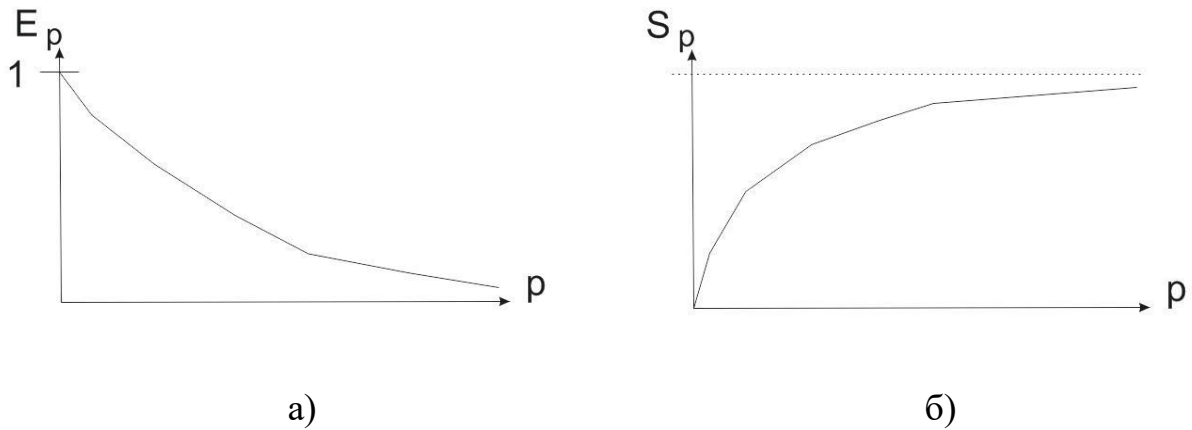


Рисунок 3.4 – Типичные зависимости эффективности (а) и ускорения (б) параллельной программы от числа узлов.

Замедление роста ускорения с ростом числа ВУ связано с рядом факторов:

- число параллельных ветвей алгоритма решения любой задачи не является бесконечным, поэтому в ситуации, когда число узлов начинает превышать число параллельных ветвей, ускорение перестает расти;

- с ростом числа ВУ растет количество накладных расходов;

Ограничение роста ускорения также связано с ограничением закона Амдала (1.3): «в случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого медленного фрагмента» [3]. Выражение (1.3) можно получить следующим образом (3.5).

$$S_p \leq \frac{(1-p) + p}{\frac{(1-p)}{1} + \frac{p}{n}} = \frac{1}{(1-p) + \frac{p}{n}} \quad (3.5)$$

где p – распараллеливаемая часть программы.

Очевидным следствием из закона Амдала является утверждение, что максимальное ускорение параллельной программы ограничивается ее последовательной частью (3.6).

$$\lim_{n \rightarrow \infty} S_n = \frac{1}{(1-p)} \quad (3.6)$$

Несмотря на то, что приведенные выше способы могут уменьшить время решения задачи, они не приведут к повышению эффективности ПС в контексте выражения (3.1), т. к. уменьшение одного показателя (Т) и соответствующее увеличение другого показателя (R) оставляют результирующую эффективность неизменной.

Для количественной оценки степени эффективности ПС будем использовать меру аппаратно-временной сложности задачи (алгоритма) [163] (3.7).

$$C = O(Q) * O(T) \quad (3.7)$$

где C – мера аппаратно-временной сложности;

$O(Q)$ – мера аппаратурной сложности реализации алгоритма;

$O(T)$ – мера временной сложности реализации алгоритма.

В качестве иллюстрации приведем пример оценки аппаратно-временной сложности алгоритма на базе задачи суммирования двух пар чисел. Даже для такой простой задачи при оценке аппаратно-временной сложности необходимо определить тип ВС, в которой будет осуществляться ее решение. Предположим, что используется аппаратное решение на одноразрядных полных двоичных сумматорах в составе более сложной системы, архитектура которой сейчас не важна. Тогда возможны два варианта, представленные на рисунке 3.5.

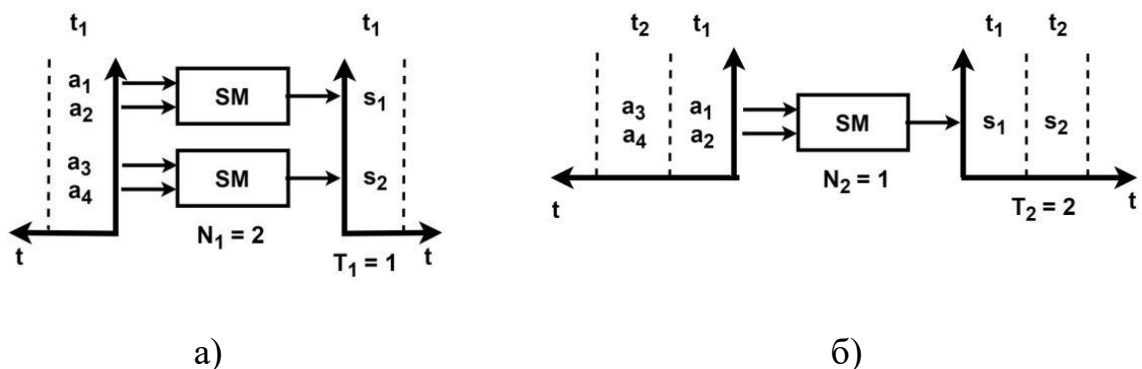


Рисунок 3.5 — Реализация задачи сложения чисел.

На рисунке 3.5 а) суммирование осуществляется на двух сумматорах за один такт, таким образом, мера аппаратурной сложности равна двум, а мера временной сложности равна единице. На рисунке 3.5 б) суммирование

осуществляется на одном сумматоре за два такта, таким образом, мера аппаратной сложности равна единице, а мера временной сложности равна двум. Полученная мера аппаратурно-временной сложности иллюстрируется на рисунке 3.6.

Здесь по оси X откладывается число единиц используемого оборудования, по оси Y — время, необходимое для решения задачи. Видно, что с увеличением числа единиц оборудования время решения задачи снижается, однако, площади прямоугольников, образованных парами $\langle N_1, T_1 \rangle$ и $\langle N_2, T_2 \rangle$ остаются равными. Очевидно, в реальной системе площади данных прямоугольников совпадать не будут. Основная причина этого — наличие накладных расходов.

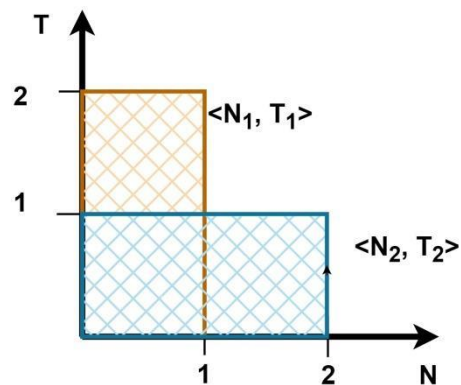


Рисунок 3.6 – Мера аппаратурно-временной сложности

Однако в идеальной ВС, которая была рассмотрена выше, можно принять данное допущение. Для оценки аппаратурно-временной сложности задачи на ГВС, имеющих в своем составе CPU, GPU и другие вычислители, в качестве меры временной сложности можно использовать время решения задачи в секундах. Для оценки меры аппаратурной сложности различных ВС могут быть использованы характеристики, применяемые для сравнения данных ВС, например, производительность или стоимость. В каждом конкретном случае решение о способе определения аппаратурной сложности ГВС с различными архитектурами принимает разработчик.

Так как меры аппаратурной и временной сложности являются характеристиками алгоритмов, имеющими разную природу, определять их

можно путем сравнения относительных показателей различных анализируемых алгоритмов решения необходимой задачи.

Пусть имеется n алгоритмов, каждый из этих алгоритмов имеет сложность C_i (аппаратурную или временную). Тогда меру сложности i -го алгоритма будем определять следующим образом (3.8).

$$O(C_i) = \frac{C_i}{\sum_{j=1}^n C_j} \quad (3.8)$$

Из выражения (3.8) следует, что количественная оценка меры аппаратурно-временной сложности имеет смысл только для определенного перечня алгоритмов. При любом изменении списка сравниваемых алгоритмов необходимо провести новую оценку.

Таким образом, повышение эффективности ПС на этапе эксплуатации можно представить в виде задачи снижения аппаратурно-временной сложности программной реализации решаемой задачи. Потребляемые ВС энергоресурсы в процессе решения задачи напрямую зависят от значения ее аппаратурно-временной сложности, т.е. от площади прямоугольника на рисунке 3.6..

Аналогичным образом можно оценить эффективность ПС на этапах разработки и модификации. В данном случае показатель ресурсов является комплексным (3.9) и зависит не только от используемых аппаратных средств (Q), но и от числа занятых работников (P).

$$R = \langle P, Q \rangle \quad (3.9)$$

В каждом отдельном случае данный параметр должен определяться с учетом текущих требований. Например, ситуации, когда один работник использует N узлов вычислительного кластера или N работников используют N персональных компьютеров, отличаются друг от друга в контексте показателя используемых ресурсов. В данном случае добавить какой-то инвариант эффективности сложнее, чем в случае с этапом эксплуатации ПС из-за определенной субъективности в оценке ресурсоемкости. Однако в некоторых частных случаях это можно сделать. Например, если

предположить использование ВС с одинаковыми характеристиками, тогда можно также ввести инвариант ресурсно-временной сложности выполнения этапа жизненного цикла. График данной характеристики будет аналогичен рисунку 3.6, т.е. по оси Y по-прежнему откладывается время выполнения этапа, однако по оси X в данном случае откладывается человеко-машинная сложность (3.10).

$$R = P * Q \quad (3.10)$$

В этом случае задача повышения эффективности этапа разработки сводится к уменьшению параметра ресурсно-временной сложности. Приведем примеры сравнения определения ресурсно-временной сложности:

- выполнение этапа одним работником за 1,5 единицы времени является более эффективным, чем выполнение этапа двумя работниками за 1 единицу времени;
- выполнение этапа одним работником за 3 единицы времени имеет равную эффективность с выполнением этапа тремя работниками за 1 единицу времени.

Очевидно, что в реальности подобный подход не всегда может быть применен напрямую из-за ряда причин:

- наличие различных ВС, используемых разработчиками (различная стоимость, характеристики, энергопотребление);
- наличие накладных расходов, связанных с работой в команде;
- доля субъективизма в оценке меры человеко-машинной сложности задачи;
- наличие различных стратегий в выборе подходящей методологии проектирования ПС.

Однако подобный подход выявляет очевидную тенденцию повышения эффективности этапов разработки и модификации ПС: снижение времени на реализацию данных этапов ЖЦ ПС ведет к повышению эффективности данного этапа в целом.

3.1.3 Анализ этапа разработки жизненного цикла программных систем

На рисунке 3.1 можно выделить основные проектные процедуры:

1. уточнение требований к проектируемой ПС;
2. выбор или разработка алгоритма решения задачи;
3. процесс кодирования (модификация алгоритма при необходимости, разработка, отладка и оптимизация проектируемой программы) и анализа полученной программы;
4. тестирование полученной ПС.

Данная модель отражает разработку ПС, предназначенного для решения научных и инженерных вычислительно-сложных задач, для которых время решения является одним из ключевых параметров эффективности получаемых ПС. Поэтому шаги выбора (разработки) базового алгоритма решения задачи, а также оптимизация проектируемой ПС, являются ключевыми в процессе разработки.

Рассмотрим проектные процедуры этапа разработки ПС подробнее.

1. Уточнение требований.

В данной проектной процедуре осуществляется учет следующих требований:

- требования к проектируемой ПС;
- требования к целевой архитектуре ВС;
- требования к используемому программному обеспечению.

Требования к проектируемой ПС включают в себя описание решаемой проблемы, необходимую точность решения, используемые форматы данных и т.д.. Под проблемой здесь понимается конкретная вычислительная задача, для которой необходимо получить эффективную программную реализацию. Например, разработку библиотеки функций можно рассматривать в данном контексте как совокупность вычислительных задач, для каждой из которых необходимо разработать и оптимизировать программное решение. Модель на рисунке 3.1 в данном случае применяется к решению каждой из задач, для

разработки библиотеки в целом может использоваться любая подходящая модель разработки ПС в команде (каскадная, инкрементная, спиральная и др. модели разработки).

Требования к архитектуре ВС являются крайне важными, т. к. архитектура в данном случае определяет перечень используемых технологий. Например, ПС может разрабатываться для ВС с общей или распределенной памятью, система может иметь в своем составе GPU, FPGA-ускорители и другие особенности ВС.

Требования к используемому программному обеспечению включают в себя, например, требования к используемой ОС, используемым инструментальным средствам (например, компилятору).

2. Выбор или разработка алгоритма.

Данный шаг является крайне важным в процессе решения вычислительно-сложных задач, т.к. в случае наличия нескольких альтернатив выбор верного алгоритма может привести к получению наиболее эффективного варианта ПС и наоборот, выбор неоптимального алгоритма может привести к значительному увеличению времени решения. В качестве примера можно привести задачу решения СЛАУ, для которой известно множество алгоритмов решения. Данные алгоритмы могут в разной степени соответствовать требованиям, учтенным на первом шаге, например, требованиям к архитектуре ВС. Можно привести также и другие примеры, даже примеры относительно простых и широко известных задач. Так, например, алгоритм умножения матриц до сих пор является одним из самых оптимизируемых алгоритмов в области ВПВ. Также стоит отметить, что часто для решения задачи возникает необходимость разработать нового алгоритма или существенной модернизации известных алгоритмов.

3. Процесс кодирования и анализа полученной программы;

Следующая проектная процедура представлена в двух вариантах:

- традиционный вариант (рисунок 3.7);
- предлагаемый вариант (рисунок 3.8).

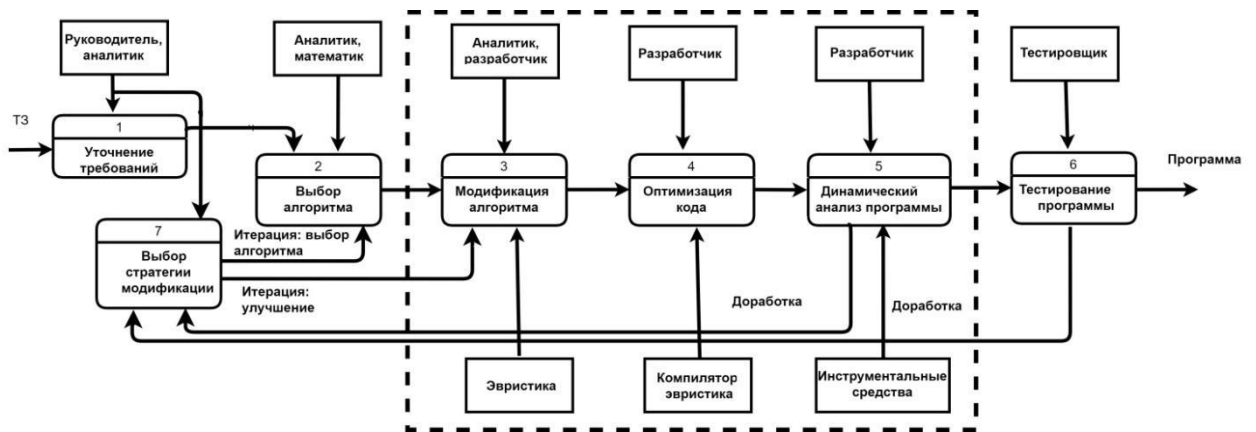


Рисунок 3.7 – Диаграмма этапа разработки ПС. Традиционный подход (AS-IS)

Рассмотрим традиционный подход к разработке ПС для выявления его недостатков.

3.1. Модификация алгоритма.

Из диаграммы на рисунке 3.7 видно, что на этапе разработки в процессе участвуют следующие специалисты:

- руководитель – осуществляет общее руководство разработкой ПС, принимает участие в управлении разработкой ПС на стадиях уточнения требований и выбора стратегии модификации;
- аналитик – принимает участие в управлении разработкой ПС на стадиях уточнения требований, выбора стратегии модификации, выбора алгоритма и модификации алгоритма;
- математик – принимает участие в управлении разработкой ПС на стадии выбора алгоритма;
- разработчик – принимает участие в управлении разработкой ПС на стадиях модификации алгоритма, оптимизации кода и динамического анализа программы;
- тестировщик – принимает участие в управлении разработкой ПС на стадии тестирования программы.

Проектные процедуры уточнения требований и выбора алгоритма были рассмотрены выше. Диаграмма, приведенная на рисунке 3.7, представляет

собой итеративный процесс разработки ПС, причем решение о доработке ПС, заключающейся в проведении новой итерации, принимают:

- разработчик при выявлении неэффективного использования вычислительных ресурсов в процессе выполнения динамического анализа ПС;
- тестировщик при обнаружении ошибок в проектируемой ПС на стадии тестирования.

В этом случае процесс переходит в проектную процедуру выбора стратегии модификации, в которой руководителем и аналитиком может быть выбрана одна из альтернатив:

- выбор нового алгоритма решения задачи;
- модификация используемого алгоритма.

Необходимость выбора нового алгоритма может возникнуть в случае получения низкой эффективности проектируемой ПС. В то же время часто для получения эффективного программного кода необходимо осуществить модификацию алгоритма. Например, для этих целей могут быть использованы перестановочные модификации, которые являются мощным средством повышения эффективности проектируемых ПС. Данные модификации не меняют общего алгоритма обработки данных, однако меняют порядок этой обработки. При этом необходимо отметить, что перестановочные модификации должны применяться таким образом, чтобы не нарушать информационных связей в алгоритме. Перестановочные модификации могут улучшить параметры эффективности алгоритма в следующих направлениях:

- эффективность использования подсистемы памяти;
- распараллеливание вычислений;
- векторизация вычислений;
- балансировка нагрузки в гетерогенных ВС.

На рисунке 3.7 не приведены очевидные действия, выполняемые в процессе разработки ПС, такие как предварительный статический анализ

исходного кода, компиляция кода, поиск и устранения ошибок, т.к. данные действия являются обязательными и напрямую не оказывают влияния на итоговую эффективность проектируемой ПС.

Недостатком традиционного механизма реализации этапа разработки ПС является отсутствие формального подхода к выполнению описанных выше модификаций. Разработчики часто используют эвристические методы, эффективность которых в большой степени зависит от квалификации разработчика. При выявлении неэффективного использования различных подсистем ВС в подавляющем большинстве случаев у разработчика отсутствует понятный алгоритм решения задачи (формальный подход) и повышение эффективности ПС будет вестись за счет квалификации разработчика, который опирается на собственный опыт и коллективный опыт сообщества разработчиков (эвристика). При этом есть общие рекомендации к разработке эффективных ПС.

Например, эвристики могут быть описаны следующим образом. Для эффективного использования кеш-памяти рекомендуется:

- обрабатывать массивы в том порядке, в котором они хранятся в памяти;
- разделять часто используемые («горячие») и редко используемые («холодные») данные;
- разрабатывать блочные алгоритмы решения задач.

Следует учитывать, что обработка протяженных блоков данных (массивов) с точки зрения использования кеш-памяти является более предпочтительным, чем обработка одиночно случайно расположенных данных.

Если учесть первые два требования в принципе не сложно, то разработка блочных алгоритмов решения задач и в целом оценка эффективности использования кеш-памяти является задачей достаточно сложной и неформализованной. Кроме того, блочные алгоритмы часто являются неэффективными при обработке матриц специального вида, для

которых возникает необходимость разработки специальных алгоритмов обработки

Эффективная векторизация вычислений также является достаточно сложной и трудно формализуемой задачей. Используются два основных подхода:

- использование оптимизирующего компилятора с блоком автовекторизации;
- выполнение векторизации кода «вручную», что требует от специалиста специальных знаний и компетенций для того, чтобы получить более эффективный результат, чем результат автоматической векторизации.

К эффективным методам повышения эффективности ПС относятся также распараллеливание вычислений и балансировка нагрузки в ГВС, использование ускорителей. Данные подходы были подробно проанализированы в главе 1.

3.2. Оптимизация кода.

При традиционном подходе возможны два варианта оптимизации кода:

- использование эвристики;
- использование оптимизирующего компилятора.

Разработчики часто используют эвристический подход с использованием рекомендаций разработчиков компиляторов, членов IT-сообщества, личного опыта. Однако такой подход не гарантирует необходимого результата.

Использование оптимизирующего компилятора также требует достаточно высокой квалификации разработчика, т. к. современные компиляторы являются сложными системами, для овладения которыми необходимы время и опыт. Не оптимально написанный код может привести к получению крайне неэффективной программы даже с использованием самых современных оптимизирующих компиляторов. Необходимо отметить, что современные оптимизирующие компиляторы генерируют подробные отчеты, которые также можно считать элементами статического анализа.

3.3. Динамический анализ.

Анализ используется в тот момент, когда программа уже написана и отлажена в том смысле, что обнаружены и исправлены все найденные ошибки. Однако, качество полученной на данном шаге программы, как уже было сказано ранее, в значительной степени зависит от квалификации разработчика. Негативное влияние на качество разрабатываемых программ оказывают различные факторы. В качестве примера можно привести увеличение сложности решаемых задач, обилие современных вычислительных архитектур и другие факторы.

На данном шаге разработчик осуществляет динамический анализ готового приложения. При этом оценивается эффективность использования различных подсистем компьютера, скорость работы и т. д. Анализ готовых приложений осуществляется инструментальными средствами, такими как, например, Intel VTune Amplifier [65]. Данные приложения являются мощными инструментами анализа приложений, однако они не могут быть применены на начальных этапах разработки программ [100].

После проведения анализа может быть принято решение о модификации базового алгоритма решения задачи и/или дополнительной оптимизации кода с дальнейшим проведением повторного анализа.

В случае получения неудовлетворительного результата может быть принято решение о выполнении фактически нового цикла разработки сначала, с выбором или разработкой нового базового алгоритма решения задачи.

4. Тестирование ПС.

В заключительной проектной процедуре перед переходом ПС на этап эксплуатации, осуществляется тестирование. В процессе тестирования выполняется ряд тестов, цель которых заключается в проверке реальных и ожидаемых результатов работы ПС. По результатам тестирования также может быть принято решение о доработке ПС, как и по результатам динамического анализа на предыдущем шаге.

3.1.4 Оценка времени выполнения проектных процедур разработки ПС при традиционном подходе

Время, затрачиваемое на этапе разработки ПС, вычисляется следующим образом.

$$T = T_1 + \sum_{i=1}^n \left(a_i * T_2 + \sum_{j=1}^{m_i} [b_{i,j} * T_2 + (T_3 + T_4 + T_5)] + T_6 \right), \quad (3.11)$$

$$a_i \in \{0, 1\}, b_{i,j} \in \{0, 1\}, a_1 = 1,$$

$$b_{i,1} = 0, \text{ если } a_i = 1$$

где T – общее время выполнения этапа разработки ПС;

T_1 – время, затрачиваемое на уточнение требований;

T_2 – время, затрачиваемое на выбор / разработку алгоритма;

T_3 – время, затрачиваемое на модификацию алгоритма;

T_4 – время, затрачиваемое на выполнение оптимизации;

T_5 – время, затрачиваемое на выполнение анализа полученного ПС;

T_6 – время, затрачиваемое на тестирование ПС;

n – число итераций цикла разработки, включая шаг тестирования;

m_i – число итераций внутреннего цикла разработки (не включая шаг тестирования, возврат после анализа), на i -ой итерации, содержащей тестирование;

$a_i, b_{i,j}$ – коэффициенты, учитывающие выполнение шага выбора / разработки алгоритма на текущей итерации.

В качестве примера рассмотрим ситуацию, когда после анализа и тестирования осуществляется только модификация алгоритма

$$T = T_1 + T_2 + \sum_{i=1}^n \left(\sum_{j=1}^{m_i} (T_3 + T_4 + T_5) + T_6 \right) \quad (3.12)$$

Вариант, где после тестирования осуществляется смена базового алгоритма, после анализа — модификация

$$T = T_1 + \sum_{i=1}^n \left(T_2 + \sum_{j=1}^m (T_3 + T_4 + T_5) + T_6 \right) \quad (3.13)$$

Возможны и другие варианты. Например, вариант без обратных связей

$$T = T_1 + T_2 + T_3 + T_4 + T_5 + T_6 \quad (3.14)$$

Все эти варианты объединяет одно — слагаемое $(T_3 + T_4 + T_5)$. Во всех вариантах модификация алгоритма, оптимизация и динамический анализ программы идут друг за другом. Такой подход не всегда может быть эффективным:

- анализ готового приложения предполагает предварительное написание кода, выполнение процедур, связанных с оптимизацией, поиском ошибок и т. д.; если в результате анализа выяснится неэффективность полученной программы, данные действия были выполнены напрасно, и разработчик должен будет выполнить их для нового или модифицированного алгоритма на новой итерации разработки ПС;

- использование эвристического подхода к модификации алгоритма не гарантирует результата; даже выполнение нескольких итераций улучшения может не привести к получению эффективной ПС; более того, данный подход не гарантирует непрерывного улучшения ПС, в результате на следующей итерации может быть получена ПС с характеристиками, худшими, чем на предыдущей итерации.

В процессе анализа традиционного подхода к разработке ПС были выявлены следующие недостатки:

- преобладание эвристического подхода на стадии модификации алгоритма;
- частично эвристический подход на стадии оптимизации кода;
- выполнение динамического анализа уже готового приложения, которое написано, оптимизировано и отлажено разработчиком.

Данный подход не позволяет выявить неэффективность используемых алгоритмов и подходов к оптимизации кода на начальных стадиях этапа разработки ПС, что позволяет выполнять итерации, приводящие к ухудшению характеристик ПС.

3.1.5 Предлагаемый подход к проектированию ПС

Решением данной проблемы может стать перенос шага анализа на более ранние стадии этапа разработки ПС, в результате чего выявление «неудачных» итераций и отказ от их выполнения позволит снизить общее количество итераций проектных процедур на данном этапе ЖЦ ПС. Для этого предлагается модель улучшения показателей эффективности ПС (рисунок 3.8), в которой:

- шаги модификации алгоритма и оптимизации проектируемой ПС совмещены, данный шаг реализуется в соответствии с формальными подходами и методами, которые содержат алгоритмы поиска паттернов доступа к данным, снижающих эффективность использования различных подсистем ВС, например, эффективность использования кеш-памяти, эффективность векторизации и других подсистем. Данные алгоритмы могут быть разработаны с использованием методов машинного обучения, интерпретирующего профилирования, кеш-симуляции и иных подходов. Кроме того, данные подходы и методы могут быть реализованы в виде системы поддержки ЖЦ программных систем, позволяющей автоматизировать анализ эффективности возможных проектных решений о стратегиях разработки и модификации ПС;

- формальные подходы и методы гарантируют получение характеристик эффективности ПС, не хуже, чем на предыдущей итерации за счет отказа от выполнения «неудачных» итераций.

- до тестирования готового приложения выполняется статический анализ полученного кода; анализ кода выполняется с использованием

аналитического подхода, позволяющего оценить потенциальную эффективность программного кода или с помощью специально разработанных инструментальных средств.

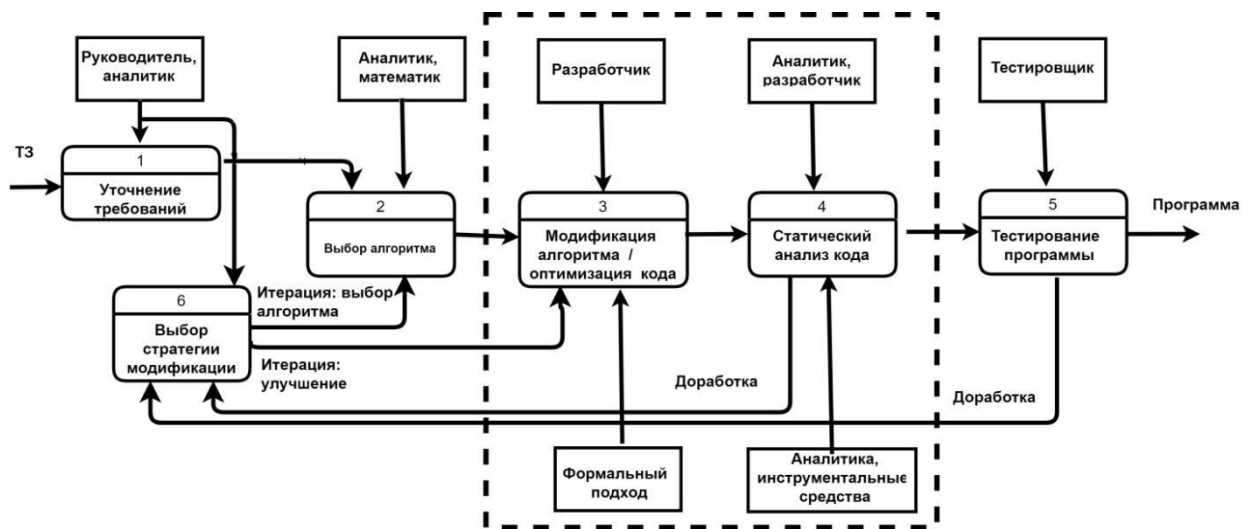


Рисунок 3.8 – Диаграмма этапа разработки ПС в рамках модели улучшения показателей эффективности ПС. Предлагаемый подход (ТО-ВЕ)

3.1.' Модификация / оптимизация.

Реализация формальных подходов и методов позволит совместить возможную модификацию алгоритма и оптимизацию кода, при этом получаемый код будет наилучшим образом соответствовать выбранной архитектуре ГВС.

3.2.' Статический анализ

В ходе проведения статического анализа осуществляется оценка целевых показателей эффективности полученного кода программы с учетом характеристик выбранной архитектуры ВС.

Проведение анализа может выявить неэффективное использование подсистем ВС, что может привести к необходимости выбора стратегии модификации, что в конечном итоге приведет к модификации алгоритма и оптимизации кода или выбору нового алгоритма решения задачи.

Следует отметить, что использование статического анализа не исключает проведение анализа готового приложения с использованием инструментальных средств, однако проведение качественного статического

анализа приводит к тому, что дальнейший анализ становится ненужным. Суть предлагаемого подхода заключается в смещении акцента с динамического анализа на статический анализ исходного кода с целью повышения эффективности ПС. Однако, возможно применение методов и инструментов динамического анализа на завершающих итерациях, когда приложение фактически спроектировано для проверки эффективности принятых проектных решений, в отличие от традиционного подхода, где выполнение такого анализа предполагается на каждой итерации.

3.1.6 Оценка времени выполнения проектных процедур разработки ПС при предлагаемом подходе

В этом случае даже без обратных связей этап разработки приложения будет характеризоваться следующим выражением

$$T = T_1 + T_2 + T'_3 + T'_4 + T_6 \quad (3.15)$$

где T'_3 – время, затрачиваемое на модификацию алгоритма и оптимизацию кода;

T'_4 – время, затрачиваемое на выполнение статического анализа кода

При этом модификация алгоритма, оптимизация кода и выполнение статического анализа осуществляется с использованием формального подхода, что позволяет принимать проектные решения по повышению эффективности проектируемых ПС.

В общем случае время, затрачиваемое на этапе разработки ПС при применении предложенной модели проектирования, можно представить следующим образом.

$$T = T_1 + \sum_{i=1}^{n'} \left(a'_i * T_2 + \sum_{j=1}^{m'_i} [b'_{i,j} * T_2 + (T'_3 + T'_4)] + T_6 \right),$$

$$a'_i \in \{0, 1\}, b'_{i,j} \in \{0, 1\}, a'_1 = 1, \quad (3.16)$$

$$b'_{i,1} = 0, \text{ если } a'_i = 1$$

где n' – число итераций цикла разработки, включая шаг тестирования;

m'_i – число итераций внутреннего цикла разработки (не включая шаг тестирования, возврат после анализа), на i -ой итерации, содержащей тестирование;

$a'_i, b'_{i,j}$ – коэффициенты, учитывающие выполнение шага выбора / разработки алгоритма на текущей итерации

В качестве примера рассмотрим те же ситуации, что и в случае традиционного подхода. Ситуация, когда после анализа и тестирование осуществляется только модификация алгоритма.

$$T = T_1 + T_2 + \sum_{i=1}^{n'} \left(\sum_{j=1}^{m'_i} (T'_3 + T'_4) + T_6 \right) \quad (3.17)$$

Вариант, где после тестирования осуществляется смена базового алгоритма, после анализа — модификация.

$$T = T_1 + \sum_{i=1}^{n'} \left(T_2 + \sum_{j=1}^{m'_i} (T'_3 + T'_4) + T_6 \right) \quad (3.18)$$

Предлагаемый подход позволяет снизить количество итераций в процессе проектирования и уменьшить время, затрачиваемое на выполнение итераций. В самом деле, реализация этапа разработки ПС, предлагаемая в рамках модели улучшения показателей эффективности ПС, предполагает выполнение только «удачных» итераций, выполнение которых не ведет к ухудшению показателей эффективности ПС. В рамках традиционного подхода к разработке ПС могут выполняться любые итерации, в т.ч. и те, которые приводят к ухудшению показателей эффективности проектируемой ПС. Соответственно, количество итераций, выполняемых на этапе разработки ПС в рамках модели улучшения показателей эффективности ПС, не может превышать количество итераций, выполняемых в рамках традиционного подхода (3.19).

$$n' \leq n, m' \leq m \quad (3.19)$$

Кроме того, время, затрачиваемое на выполнение итерации разработки ПС в рамках предлагаемого подхода, не превышает времени, затрачиваемого на выполнение итерации в рамках традиционного подхода (3.20).

$$T'_3 + T'_4 < T_3 + T_4 + T_5 \quad (3.20)$$

Справедливость неравенства (3.20) доказывается следующими утверждениями:

- предлагаемая модель улучшения показателей эффективности ПС позволяет выполнять модификацию алгоритма и оптимизацию кода, основываясь на формальных подходах, в отличие от традиционного подхода, предполагающего использование эвристики на данном этапе;

- статический анализ, проводимый на основе новых предлагаемых методов, позволяет выявить неэффективное использование вычислительных ресурсов на основе анализа исходного кода программы, в отличие от традиционного подхода, предполагающего выполнение компиляции, отладки и динамического анализа готового приложения на каждой итерации разработки ПС.

С учетом (3.19) и (3.20) предлагаемый подход позволяет повысить эффективность проектируемых ПС на этапе разработки.

3.1.7 Сравнение традиционного и предлагаемого подходов к реализации проектных процедур разработки ПС

В таблице 3.1 приведено сравнение двух подходов к реализации этапа разработки ПС с использованием нескольких показателей:

- анализ приложения: возможность переноса анализа эффективности ПС на начало этапа разработки;
- выполнение итераций: возможность исключения неудачных итераций из процесса проектирования ПС;

- эффективность ПС на следующей итерации: возможность непрерывного улучшения показателей эффективности ПС;
- преобладающие подходы к модификации алгоритма и оптимизации кода: возможность реализации формальных подходов с исключением или значительным уменьшением доли эвристики;
- возможность совмещения проектных процедур модификации алгоритма и оптимизации кода.

Таблица 3.1 Этап разработки ПС. Сравнение подходов AS_IS и TO_BE.

Показатель	AS_IS	TO_BE
Анализ приложения	Динамический анализ готового приложения, завершающие проектные процедуры	Статический анализ исходного кода, начальные проектные процедуры
Выполняемые итерации	Любые	Не ухудшающие показатели эффективности проектируемой ПС
Эффективность ПС на следующей итерации	Может ухудшиться	Не ухудшается (непрерывное улучшение)
Преобладающие подходы к модификации алгоритма и оптимизации кода	Эвристика	Формальные подходы

Продолжение таблицы 3.1

Показатель	AS_IS	TO_BE
Модификация алгоритма	Эвристика	Совмещенная проектная процедура, выполняемая с использованием формального подхода
Оптимизация кода	Компилятор, эвристика	

В таблице 3.2 приведено сравнение времени реализации указанных подходов на основе ранее проведенного анализа.

Таблица 3.2. Сравнение времени реализации этапа разработки ПС с использованием традиционного и предлагаемого подходов

Традиционный подход
$T = T_1 + \sum_{i=1}^n \left(a_i * T_2 + \sum_{j=1}^{m_i} [b_{i,j} * T_2 + (T_3 + T_4 + T_5)] + T_6 \right),$ $a_i \in \{0, 1\}, b_{i,j} \in \{0, 1\}, a_1 = 1,$ $b_{i,1} = 0, \text{ если } a_i = 1$
Предлагаемый подход
$T = T_1 + \sum_{i=1}^{n'} \left(a'_i * T_2 + \sum_{j=1}^{m'_i} [b'_{i,j} * T_2 + (T'_3 + T'_4)] + T_6 \right),$ $a'_i \in \{0, 1\}, b'_{i,j} \in \{0, 1\}, a'_1 = 1,$ $b'_{i,1} = 0, \text{ если } a'_i = 1$
Ограничения
$T'_3 + T'_4 < T_3 + T_4 + T_5$ $n' \leq n, m' \leq m$

Предлагаемый подход позволяет повысить эффективность ПС на этапе разработки за счет уменьшения общего времени реализации этапа. Это становится возможным за счет непрерывного улучшения показателей эффективности в результате применения формальных подходов к модификации алгоритма и оптимизации кода, что в целом приводит к сохранению или уменьшению количества итераций разработки.

3.1.8 Этап модификации ЖЦ ПС

Необходимость в модификации ПС может возникнуть на этапе эксплуатации по следующим причинам:

- выявлены ошибки ПС;
- выявлена недостаточная эффективность ПС на ГВС с целевой архитектурой;
- возникла необходимость разработки эффективной версии ПС для ГВС с другой архитектурой.

Первые две причины связаны с невысокими эксплуатационными характеристиками или некорректной работой ПС, возникновением ошибок. В этом случае ПС переходит на этап на модификации, для устранения обнаруженных ошибок или повышения эффективности.

Последняя причина модификации связана с использованием в качестве целевых архитектур ВС тех архитектур, для которых ранее не осуществлялась оптимизация и в целом разработка эффективной версии ПС. здесь возможны два варианта:

- использование принципиально новой архитектуры; например, добавление в целевую архитектуру FPGA-ускорителей; если ранее код для этих ускорителей не разрабатывался, то они не могут быть использованы в процессе вычислений; необходима «глубокая» доработка кода, использование дополнительных технологий программирования, библиотек и т.д., разработка новых программных модулей;

- изменение характеристик целевой ГВС, при этом целевая ГВС принадлежит к классу систем, для которых уже осуществлялась оптимизация ПС; например, ранее уже была разработана эффективная версия ПС для конкретной ГВС с общей памятью; новая ГВС принадлежит к тому же классу, однако, имеет другие характеристики (размер кеш-памяти, векторное расширение системы команд); кроме того, в ВПВ часто используется оптимизация для конкретного поколения МП.

Таким образом, можно выделить следующие варианты модификации ПС:

- устранение ошибок;
- повышение эффективности ПС на целевой ВС;
- оптимизация для использования новой архитектуры ВС;
- оптимизация для использования конкретного поколения МП в рамках известной архитектуры ВС;
- оптимизация для использования в рамках известной архитектуры ВС с новыми параметрами.

На рисунке 3.2 можно выделить те же основные проектные процедуры, что и на этапе разработки ПС (рисунок 3.1), рассмотрим их подробнее.

1. Уточнение требований.

Данный шаг содержит уточнение тех же требований, что и на этапе разработки ПС с некоторыми изменениями:

- в случае устранения ошибок и / или повышения эффективности на целевой ГВС требования к проектируемому ПС не изменяются, поэтому данный шаг пропускается;
- в случае изменения целевой архитектуры ГВС или иных требований (используемая операционная система, инструментальные средства и т.д.) уточнение требований осуществляется аналогично шагу на этапе разработки ПС.

2. Выбор или разработка алгоритма решения задачи.

Данный шаг может быть использован во всех вариантах выполнения этапа модификации ПС:

- исправление ошибок может привести к изменению базового алгоритма решения задачи;
- оптимизация для использования новой архитектуры ГВС может привести к изменению базового алгоритма решения задачи;
- и т.д.

Однако в ряде случаев данный шаг может не использоваться:

- оптимизация для использования конкретного поколения МП в рамках известной архитектуры ГВС не всегда требует изменения базового алгоритма решения задачи;
- оптимизация для использования в рамках известной архитектуры ГВС с новыми параметрами не всегда требует изменения базового алгоритма решения задачи.

3. Процесс кодирования и анализа полученной программы.

Следующая проектная процедура содержит процесс разработки кода программы. Она представлена в трех вариантах:

- традиционный вариант (рисунок 3.9);
- предлагаемый новый вариант реализации проектной процедуры, требующий модификации и оптимизации кода программы (рисунок 3.10);
- предлагаемый новый вариант реализации проектной процедуры, представляющий собой оптимизацию программы для использования в рамках известной архитектуры ВС с новыми параметрами (рисунок 3.11).

Как видно из рисунков 3.9 и 3.10, проектные процедуры этапа модификации ПС для традиционного подхода и предлагаемого подхода, требующего модификации и оптимизации кода программы, практически полностью совпадают с соответствующими проектными процедурами этапа разработки ПС.

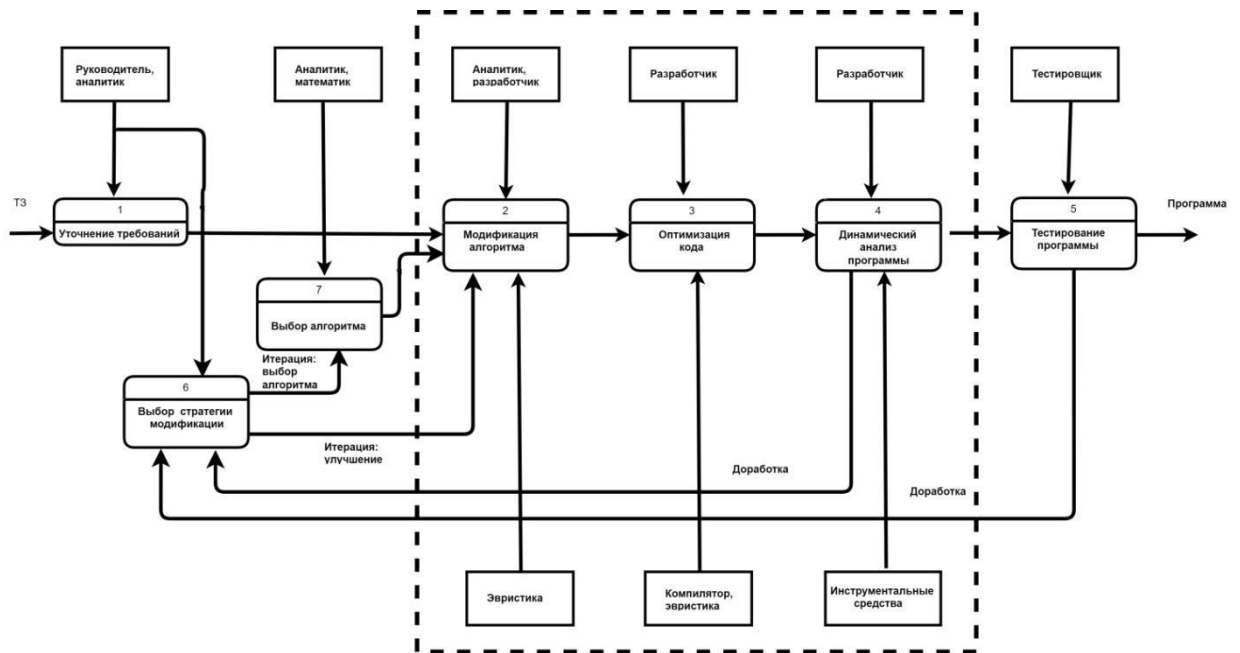


Рисунок 3.9 — Диаграмма этапа модификации ПС. Традиционный подход (AS-IS)

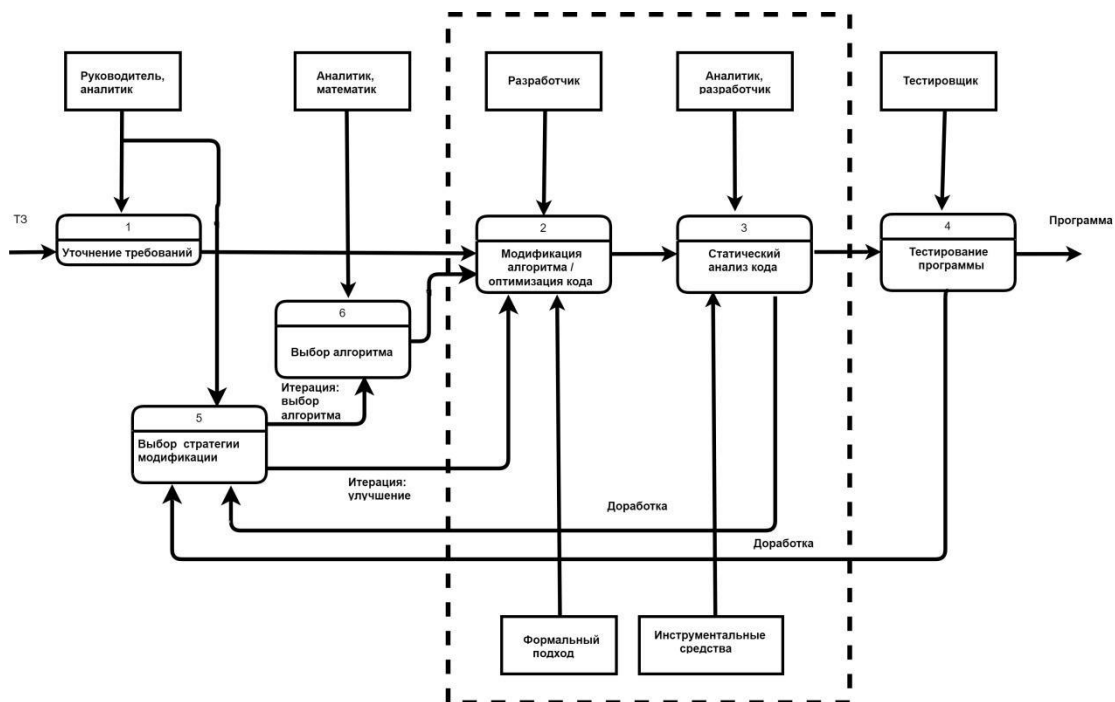


Рисунок 3.10 — Диаграмма этапа модификации ПС в рамках модели улучшения показателей эффективности ПС. Предлагаемый общий подход (TO-BE)

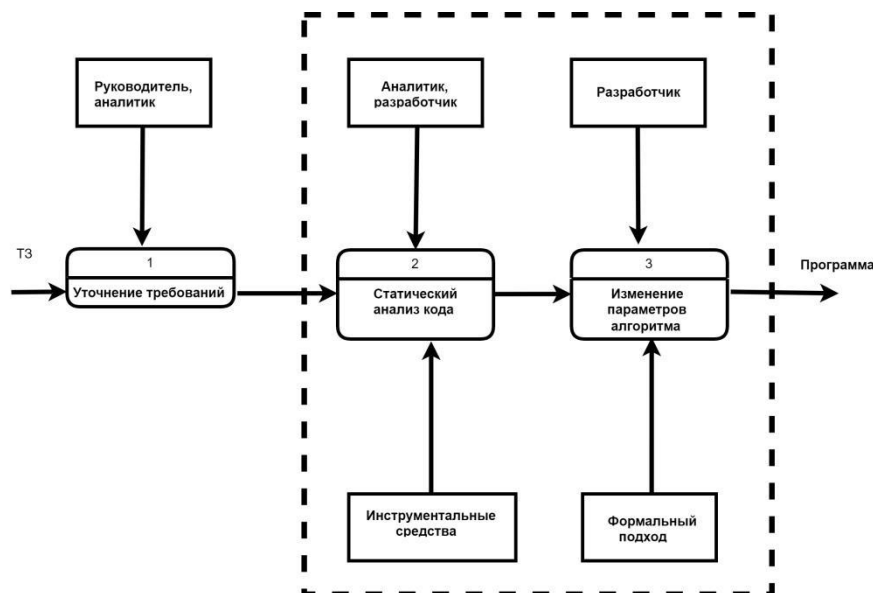


Рисунок 3.11 — Диаграмма этапа модификации ПС в рамках модели улучшения показателей эффективности ПС. Предлагаемый подход, параметризованные алгоритмы (ТО-ВЕ)

Аналогично этапу разработки ПС, основные отличия заключаются в следующем:

- традиционный подход предполагает предварительную разработку кода программы, оптимизацию, отладку, после чего осуществляется анализ полученного ПС с использованием инструментальных средств, позволяющих в частности выявить неэффективное использование различных подсистем ВС;

- предлагаемый подход предполагает перенос шага анализа ближе к началу этапа модификации ПС, в частности, предлагается проводить статический анализ кода программы после шага модификации/оптимизации; анализ осуществляется с использованием аналитических подходов либо с использованием новых инструментальных средств; использование статического анализа позволяет отказаться от шага анализа готового приложения с использованием инструментальных средств.

Основным отличием данных проектных процедур модификации ПС от соответствующих процедур разработки ПС, рассмотренных ранее, является то, что на первой итерации не предполагается выбор/разработка базового

алгоритма. Предполагается, что для модификации ПС достаточно будет использовать алгоритм, выбранный на этапе разработки. Смена базового алгоритма возможна на следующих итерациях модификации ПС.

По аналогии с этапом разработки был сделан вывод о том, что предлагаемый подход позволяет снизить количество итераций проектных процедур в процессе модификации и уменьшить время, затрачиваемое на выполнение итераций, т.е. повысить эффективность ПС на данном этапе ЖЦ.

На рисунке 3.11 представлена диаграмма этапа модификации ПС в рамках модели улучшения показателей эффективности ПС, представляющего собой оптимизацию ПС для использования в рамках известной архитектуры ГВС с новыми параметрами. Она содержит следующие проектные процедуры:

- уточнение требований;
- модификация программы.

В рамках последней процедуры выполняется статический анализ исходного кода и изменение параметров алгоритма, причем данный вариант ее реализации не был доступен на этапе разработки ПС, т. к. на данном этапе отсутствовал программный продукт, который можно было модифицировать.

Уточнение требований в данном случае заключается в получении характеристик ГВС, для которой выполняется модификация ПС. При этом необходимо, чтобы ранее уже был создан вариант ПС для ГВС с подобной архитектурой. Выполняется руководителем или аналитиком.

Т.к. код программы для ГВС с требуемой архитектурой уже был получен ранее, в данном случае можно сразу приступить к статическому анализу с учетом характеристик системы, которые были уточнены на предыдущем шаге. Статический анализ проводится аналитиком или разработчиком с использованием аналитических подходов или новых инструментальных средств.

После проведения статического анализа необходимо подобрать параметры алгоритма с учетом характеристик целевой архитектуры ГВС,

подбор параметров осуществляется разработчиком на основе формального подхода. Подобный подход (рисунок 3.11) может быть использован на этапе модификации ПС для реализации параметризованных алгоритмов [13], т.к.:

- параметризованные алгоритмы должны проектироваться таким образом, чтобы изменение параметров не влияло на логику работы ПС, т.е. ПС должна выдавать одни и те же результаты при задании различных параметров на одинаковых наборах входных данных;;
- различные наборы параметров должны влиять только на эффективность получаемой ПС в целевой ГВС;
- рассматривается этап модификации ПС, т.е. этап разработки жизненного цикла ПС уже был выполнен, соответственно полученная ПС уже была отлажена и протестирована;
- причина модификации ПС не связана с ошибками, обнаруженными в процессе эксплуатации программы.
- проведение статического анализа позволяет выявить эффективность используемых паттернов доступа к данным и выбрать значения параметров, максимизирующих показатели эффективности ПС.

В предлагаемом подходе параметризация осуществляется на этапе статического анализа, что повышает эффективность ПС, т.к. отпадает необходимость в получении различных вариантов исполняемого модуля, анализе их производительности, выборе лучшего варианта [75]. Более того, параметров алгоритма в классическом понимании смысла этого слова может и не быть совсем. Статический анализатор подбирает оптимальные параметры циклов и других паттернов доступа к данным автоматически, либо это делает специалист с использованием соответствующего аналитического аппарата. Стоит отметить, что в данном случае специалист также должен обладать достаточно высокой квалификацией.

Кроме того предлагаемый подход полностью избавляет от необходимости получать различные варианты исполняемого модуля, анализа их производительности, выбора лучшего варианта.

3.1.9 Оценка времени, затрачиваемого на этап модификации ПС

Рассмотрим оценку времени модификации ПС с использованием традиционного варианта, вариант, где после анализа и тестирования осуществляется только модификация алгоритма

$$T = T_1 + \sum_{i=1}^n \left(a_i * T_2 + \sum_{j=1}^{m_i} [b_{i,j} * T_2 + (T_3 + T_4 + T_5)] + T_6 \right), \quad (3.21)$$

$$a_i \in \{0, 1\}, b_{i,j} \in \{0, 1\}, a_1 = 0, b_{1,1} = 0$$

$$b_{i,1} = 0, \text{ если } a_i = 1$$

Переменные в (3.21) имеют тот же смысл, что и в (3.11). Отличие выражений заключается в отсутствии шага выбора / разработки алгоритма на первой итерации модификации программы.

Рассмотрим несколько вариантов.

Вариант, где после тестирования осуществляется смена базового алгоритма, после анализа — модификация

$$T = T_1 + T_3 + T_4 + T_5 + \sum_{i=1}^n \left(T_2 + \sum_{j=1}^{m_i} (T_3 + T_4 + T_5) + T_6 \right) \quad (3.22)$$

Возможны и другие варианты. Например, вариант без обратных связей

$$T = T_1 + T_3 + T_4 + T_5 + T_6 \quad (3.23)$$

Здесь также во всех вариантах, как и в случае этапа разработки ПС, используется слагаемое $(T_3 + T_4 + T_5)$. Во всех вариантах модификация алгоритма, оптимизация и анализ программы идут друг за другом. Такой подход также не всегда может быть эффективным по причинам, описанным ранее.

Предлагается два новых варианта реализации этапа модификации ПС:

- модификация ПС с использованием предлагаемого подхода, требующего модификации и оптимизации кода программы;
- модификация ПС с использованием предлагаемого подхода, представляющего собой оптимизацию программы для использования в рамках известной архитектуры ВС с новыми параметрами.

Для первого случая время выполнения этапа модификации будет определяться следующим выражением (3.24).

$$T = T_1 + \sum_{i=1}^{n'} \left(a'_i * T_2 + \sum_{j=1}^{m'_i} [b'_{i,j} * T_2 + (T'_3 + T'_4)] + T_6 \right), \quad (3.24)$$

$$a'_i \in \{0, 1\}, b'_{i,j} \in \{0, 1\}, a_1 = 0, b'_{1,1} = 0,$$

$$b'_{i,1} = 0, \text{ если } a'_i = 1$$

Вариант, где после тестирования осуществляется смена базового алгоритма, после анализа — модификация

$$T = T_1 + T'_3 + T'_4 + \sum_{i=1}^{n'} \left(T_2 + \sum_{j=1}^{m'_i} (T'_3 + T'_4) + T_6 \right) \quad (3.25)$$

Вариант без обратных связей

$$T = T_1 + T'_3 + T'_4 + T_6 \quad (3.26)$$

Предлагаемый подход к реализации этапа модификации ПС оказывается эффективнее традиционного подхода по причинам, которые были описаны для этапа разработки ПС:

- количество итераций при реализации нового подхода не превышает количество итераций при реализации традиционного подхода;
- время на выполнение одной итерации при реализации нового подхода не превышает времени на выполнение одной итерации при реализации традиционного подхода.

Еще большую эффективность ПС на этапе модификации можно получить в случае оптимизации программы для использования в рамках

известной архитектуры ГВС с новыми параметрами. В этом случае время на модификацию программы составит (3.27).

$$T = T_1 + T_3'' + T_4'' \quad (3.27)$$

где T_3'' – время на статический анализ кода;

T_4'' – время на изменение параметров алгоритма.

Стоит отметить, что данный вариант реализации этапа модификации ПС является однократным, реализуется за одну итерацию.

3.1.10 Сравнение традиционного и предлагаемого подходов к реализации проектных процедур модификации ПС

Таким образом, использование предложенной модели улучшения показателей эффективности ПС, представляющей собой модель итеративного процесса проектирования ПС, приводит к уменьшению времени, затрачиваемого на этапах разработки и модификации ПС, что повышает эффективность ПС на данных этапах ЖЦ. Более наглядно сравнение традиционного и предлагаемого подходов к модификации ПС представлено в таблицах 3.3 и 3.4.

Таблица 3.3. Сравнение времени реализации этапа модификации ПС с использованием традиционного и предлагаемых подходов

Традиционный подход
$T = T_1 + \sum_{i=1}^n \left(a_i * T_2 + \sum_{j=1}^{m_i} [b_{i,j} * T_2 + (T_3 + T_4 + T_5)] + T_6 \right),$ $a_i \in \{0, 1\}, b_{i,j} \in \{0, 1\}, a_1 = 0, b_{1,1} = 0$ $b_{i,1} = 0, \text{ если } a_i = 1$

Продолжение таблицы 3.3.

Предлагаемый подход
$T = T_1 + \sum_{i=1}^{n'} \left(a'_i * T_2 + \sum_{j=1}^{m'_i} [b'_{i,j} * T_2 + (T'_3 + T'_4)] + T_6 \right),$ $a'_i \in \{0, 1\}, b'_{i,j} \in \{0, 1\}, a_1 = 0, b'_{1,1} = 0,$ $b'_{i,1} = 0, \text{ если } a'_i = 1$
Ограничения
$T'_3 + T'_4 < T_3 + T_4 + T_5$ $n' \leq n, m' \leq m$

Таблица 3.4. Время реализации этапа модификации ПС с использованием предлагаемого подхода для параметризованных алгоритмов

Параметризованные алгоритмы
$T = T_1 + T''_3 + T''_4$
Ограничения
$T''_3 + T''_4 < T_3 + T_4 + T_5$

Аналогично этапу разработки, повышение эффективности этапа модификации ПС становится возможным за счет непрерывного улучшения показателей эффективности в результате применения формальных подходов к модификации алгоритма и оптимизации кода, что в целом приводит к сохранению или уменьшению количества итераций проектных процедур и сокращению времени реализации отдельных итераций.

3.2 Математическая модель оценки эффективности проектируемых ПС

Для количественной оценки степени эффективности ПС было предложено использовать меру аппаратурно-временной сложности задачи

(3.7). Оценить эффективность ПС на этапе эксплуатации в соответствии с (3.7) можно только путем реального использования готовой к применению ПС с оценкой мер аппаратурной и временной сложности ПС, т.е. с замером времени решения задачи, оценкой используемых аппаратных ресурсов.

Для оценки эффективности ПС анализ осуществляется на этапе эксплуатации, где по результатам выполнения ПС (решения задач, для которых была спроектирована ПС) осуществляется анализ эффективности, включающий в себя:

- получение характеристик ПС;
- анализ характеристик ПС.

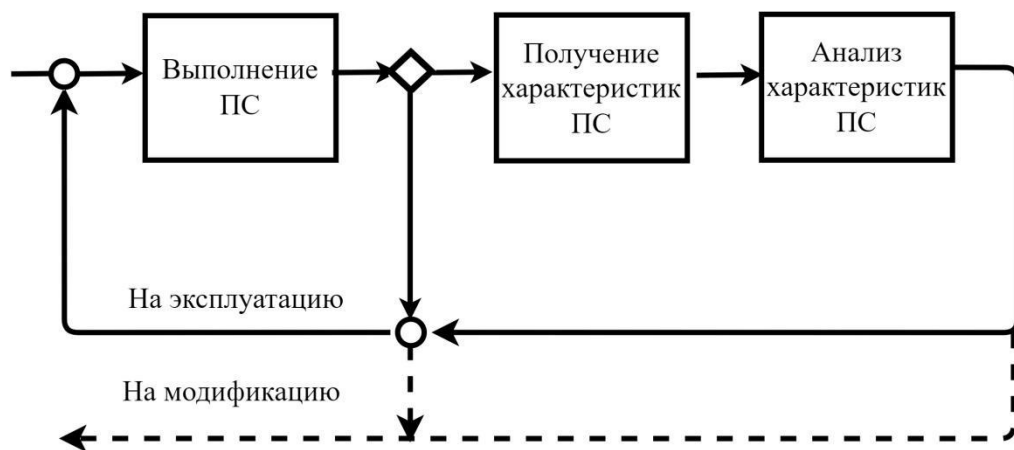


Рисунок 3.12 – Этап эксплуатации ПС

Данный подход не соответствует модели улучшения показателей эффективности ПС, т.к. предполагает оценку эффективности уже готовой и отлаженной ПС, в то время как в рамках предлагаемого подхода этап анализа переносится с динамического анализа готового приложения на статический анализ исходного кода.

Ранее были рассмотрены существующие подходы и инструментальные средства, использующиеся для проведения статического анализа, выявлена невозможность применения данных подходов и средств для повышения эффективности ПС в соответствии с (3.7). Для проведения подобного статического анализа необходимо выработать соответствующие научные основы, заключающиеся в моделях и методах оценки эффективности

проектируемых ПС, разработке на базе данных моделей и методов инструментальных средств, предназначенных для поддержки ЖЦ ПС в рамках предложенной модели улучшения показателей эффективности ПС на этапах разработки и модификации ЖЦ ПС.

Для принятия проектных решений по повышению эффективности ПС в рамках модели улучшения показателей эффективности была предложена математическая модель оценки эффективности проектируемых ПС в ГВС с различными характеристиками. В рамках данной модели предложены метрики, позволяющие осуществлять оценку эффективности ПС на различных этапах ЖЦ.

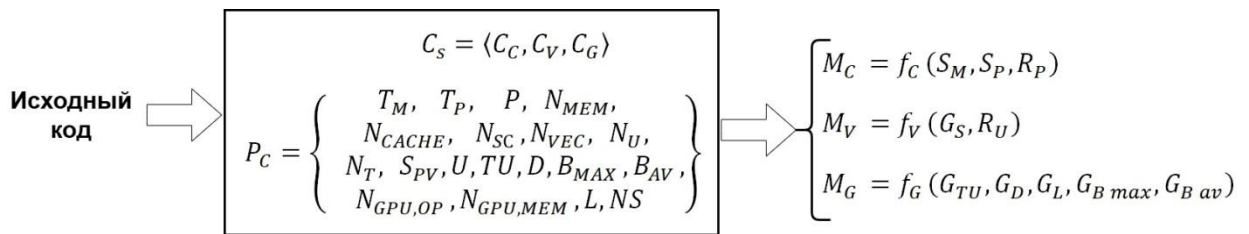


Рисунок 3.13 - Математическая модель оценки эффективности проектируемых ПС,

На рисунке 3.13 используются следующие обозначения:

C_s – характеристики ГВС, в т.ч.:

C_C – характеристики подсистемы памяти.

C_V – характеристики подсистемы векторизации вычислений,

C_G – характеристики GPU;

P_C – , множество определяемых параметров ПС, в т.ч.:

T_M – время работы с памятью,

T_P – время выполнения программы,

P – усредненное значение вероятности кеш-промаха,

N_{MEM} - количество обращений к основной памяти,

N_{CACHE} - количество обращений к кеш-памяти,

N_{SC} – необходимое число скалярных операций,

N_{VEC} – число операций с учетом векторизации,

N_U – число невыровненных векторных обращений к памяти,

N_T – общее число векторных обращений к памяти,
 S_{PV} – потенциальное ускорение,
 U – доля невыровненных операций,
 TU – доля использованных потоков,
 D – средняя плотность вычислений,
 B_{max} – максимальный уровень несбалансированности потоков,
 B_{av} – средний уровень несбалансированности потоков,
 $N_{GPU,OP}$ – число операций в потоках GPU,
 $N_{GPU,MEM}$ – число операций обращения к памяти в потоках GPU,
 L – степень локальности данных,
 NS – число потоковых обращений к памяти.

На вход модели подается исходный код проектируемой ПС, для которого определяются параметры, необходимые для вычисления соответствующих метрик ПС. Возможно формирование трех групп метрик:

- M_C – группа метрик, характеризующая эффективность стратегии кеширования данных;
- M_V – группа метрик, характеризующая эффективность векторизации вычислений;
- M_G – группа метрик, характеризующая эффективность использования GPU.

Полученные значения метрик ПС оказывают влияние на коэффициент K_v в выражении (1.1). Таким образом, оценка эффективности в соответствии с (1.1) осуществляется следующим образом:

- формируется множество S_O операций, выполняемых в процессе исполнения ПС, потенциально использующих анализируемый вычислительный ресурс (в соответствии с требуемой группой метрик ПС);
- формируется множество S_R операций, использующих данный ресурс;

- определяются метрики соответствующей группы, формирующие значение коэффициента K , характеризующего способ выполнения операций из множества S_R ;

- вычисляется результирующая эффективность в соответствии с (1.1).

Данная модель позволяет оценить эффективность ПС в ГВС с целевой архитектурой, описанной в главе 1.

3.2.1 Метрики, используемые для оценки эффективности использования подсистемы памяти

В таблице 3.5 приведены метрики, которые используются в процессе повышения эффективности использования ПС подсистемы памяти.

Таблица 3.5. Метрики, используемые в процессе повышения эффективности использования ПС подсистемы памяти.

Метрика	Способ оценки
Ускорение программы	$S_P = T_{P1}/T_{P2}$
Ускорение работы с памятью	$S_M = T_{M1}/T_{M2}$
Снижение вероятности кеш - промаха	$R_P = P_1/P_2$

Рассмотрим подробнее предложенные метрики ПС.

Ускорение программы. Для вычисления данного параметра необходимо оценить время выполнения программы. Оценку предлагается осуществлять в соответствии с выражением (3.28).

$$T_P = \sum_{s=1}^{N_s} \left(N_{s,op} * t_{op} + \sum_{g=1}^{NG_s} N_{s,g} (\alpha_{s,g} * t_{mem} + (1 - \alpha_{s,g}) * t_{cache}) \right) \quad (3.28)$$

где N_s – число шагов алгоритма;

s – номер шага алгоритма;

$N_{s,op}$ – число операций обработки данных, выполняемых на s -м шаге алгоритма;

t_{op} – среднее время выполнения операций обработки данных;

NG_s – число подмножеств, содержащих операции обращения к памяти с равными значениями вероятности кеш-промаха в рамках s -го шага алгоритма;

g – номер подмножества, содержащего операции обращения к памяти в рамках s -го шага алгоритма;

$N_{s,g}$ – число операций в подмножестве номер g ;

$\alpha_{s,g}$ – вероятность возникновения кеш-промаха в процессе обращений к памяти в подмножестве g при выполнении s -го шага алгоритма;

t_{mem} – значение времени обращения к основной памяти на ЦВА;

t_{cache} – значение времени обращения к кеш-памяти на ЦВА.

При оценке времени выполнения программы необходимо разбить множество операций доступа к памяти на непересекающиеся подмножества в соответствии со способом доступа к памяти, который влияет на вероятность кеш-промаха; выделить подмножества операций с равными значениями вероятности кеш-промаха.

Ускорение программы в качестве метрики ПС целесообразно использовать в том случае, если корректировка исходного алгоритма приводит к изменению числа арифметико-логических операций. В случае, когда число арифметико-логических операций не меняется, более подходящими являются метрики, оценивающие работу с подсистемой памяти.

Ускорение работы с памятью. Для вычисления данного параметра необходимо оценить время, затрачиваемое на работу с памятью в процессе выполнения программы. Оценку предлагается осуществлять по формуле (3.29).

$$T_M = \sum_{s=1}^{N_s} \sum_{g=1}^{NG_s} N_{s,g} (\alpha_{s,g} * t_{mem} + (1 - \alpha_{s,g}) ** t_{cache}) \quad (3.29)$$

Ускорение работы с памятью в качестве метрики ПС целесообразно использовать в том случае, если корректировка исходного алгоритма не приводит к изменению числа арифметико-логических операций, однако число операций обращения к памяти изменяется.

Снижение вероятности кеш-промаха. Для вычисления данной метрики необходимо выполнить оценку усредненного значения вероятности кеш-промаха в соответствии с (3.30).

$$P = N_{MEM} / (N_{MEM} + N_{CACHE}) \quad (3.30)$$

Для вычисления данного параметра необходимо оценить количество операций работы с основной памятью и кеш-памятью. Оценку предлагается осуществлять по формулам (3.31) и (3.32).

$$N_{MEM} = \sum_{s=1}^{N_s} \sum_{g=1}^{NG_s} (N_{s,g} * \alpha_{s,g}) \quad (3.31)$$

$$N_{CACHE} = \sum_{s=1}^{N_s} \sum_{g=1}^{NG_s} (N_{s,g} * (1 - \alpha_{s,g})) \quad (3.32)$$

Снижение вероятности кеш-промаха в виде метрики ПС целесообразно использовать в случае, если корректировка исходного алгоритма не изменяет количество арифметико-логических операций и операций работы с памятью.

Стоит отметить, что в указанных метриках используются параметры единственной кеш-памяти, в качестве которой можно использовать любой из кешей L1, L2 и L3. Данный подход основан на группе алгоритмов и методов, имеющих общее название «кеш-независимые алгоритмы», описанные в главе 2. Несмотря на некоторое упрощение системы, данный подход дает хорошие результаты и показывает высокую эффективность. Для проведения более детального анализа необходимо учитывать параметры кешей нескольких уровней.

В отдельных случаях можно использовать другие критерии, а также комплексные показатели, рассчитываемые на основе нескольких отдельных метрик ПС.

3.2.2 Метрики, используемые для оценки эффективности векторизации вычислений

В таблице 3.6 приведены метрики ПС, которые используются в процессе повышения эффективности векторизации вычислений.

Таблица 3.6. Метрики ПС, используемые в процессе повышения эффективности векторизации вычислений.

Метрика	Способ оценки
Увеличение потенциального ускорения	$G_S = S_{PV2}/S_{PV1}$
Снижение доли невыровненных обращений	$R_U = U_1/U_2$

Векторизация вычислений является одним из способов распараллеливания вычислений, поэтому крайне важной характеристикой является получаемое в результате ускорение. Соответственно, первой метрикой ПС в данном случае является метрика увеличения потенциального ускорения (3.33).

$$G_S = S_{PV2}/S_{PV1}, \quad (3.33)$$

где S_{PV} – потенциальное ускорение, вычисляемое в соответствии с (3.34)

$$S_{PV} = N_{SC}/N_{VEC}, \quad (3.34)$$

Необходимо отметить, что характеристика потенциального ускорения часто применяется для оценки качества получаемого в результате векторизации кода. Ниже приведен фрагмент отчета Intel C++ Compiler.

```

LOOP BEGIN at a.cpp(70,2)
    remark #15389: vectorization support: reference ar[i*N+j] has
unaligned access [ a.cpp(71,17) ]
    remark #15388: vectorization support: reference vec[j] has aligned
access [ a.cpp(71,33) ]
    remark #15381: vectorization support: unaligned access used inside
loop body
    remark #15305: vectorization support: vector length 4
    remark #15399: vectorization support: unroll factor set to 2
    remark #15309: vectorization support: normalized vectorization
overhead 2.125
    remark #15448: unmasked aligned unit stride loads: 1
    remark #15450: unmasked unaligned unit stride loads: 1
    remark #15475: --- begin vector cost summary ---
    remark #15476: scalar cost: 8
    remark #15477: vector cost: 2.000
    remark #15478: estimated potential speedup: 3.640
    remark #15488: --- end vector cost summary ---
LOOP END

```

Данный фрагмент отчета компилятора сообщает о векторизованном цикле следующую информацию:

- remark #15389 – невыровненный доступ к `ar[i*N+j]`;
- remark #15388 - выровненный доступ к `vec[j]`;
- remark #15381 – внутри тела цикла используется невыровненный доступ;

- remark #15305 – длина вектора равна 4 (длина вектора в общем случае зависит от используемого типа данных и используемой технологии векторизации, влияющей на ширину векторного регистра; в ряде случаев на этот показатель может оказывать влияние расположение данных в памяти);

- remark #15399 – фактор раскрутки цикла;

- remark #15309 – “накладные расходы на векторизацию” показывают нормализованную (по стоимости итерации вектора) стоимость инициализации/завершения вектора до/после цикла;

- remark #15448, remark #15450 – количество загрузок и сохранений данных (выровненные / невыровненные обращения, равномерный / неравномерный шаг, маскированные / немаскированные (все подряд) загрузки / сохранения);

- remark #15399 – «стоимость» цикла в скалярных операциях;

- remark #15477 – «стоимость» цикла в векторных операциях;

- remark #15478 – ожидаемое ускорение (может не равняться отношению предыдущих двух параметров из-за разных долей накладных расходов в скалярной и векторной реализациях).

Видно, что компилятор рассчитывает характеристику потенциального ускорения, поэтому он может быть использован для расчета метрики увеличения потенциального ускорения.

В процессе выполнения итераций разработки или модификации ПС выполняются только те итерации, которые не приводят к ухудшению показателей эффективности ПС, в частности, не уменьшающие величину потенциального ускорения.

Видно, что большое внимание компилятором уделяется выравниванию векторизуемых данных. Это связано с тем, что обращение к невыровненным данным снижает эффективность ПС. Несмотря на то, что проблема обращения к невыровненным адресам может быть решена за счет формирования прологов и эпилогов циклов, причем доля накладных расходов, связанных с ними, уменьшается с ростом размера массива,

существует проблема, которая, как было описано в главе 1, не решена в современных оптимизирующих компиляторах. Данная проблема связана с доступом к нескольким массивам. Если в цикле используются несколько массивов, выровнены они по-разному, то компилятор выбирает один из них, обработка этого массива будет осуществляться с использованием выровненных считываний из памяти, а обработка остальных – с использованием невыровненных обращений; пролог и эпилог цикла формируется на основе выбранного массива. Если обратиться к примеру отчета компилятора, приведенному выше, можно обратить внимание, что в результате оптимизации проблема невыровненного доступа к массиву `ar[]` решена не была.

В связи с этим предлагается ввести новую метрику ПС, связанную с эффективностью векторизации вычислений – снижение доли невыровненных обращений (3.35).

$$R_U = U_1 / U_2, \quad (3.35)$$

где U - доля невыровненных обращений (3.36)

$$U = N_U / N_T. \quad (3.36)$$

В отчетах компилятора отсутствует данная метрика, есть только сведения о наличии или отсутствии невыровненного доступа к данным. Количественная оценка данного параметра дает возможность судить о степени эффективности проектируемой ПС, повышать эффективность ПС путем максимизации значения параметра (3.35).

3.2.3 Метрики, используемые для оценки эффективности использования GPU

В таблице 3.7 приведены метрики, которые используются в процессе повышения эффективности использования GPU..

Таблица 3.7. Метрики, используемые в процессе повышения эффективности использования GPU.

Метрика	Способ оценки
Увеличение доли использованных потоков	$G_{TU} = TU_2 / TU_1$
Увеличение средней плотности вычислений	$G_D = D_2 / D_1$
Увеличение средней степени локальности данных	$G_L = L_2 / L_1$
Увеличение максимальной степени сбалансированности потоков	$G_{B, max} = B_{1, max} / B_{2, max}$
Увеличение средней степени сбалансированности потоков	$G_{B, av} = B_{1, av} / B_{2, av}$

Метрики, приведенные в таблице 3.7, описывают различные аспекты эффективности использования вычислительных ресурсов GPU.

Доля использованных потоков TU демонстрирует, какая доля из вычислительных потоков, предоставляемых GPU, используется в вычислениях. Чем ближе этот параметр к единице, тем более полно используются вычислительные ресурсы GPU. Увеличение доли использованных потоков, рассчитываемое в соответствии с (3.37), показывает, как изменился данный параметр на следующей итерации проектирования или модификации. Если данный параметр увеличился, то итерация является успешной, в противном случае – неуспешной.

$$G_{TU} = TU_2 / TU_1 \quad (3.37)$$

где TU - доля использованных потоков (3.38), .

$$TU = \sum_{i=1}^{NB} NT_i / (NB * NT_b) \quad (3.38)$$

где NB - число блоков потоков;

NT_b - число потоков в блоке;

NT_i - число использованных потоков в i -м блоке.

Большое влияние на эффективность реализаций ПС на GPU оказывает плотность вычислений, которая показывает отношение количества вычислительных операций к количеству запросов в память. Чем меньше плотность вычислений, тем больше производительность зависит от скорости обмена данными с памятью, чем выше – от скорости вычислений. В современных вычислительных устройствах скорость вычислений превышает скорость обмена, поэтому плотность вычислений необходимо повышать. Повышение производительности в данном случае связано с тем, что данные, загруженные один раз, участвуют во множестве арифметико-логических операций. Для каждого конкретного устройства GPU существуют теоретические значения плотности вычислений, описывающие максимальное количество вычислительных операций, выполняемых за время одной операции обмена с памятью. Но данные значения обычно значительно превышают значение плотности вычислений, достигаемых в различных программных реализациях алгоритмов решения вычислительных задач. Поэтому повышение плотности вычислений является важным аспектом повышения эффективности ПС. Соответственно, следующей метрикой ПС является увеличение средней плотности вычислений (3.39).

$$G_D = D_2 / D_1, \quad (3.39)$$

где D - средняя плотность вычислений (3.40).

$$D = \left(\sum_{b=1}^{NB} \sum_{t=1}^{NT_b} N_{op,b,t} / N_{mem,b,t} \right) / (NB * NT_b) \quad (3.40)$$

где $N_{op,b,t}$ – число операций обработки данных в t -м потоке b -го блока;

$N_{mem,b,t}$ – число операций обращения к памяти в t -м потоке b -го блока.

В соответствии с 3.40 для каждого потока вычисляется плотность вычислений, данные значения суммируются и делятся на общее количество

потоков. В результате получается среднее значение плотности вычислений в расчете на один поток.

Эффективность ПС, использующей GPU, зависит также от способа доступа к данным. Для эффективного обращения к данным необходимо выполнение двух условий:

- данные должны быть выровнены соответствующим образом;
- потоки внутри варпа должны обращаться к данным, расположенным в памяти последовательно.

Следующая метрика, увеличение средней степени локальности данных (3.41), оценивает второе условие доступа к данным, а именно расположение данных в памяти, к которым осуществляется обращение потоками одного варпа.

$$G_L = L_2/L_1, \quad (3.41)$$

где L – средняя степень локальности данных (3.42).

$$L = \left(\sum_{b=1}^{NB} \sum_{w=1}^{NW_b} NS_{b,w} / N_{b,w} \right) / (NB * NW_b) \quad (3.42)$$

где NW_b – число варпов в блоке;

$NS_{b,w}$ – число потоковых обращений к памяти в w -м варпе b -го блока;

$N_{b,w}$ – общее число обращений к памяти в w -м варпе b -го блока.

В соответствии с 3.42 для каждого варпа определяется степень локальности данных как отношение количества потоковых обращений к общему количеству обращений, данные значения суммируются и делятся на общее количество варпов. В результате получается среднее значение степени локальности данных в расчете на один варп.

Увеличение степени сбалансированности потоков. Вводится две метрики - увеличение максимальной степени сбалансированности потоков и увеличение средней степени сбалансированности потоков.

Данные метрики связаны с проблемой расхождения потоков, которая оказывает сильное влияние на итоговую эффективность ПС. Под расхождением потоков понимают ситуацию, когда потоки в одном варпе выполняют различный код. Чаще всего это связано с наличием инструкций ветвления, где условием является номер потока. Особенность CUDA заключается в том, что потоки в одном варпе либо выполняют одну и ту же инструкцию, либо простаивают. Соответственно, расхождение потоков приводит к частично-последовательному выполнению кода в варпе и снижает результирующую эффективность ПС. Расхождение потоков приводит к снижению сбалансированности потоков варпа. Соответственно следующей метрикой является увеличение максимальной степени сбалансированности потоков (3.43).

$$G_{B,max} = B_{1,max} / B_{2,max}, \quad (3.43)$$

где B_{max} – максимальный уровень несбалансированности потоков (3.44).

$$B_{max} = \max_w \left(\max_i \left(N_{op,i,w} / \sum_{k=1}^{NT_w} N_{op,k,w} \right) \right) \quad (3.44)$$

где w - номер варпа;

i, k - номер потока в варпе;

NT_w – число потоков в варпе;

$N_{op,i,w}$ - число операций в i -м потоке w -го варпа;

$N_{op,k,w}$ - число операций в k -м потоке w -го варпа.

Увеличение средней степени сбалансированности потоков (3.45).

$$G_{B,av} = B_{1,av} / B_{2,av}, \quad (3.45)$$

где B_{av} - средний уровень несбалансированности потоков (3.46).

$$B_{av} = \sum_{w=1}^{NW} \sum_{i=1}^{NT_w} \left(N_{op,i,w} / \sum_{k=1}^{NT_w} N_{op,k,w} \right) / (NW * NT_w) , \quad (3.46)$$

где NW – общее число варпов.

Если потоки в варпах сбалансированы, то оба параметра (3.44) и (3.46) будут равны $1/NT_w$. В остальных случаях значение параметров будет выше.

3.2.4 Прикладной протокол определения метрик ПС

Проведем оценку метрик ПС, используемых для оценки эффективности использования подсистемы памяти. Как видно, в приведенных выше выражениях (3.28) – (3.32) важную роль играет оценка вероятности кеш-промаха для каждой из групп операций доступа к памяти. Для выработки метода оценки вероятности кеш-попаданий необходимо рассмотреть несколько ситуаций.

3.2.4.1 Оценка вероятности «холодных» кеш-промахов

Каждая кеш-строка при первом обращении вызывает обязательный кеш-промах. Таким образом, количество обязательных промахов равно количеству кеш-строк, которые использовались хотя бы один раз. Избежать подобных промахов можно только в случае выполнения предвыборки необходимого блока памяти. На данном этапе анализа предвыборка данных не учитывается, однако, в целом она может оказать положительное влияние на показатель кеш-попаданий и на эффективность работы кеш-памяти.

Обязательные или «холодные» промахи генерируются в следующих случаях:

- внецикловой доступ к данным, без последующих повторных обращений (единственный проход по массиву данных, например);

- первый проход по данным в рамках циклового доступа (первый проход по массиву).

Рассматриваем регулярный доступ к данным, где обращения осуществляются к блокам данным, разделенным промежутками, при этом длины блоков данных и длины промежутков являются константами на всем протяжении доступа к данным.

Сначала рассмотрим ситуацию, когда данные расположены в памяти непрерывно. В этом случае кеш-промах генерируется при обращении к первому элементу в очередной кеш-строке, доступ к остальным элементам будет осуществляться из кеш-памяти. Вероятность кеш-попадания в данном случае можно оценить следующим образом.

$$P_{ch} = \frac{n - 1}{n} \quad (3.47)$$

где n определяется как

$$n = \frac{L}{B} \quad (3.48)$$

где L – длина кеш-строки в байтах,

B – длина (sizeof) переменной используемого типа данных в байтах.

Например, при длине строки $L = 64$, использовании типа `double` (`sizeof(double) = 8`), получаем $n=64/8=8$, тогда $P_{ch}=7/8$. При использовании типа `float` (`sizeof(float) = 4`), получаем $n=64/4=16$, тогда $P_{ch}=15/16$.

При изменении схемы доступа, например, считывании значений не подряд, а с некоторыми интервалами, вероятность кеш-попаданий можно оценить как

$$P_{ch} = \frac{k - 1}{k} \quad (3.49)$$

где k – число элементов, используемых в кеш-строке.

Подобную схему доступа иллюстрирует рисунок 3.14.

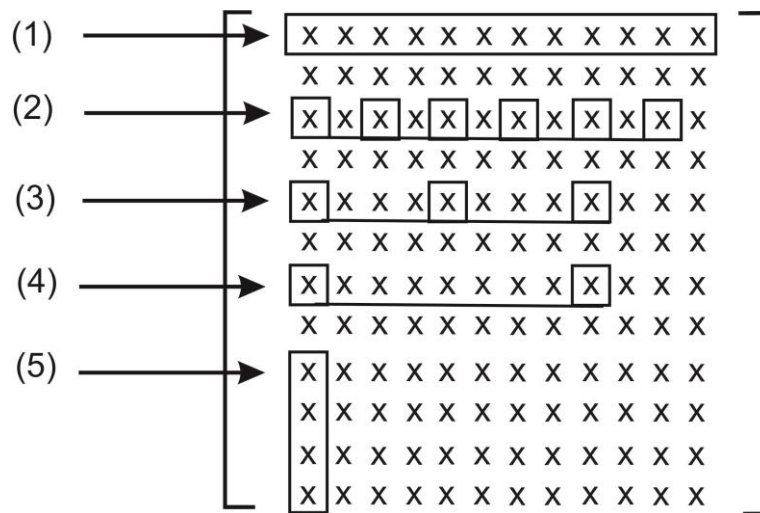


Рисунок 3.14 – Различные схемы доступа к данным в памяти.

На рисунке 3.14 приведены пять схем обращения к элементам матрицы, расположенной в памяти и хранящейся по строкам:

- последовательное обращение к элементам матрицы (1);
- обращение к каждому второму элементу (2);
- обращение к каждому четвертому элементу (3);
- обращение к каждому восьмому элементу (4);
- обращение к элементам матрицы по столбцам (5).

При последовательном обращении (1) вероятность кеш-попаданий рассчитывается в соответствии с 3.47 и зависит от используемого типа данных. В остальных случаях необходимо использовать выражение (3.49), в котором необходимо оценить значение параметра k . Так, например, при использовании типа `double` и обращении к каждому второму элементу (2) $k=4$, т.к. используется лишь 4 элемента из 8. При обращении к каждому четвертому элементу $k = 2$, к каждому восьмому $k = 1$. При использовании типа `float` ситуация изменится, т.к. в этом случае в одну кеш – строку помещается 16 элементов. В таблице 3.8 приведены значения вероятностей попаданий для различных типов данных. В качестве примера используются `double` (8 байт), `float` (4 байта) и `char` (1 байт).

Таблица 3.8 – Вероятность кеш-попадания для различных типов данных.

	double	float	char
по строкам	7/8	15/16	63/64
каждый 2 элемент	3/4	7/8	31/32
каждый 4 элемент	1/2	3/4	15/16
каждый 8 элемент	0	1/2	7/8
каждый 16 элемент	0	0	3/4
каждый 32 элемент	0	0	1/2
каждый 64 элемент	0	0	0
по столбцам	0	0	0

Следует отметить, что при рассмотрении пятой схемы обращения к данным на рисунке 3.14 необходимо учитывать, что выводы, сделанные выше справедливы в случае выбора массивов, которые имеют размер достаточный, чтобы занять весь объем кеш-памяти; тогда к тому моменту, когда обработан последний элемент в первом столбце, строка кеш-памяти, содержащая его первый элемент будет уже вытеснена в основную память и ее придется подгружать заново; в этом случае частота кеш-промахов для результирующего массива действительно может вырасти до единицы (а частота кеш-попаданий упадет до нуля), что приведет к необходимости обращаться к основной памяти каждый раз, когда необходима загрузка нового элемента.

Также необходимо отметить, что описанный выше подход позволяет лишь приблизительно оценить вероятность кеш-промахов. Это происходит по нескольким причинам. Основная причина заключается в том, что алгоритмы работы кеш-памяти отличаются в ВС различных производителей, кроме того, эти алгоритмы постоянно совершенствуются. Поэтому в случаях,

когда вероятность кеш-промаха была оценена как «1», в реальности она может быть несколько меньше, т.к. при последующих загрузках необходимые данные могли быть еще не вытеснены из кеш-памяти.

3.2.4.2 Оценка вероятности кеш-промахов при цикловом доступе к данным

При доступе к данным в цикле ситуация меняется за счет многократного обращения к одним и тем же кеш-линиям. Рассмотрим метод определения вероятности кеш-промаха на основе статического анализа исходного кода на примере обработки одного массива, обход которого осуществляется аналогично обходу массивов в алгоритме матричного умножения. Пусть обработка массива осуществляется в соответствии со следующим псевдокодом.

```
void ijk_array(double *const A, const int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                A[i * N + k] = func(A[i * N + k], i, j, k);
            }
        }
    }
}
```

Пусть функция `func()` не выделяет дополнительную память, работает только с переданными параметрами. Данный подход позволит рассмотреть и проанализировать основные шаблоны (паттерны) доступа к данным в цикле.

В приведенном случае обращение осуществляется вдоль строк матрицы *A*, осуществляется *N* проходов вдоль каждой строки (рисунок 3.15). Такой порядок обхода назовем Паттерн 1.

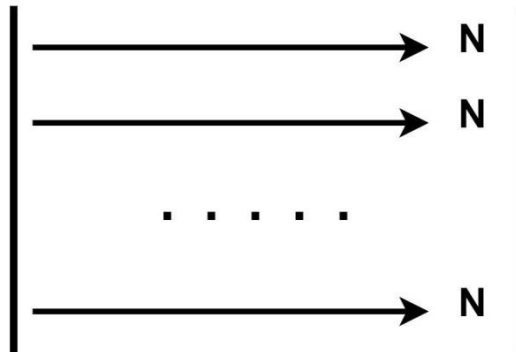


Рисунок 3.15 – Обход матрицы в соответствии с Паттерном доступа 1.

Проанализируем данный паттерн доступа. Возможны два различных случая:

- массив помещается в кеш-память;
- массив в кеш-память не помещается.

Расчет метрик ПС (3.28) – (3.32) для различных уровней кеш-памяти необходимо производить отдельно. Поэтому в данной оценке примем допущение, что в подсистеме памяти присутствует кеш-память с одним уровнем, и оценим усредненную вероятность кеш-промаха для подобной ВС («кеш-независимые алгоритмы», глава 2). Несмотря на сделанные допущения, подобный подход дает достоверные результаты и позволяет оценивать эффективность проектируемой ПС.

В случае, когда массив помещается в кеш-память, кеш-промахи будут генерироваться только при первом проходе вдоль каждой строки. Примем дополнительно следующие допущения:

- используется тип данных `double` протяженностью 8 байт;
- длина кеш-линии составляет 64 байта, т.е. одна кеш-линия содержит 8 элементов базового типа.

Тогда усредненная вероятность кеш-промаха для данного случая будет равна

$$P = \left[N^2 / 8 \right] / N^3 = 1 / (8 * N). \quad (3.50)$$

Действительно, общее число элементов матрицы равно N^2 , при обходе матрицы вдоль строк генерируется $N^2/8$ кеш-промахов, а общее число обращений к памяти равно N^3 .

Вероятность (3.50) мала и уменьшается с ростом размерности задачи при условии, что матрица помещается в кеш-память. Однако массив может быть помещен в кеш-память далеко не всегда, рассчитаем усредненную вероятность кеш-промаха для данного случая.

Введем дополнительный параметр, указывающий на размер строки матриц (3.51).

$$k = \begin{cases} 1, & \text{если } N * b < (V - \partial) \\ 0, & \text{если } N * b \geq (V - \partial) \end{cases} \quad (3.51)$$

где b – размер одного элемента в байтах (8 для double);

V – доступный объем кеш-памяти.

∂ – поправочный коэффициент, означающий, что кроме строки матрицы в кеш-память должны поместить еще несколько кеш-линий, содержащих переменные циклов и другие вспомогательные переменные.

Тогда вероятность кеш-промаха для данного паттерна доступа может быть рассчитана следующим образом (таблица 3.9).

Таблица 3.9 – Вероятность кеш-промаха для Паттерна доступа 1.

Количество обращений к основной памяти N_{MEM}	Количество обращений к кеш-памяти N_{CACHE}	Вероятность кеш- промаха P
$(1 - k) * \frac{N^3}{8} + k * \frac{N^2}{8}$	$N^3 - k * \frac{N^2}{8} - (1 - k) * \frac{N^3}{8}$	$\frac{((1 - k) * N + k)}{8 * N}$

Другими словами, до тех пор, пока обрабатываемая строка матрицы помещается в кеш-память, вероятность кеш-промаха равна $1/(8 * N)$, как только становится она слишком большой, вероятность вырастает до $1/8$, т.е. промах генерируется при обращении к каждой кеш-линии.

Рассмотренную схему доступа назовем ijk по порядку следования циклов, очевидно, что можно предложить еще пять схем доступа.

Паттерн 2 – jki ; обращение вдоль столбцов матрицы, N проходов вдоль матрицы по столбцам (рисунок 3.16).

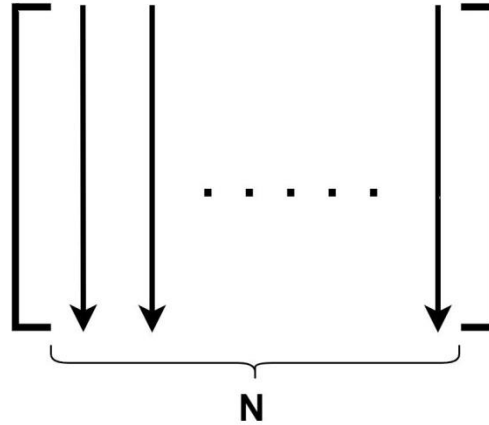


Рисунок 3.16 – Обход матрицы в соответствии с Паттерном доступа 2.

Введем еще один дополнительный параметр (3.52).

$$l = \begin{cases} 1, \text{если } d * N * b < (V - \partial) \\ 0, \text{если } d * N * b \geq (V - \partial) \end{cases} \quad (3.52)$$

где d – количество элементов в кеш-линии (8 в рассматриваемом случае).

Другими словами, $d=1$, если в кеш-память помещаются 8 столбцов матрицы. Тогда при обращении к первому элементу матрицы в кеш-памяти оказываются следующие семь элементов и т.д. При обращении ко второму столбцу, его первый элемент будет к этому времени закеширован.

Вероятность кеш-промаха для данного паттерна доступа может быть рассчитана следующим образом (таблица 3.10).

Таблица 3.10 – Вероятность кеш-промаха для Паттерна доступа 2.

Количество обращений к основной памяти N_{mem}	Количество обращений к кеш-памяти N_{cache}	Вероятность кеш- промаха P
$(1 - l) * N^3 + l * \frac{N^3}{8}$	$N^3 - (1 - l) * N^3 - l * \frac{N^3}{8}$	$(1 - l) + \frac{l}{8}$

До тех пор, пока 8 столбцов матрицы помещаются в кеш-память, вероятность кеш-промаха равна $1/8$, если перестают помещаться, вероятность вырастает до 1, т.е. промах генерируется при обращении к каждому элементу матрицы.

Паттерн 3 – ikj ; один проход вдоль матрицы по строкам, в рамках каждого прохода N обращений к каждому элементу (рисунок 3.17).

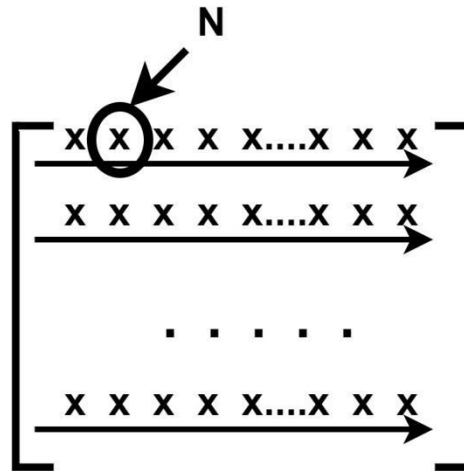


Рисунок 3.17 – Обход матрицы в соответствии с Паттерном доступа 3.

Вероятность кеш-промаха для данного паттерна доступа может быть рассчитана следующим образом (таблица 3.11).

Таблица 3.11 – Вероятность кеш-промаха для Паттерна доступа 3.

Количество обращений к основной памяти N_{mem}	Количество обращений к кеш-памяти N_{cache}	Вероятность кеш- промаха P
$\frac{N^2}{8}$	$N^3 - \frac{N^2}{8}$	$\frac{1}{8 * N}$

Т.к. обращение осуществляется вдоль строк, то вероятность кеш-промаха равна $1/8$, за счет того, что к каждому элементу обращение производится N раз, вероятность уменьшается в N раз.

Паттерн 4 – jik ; обращение вдоль строк матрицы, N проходов вдоль матрицы (рисунок 3.18)

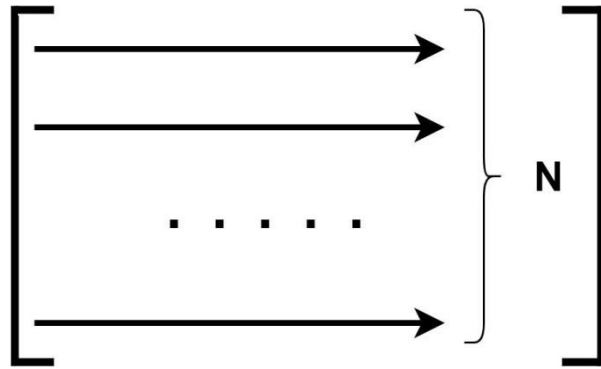


Рисунок 3.18 – Обход матрицы в соответствии с Паттерном доступа 4

Вероятность кеш-промаха для данного паттерна доступа может быть рассчитана следующим образом (таблица 3.12).

Таблица 3.12 – Вероятность кеш-промаха для Паттерна доступа 4.

Количество обращений к основной памяти N_{mem}	Количество обращений к кеш-памяти N_{cache}	Вероятность кеш- промаха P
$\frac{N^3}{8}$	$\frac{7 * N^3}{8}$	$\frac{1}{8}$

Т.к. матрица целиком не помещается в кеш-память, то промахи генерируются обращениями к каждой очередной кеш-линии.

Паттерн 5 – kji ; обращение вдоль столбцов, N проходов вдоль каждого столбца (рисунок 3.19)

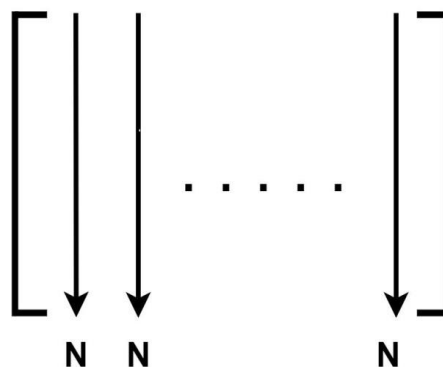


Рисунок 3.19 – Обход матрицы в соответствии с Паттерном доступа 5.

Вероятность кеш-промаха для данного паттерна доступа может быть рассчитана следующим образом (таблица 3.13).

Таблица 3.13 – Вероятность кеш-промаха для Паттерна доступа 5.

Количество обращений к основной памяти N_{mem}	Количество обращений к кеш-памяти N_{cache}	Вероятность кеш- промаха P
$(1 - l) * N^3 + l * \frac{N^2}{8}$	$N^3 - (1 - l) * N^3 - l * \frac{N^2}{8}$	$(1 - l) + \frac{l}{8 * N}$

До тех пор, пока восемь столбцов матрицы помещаются в кеш-память, вероятность кеш-промаха равна $1/(8 * N)$, как только перестают помещаться, вероятность вырастает до 1, т.е. промах генерируется при обращении к каждому элементу матрицы.

Паттерн 6 – kij ; один проход вдоль матрицы по столбцам, в рамках каждого прохода N обращений к каждому элементу (рисунок 3.20)

Вероятность кеш-промаха для данного паттерна доступа может быть рассчитана следующим образом (таблица 3.14).

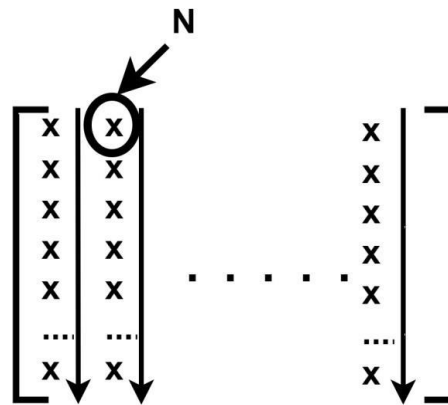


Рисунок 3.20 – Обход матрицы в соответствии с Паттерном доступа 6.

Таблица 3.14 – Вероятность кеш-промаха для Паттерна доступа 6.

Количество обращений к основной памяти N_{mem}	Количество обращений к кеш-памяти N_{cache}	Вероятность кеш- промаха P
$l * \frac{N^2}{8} + (1 - l) * N^2$	$N^3 - l * \frac{N^2}{8} - (1 - l) * N^2$	$\frac{(1 - l)}{N} + \frac{L}{8 * N}$

Если восемь столбцов не помещаются в кеш-память, вероятность кеш-промаха равна $1/N$ за счет того, что к каждому элементу производится N обращений, если помещаются, вероятность кеш-промаха уменьшается в восемь раз.

.В таблицы 3.15 приведены вероятности кеш-промахов для различных паттернов доступа к данным. Расчеты, приведенные в таблице 3.15, не учитывают предвыборку данных.

Современные МП выполняют предвыборку данных, которая заключается в том, что данные загружаются в кеш-память до того, как к ним будет осуществлено обращение, что сокращает простои МП, которые связаны с ожиданием данных из памяти. Различают аппаратную и программную предвыборку данных, они были описаны ранее в главе 1.

В случае аппаратной предвыборки МП пытается распознать паттерн доступа к данным и заранее загрузить кеш-линию, которая будет необходима на следующих этапах вычислений. Можно выделить несколько вариантов аппаратной предвыборки данных:

Таблица 3.15 – Вероятности кеш-промахов для различных паттернов доступа.

Паттерн доступа	Вероятность кеш-промаха P
Паттерн 1	$\frac{(1-k)}{8} + \frac{k}{8 * N}$
Паттерн 2	$(1-l) + \frac{l}{8}$
Паттерн 3	$\frac{1}{8 * N}$
Паттерн 4	$\frac{1}{8}$
Паттерн 5	$(1-l) + \frac{l}{8 * N}$
Паттерн 6	$\frac{(1-l)}{N} + \frac{L}{8 * N}$

- потоковая предвыборка (stream prefetching), выбирается следующая кеш-линия в порядке следования адресов; самый простой вариант, реализуется всеми современными МП;

- пошаговая предвыборка (stride prefetching), МП анализирует адреса обращений к памяти: определяется шаг, с которым в памяти следуют кеш-линии, к которым выполняются обращения, после чего предвыборка осуществляется с этим шагом.

В обоих случаях возможны как серьезное увеличение производительности за счет уменьшения количества кеш-промахов в результате заблаговременного помещения необходимых данных в кеш-память, так и снижение производительности за счет предвыборки ненужных данных, вытеснения из кеш-памяти нужных данных и увеличения вероятности кеш-промахов.

В предлагаемой модели будем учитывать предвыборку, которая сочетает в себе элементы как потоковой, так и пошаговой предвыборки. Считаем, что алгоритм предвыборки следующий:

- если произошло более одного обращения с одним и тем же шагом в памяти, то будет осуществлена предвыборка следующего элемента.

Другими словами, если произошло обращение к элементам массива $a[N * i]$ и $a[N * (i + 1)]$, то будет осуществлена предвыборка элемента массива $a[N * (i + 2)]$. В зависимости от значения N будет реализована потоковая или пошаговая предвыборка.

Алгоритмы работы блоков предвыборки данных в современных МП используют более сложные алгоритмы, в частности используются несколько потоков предвыборки, т.е. осуществляется предвыборка сразу нескольких кеш-линий, могут использоваться комбинация нескольких паттернов доступа и т.д. Однако, алгоритм, используемый в данной модели, также обладает характеристиками, позволяющими делать достаточно точные оценки эффективности различных паттернов доступа в память.

С учетом предложенных ранее метрик ПС, представляется возможным учет влияния предвыборки данных на конечную эффективность ПС двумя способами:

- учесть предвыборку в метрике ускорения работы с памятью за счет внесения изменений в формулы расчета времени, затрачиваемого на работу с памятью в процессе выполнения программы;
- учесть предвыборку в метрике снижения вероятности кеш-промаха за счет внесения изменений в методику определения усредненного значения вероятности кеш-промаха.

В случае использования перестановочных оптимизаций целесообразно использовать второй вариант, т.к. в случае использования метрики, связанной со временем работы с памятью, придется использовать характеристики конкретных МП, в случае оценки вероятностей кеш-промаха, можно делать оценки вне зависимости от типа МП с учетом описанных выше допущений.

Количество обращений к памяти будем оценивать следующим образом:

$$N'_{mem} = N_{mem} * (b + g * (1 - b)). \quad (3.53)$$

где N_{mem} – число обращений к памяти, оцененное без учета предвыборки данных;

N'_{mem} – число обращений к памяти, оцененное с учетом предвыборки данных;

b – доля неуспешных предвыборок (промахи без предвыборки, т.е. кеш-линия не была предвыбрана); зависит от паттерна доступа к данным;

g – доля времени на считывание предвыбранной ранее строки от полного времени обращения к памяти без предвыборки; в идеале равна 0; в реальности зависит от аппаратуры МП, отношения дистанции предвыборки к длине паттерна (полностью предвыбраны к данному моменту могут быть не все кеш-линии) и плотности вычислений (чем больше плотность, тем больше шанс, что МП полностью загрузит следующую строку).

Количество обращений к кеш-памяти в этом случае будем оценивать следующим образом:

$$N'_{cache} = N_{cache} + N_{mem} * [1 - (b + g * (1 - b))]. \quad (3.54)$$

где N_{cache} – число обращений к кеш-памяти, оцененное без учета предвыборки данных;

N'_{cache} – число обращений к кеш-памяти, оцененное с учетом предвыборки данных

В процессе анализа примем $g = 0$, что даст нам верхнюю (лучшую) оценку исследуемых паттернов доступа. Оценки, приведенные в таблице 3.15, дают нижнюю (худшую) оценку исследуемых схем.

Т.к. общее число операций обращения к памяти в результате выполнения перестановочных оптимизаций не меняется, необходимо пересчитать число обращений в память, выполняемых в результате кеш-промахов для каждого паттерна доступа в соответствии с (3.53).

Определим значение параметра b для всех используемых паттернов доступа к данным.

Паттерн 1 - обращение вдоль строк, N проходов вдоль каждой строки.

При переходе с i -й строки на строку с номером $(i + 1)$, первый элемент строки $(i + 1)$ уже предвыбран с использованием механизма потоковой предвыборки. Однако на вероятность кеш-промаха также влияет, помещается ли строка матрицы в кеш-память (это зависит от значения k): в частности, промахи при обращении к первым элементам строк матрицы возникают именно в тех случаях, когда строка целиком не умещается в кеше.

.Если строка помещается в кеш-память ($k=1$), то промах будет сгенерирован только при обращении к первому элементу матрицы. Если строка не помещается, то промахи будут генерироваться каждый раз при обращении к первому элементу первой строки (к концу обработки строки он будет вытеснен из кеш-памяти) и при вторых и дальнейших обращениях к первым элементам остальных строк (при первом обращении кеш-линия предвыбрана во время обработки предыдущей строки) (3.55).

$$b_1 = \begin{cases} 8/N^2, & \text{если } k = 1 \\ [8 * (N + (N - 1)^2)]/N^3, & \text{если } k = 0 \end{cases} \quad (3.55)$$

Параметр b для паттерна доступа 1 определим следующим образом

$$b_1 = 8/N^2 * k + \left[[8 * (N + (N - 1)^2)]/N^3 \right] * (1 - k). \quad (3.56)$$

Таким образом, общее количество обращений к основной памяти для паттерна 1 равно

$$N'_{mem} = \begin{cases} 1, & \text{если } k = 1 \\ N + (N - 1)^2 \approx N^2, & \text{если } k = 0 \end{cases} \quad (3.57)$$

Параметр N'_{mem} для паттерна доступа 1 определим следующим образом

$$N'_{mem} = (1 - k) * [N + (N - 1)^2] + k. \quad (3.58)$$

Необходимо отметить, что при получении (3.58) использовались выражения (3.59), справедливость которых легко доказывается на основе того, что параметр k может принимать только значения 0 или 1.

$$\begin{aligned} (1 - k) * k &= 0, \\ (1 - k)^2 &= (1 - k), \\ k^2 &= k, \end{aligned} \quad (3.59)$$

Паттерн 2 – N проходов по матрице вдоль столбцов.

Потоковая предвыборка здесь не работает, т.к. паттерн является «столбцовым», т.к. доступ к данным осуществляется вдоль столбцов. Однако доступ осуществляется регулярным способом, соответственно работает пошаговая предвыборка. Параметр b можно определить следующим образом.

$$b_2 = \frac{c}{N} \quad (3.60)$$

Другими словами, при обращении вдоль столбца, промах генерируется при обращении к первому элементу столбца, остальные выбираются с использованием предвыборки. Т.е. вероятность кеш-промаха уменьшается в N раз. При этом вводится дополнительный параметр c , больший единицы,

учитывающий меньшую эффективность столбцовых паттернов доступа в множественно-ассоциативном кеше. Это связано с тем, что для множественно-ассоциативного кеша при доступе с шагом, превышающем длину кеш-линии, кеш-память заполняется неравномерно и промахи могут начать генерироваться до полного заполнения кеш-памяти. Для полностью ассоциативного кеша параметр s равен 1. Кроме того необходимо отметить, что при строковом паттерне доступа кеш-память заполняется равномерно. Таким образом, общее количество обращений к основной памяти для паттерна 2 равно

$$N'_{mem} = \begin{cases} \frac{s * N^2}{8}, & \text{если } l = 1 \\ s * N^2, & \text{если } l = 0 \end{cases} . \quad (3.61)$$

Паттерн 3 - один проход вдоль матрицы по строкам, в рамках каждого прохода N обращений к каждому элементу. По сути, данный паттерн доступа не является цикловым, поэтому в процессе анализа необходимо учитывать только влияние «холодных промахов» и предвыборки. При описании модели было акцентировано внимание, что работы блока предвыборки необходимо осуществлять регулярный доступ с некоторым шагом. В данном паттерне подобный доступ отсутствует (сначала адрес повторяется n раз, далее осуществляется переход к следующему элементу массива), будем считать параметр b равным 1..

$$b_3 = 1 \quad (3.62)$$

Паттерн 4 - обращение вдоль строк, N проходов вдоль матрицы. Промах без предвыборки на каждом первом элементе матрицы при очередном проходе. Всего в процессе обхода матрицы по строкам формируется $(N^2/8)$ обращений к основной памяти, и только на первой кеш-линии произойдет кеш-промах без предвыборки. Таким образом, параметр b для паттерна 4 равен (3.63).

$$b_4 = \frac{8}{N^2} \quad (3.63)$$

Паттерн 5 - обращение вдоль столбцов, N проходов вдоль каждого столбца.

Доступ осуществляется регулярным способом, соответственно работает пошаговая предвыборка. Параметр b можно определить следующим образом.

$$b_5 = \begin{cases} c, & \text{если } l = 1 \\ \frac{c}{N}, & \text{если } l = 0 \end{cases} \quad (3.64)$$

Если 8 столбцов помещаются в кеш-память, то предвыборка при обращении в пределах данных столбцов после их помещения в кеш-память влияния оказывать не будет.

Паттерн 6 - один проход вдоль матрицы по столбцам, в рамках каждого прохода N обращений к каждому элементу.

Так же, как и в случае использования паттерна 3, параметр b здесь принимает значение 1.

Таблица 3.16 – Вероятности кеш-промахов для различных паттернов доступа с учетом предвыборки.

Паттерн доступа	Вероятность кеш-промаха P
Паттерн 1	$\frac{(1-k)}{N} + \frac{k}{N^3}$
Паттерн 2	$\frac{(1-l) * c}{N} + \frac{l * c}{8 * N}$
Паттерн 3	$\frac{1}{8 * N}$
Паттерн 4	$\frac{1}{N^2}$
Паттерн 5	$\frac{(1-l) * c}{N} + \frac{l * c}{8 * N}$
Паттерн 6	$\frac{(1-l)}{N} + \frac{L}{8 * N}$

Анализируя данные в таблице 3.16 можно сделать вывод, что представленные паттерны доступа можно разделить на 3 группы со схожей эффективностью:

- паттерны 1 и 4;
- паттерны 2 и 5;
- паттерны 3 и 6.

Наиболее эффективной является первая группа, наименее – третья. Внутри групп паттерны могут незначительно отличаться друг от друга в зависимости от типа используемого МП. Наибольшие различия в формулах, оценивающих вероятности кеш-промахов, наблюдаются в паттернах из первой группы. Однако можно предположить, что в современных МП, которые имеют возможность отслеживания более одного паттерна доступа, вероятность будет стремиться к выражению для паттерна 4 т.к. для паттерна 1 МП будет отслеживать как доступ вдоль очередной строки, так и доступ по строкам.

Для подтверждения сделанных выводов был проведен вычислительный эксперимент на вычислительной системе, оснащенной МП 11th Gen Intel(R) Core(TM) i9-11900 @ 2.50GHz, который имеет следующие характеристики подсистемы памяти:

- кеш L1D имеет объем 48 кБ и степень ассоциативности 12;;
- кеш L2 имеет объем 512 кБ и степень ассоциативности 8;
- длина кеш-линии составляет 64 байта.

Результаты, представленные на рисунке 3.21, полностью подтверждают выводы, сделанные ранее. На графике представлены зависимости значений ускорения выполнения рассматриваемой операции с использованием различных паттернов доступа по отношению ко времени, необходимому для реализации операции с использованием паттерна ikj (Паттерн 3). Самыми быстрыми оказались паттерны 1 и 4, самыми медленными – 3 и 6. Разница внутри групп паттернов во времени выполнения минимальна.

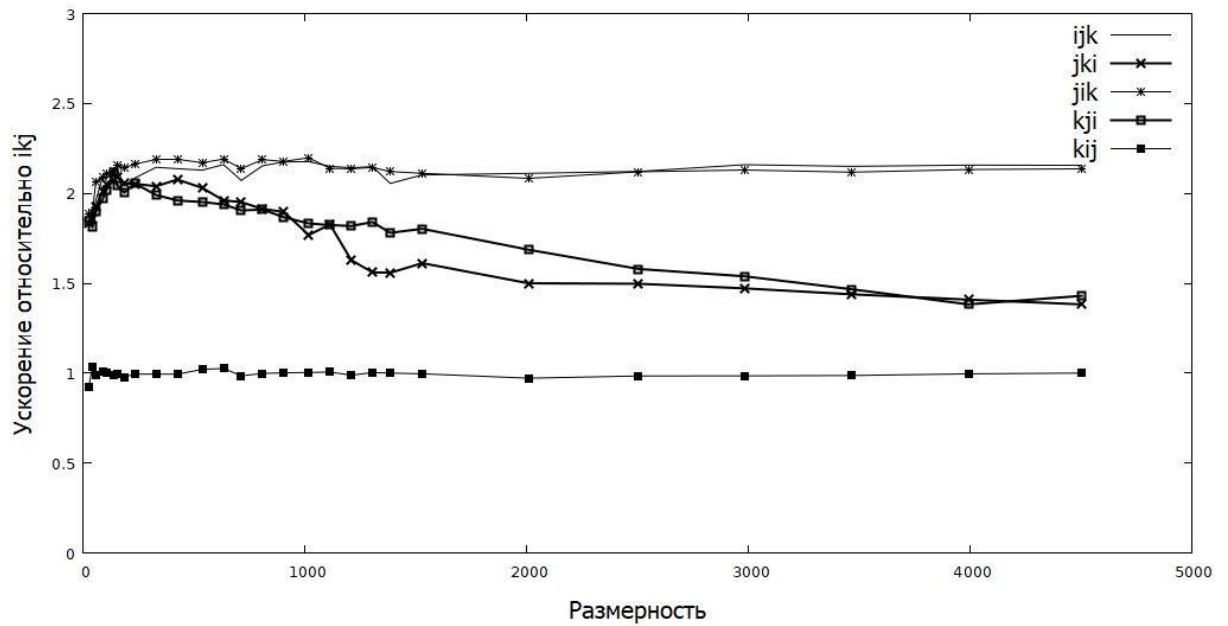


Рисунок 3.21 – Результат вычислительного эксперимента на МП 11th Gen Intel(R) Core(TM) i9-11900 @ 2.50GHz 2.50 GHz.

На рисунке 3.22 приведены результаты вычислительного эксперимента, проведенного на вычислительной системе, оснащенной МП Xeon E5-2650v3(x2) 2.3 GHz.

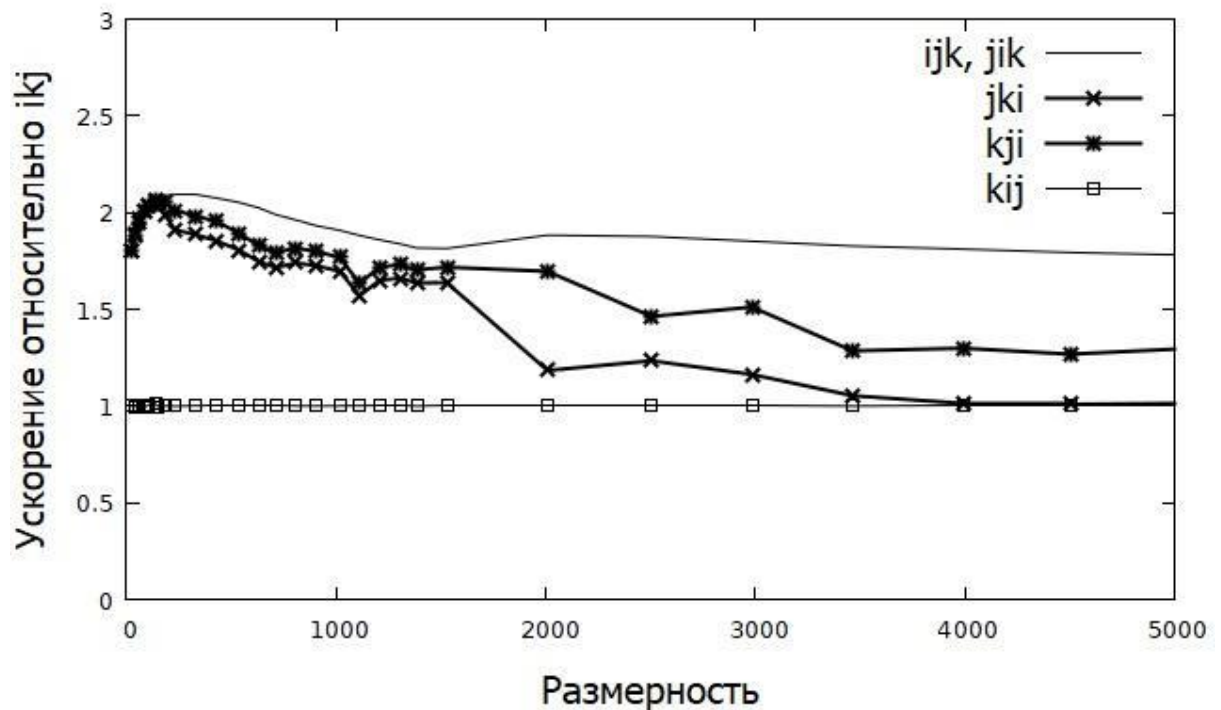


Рисунок 3.22 – Результат вычислительного эксперимента на МП Xeon E5-2650v3(x2) 2.3 GHz

Результаты строковых паттернов доступа практически не отличаются друг от друга, представлены одной линией. Столбцовые паттерны имеют показатели гораздо худшие, чем на рисунке 3.21. Это связано с тем, что на данном МП алгоритм пошаговой предвыборки имеют гораздо более скромные возможности, что неудивительно, т.к. данный МП «старше» МП, использованного для опыта, результат которого представлен на рисунке 3.21. Кроме того, он имеет кеш L1D меньший по размеру и с меньшей степенью ассоциативности, что негативно влияет на столбцовые паттерны доступа. Однако в целом можно сказать, что полученные результаты также подтверждают выводы, сделанные ранее.

Рассмотренные паттерны доступа к данным являются базовыми, на основе которых можно строить более сложные шаблоны обращения к данным, в процессе анализа использовать полученные результаты

Рассмотрим метод определения вероятности кеш-промаха на основе статического анализа исходного кода в случае обработки нескольких массивов на примере задачи матричного умножения. Код реализации базового алгоритма умножения матриц приведен ниже.

```
void ijk(double *const A, double *const B, double *const C, const int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

Для упрощения анализа опустим этап обнуления элементов матрицы C. Здесь также возможны несколько различных вычислительных схем в соответствии с порядком следования циклов (таблица 3.17).

Таблица 3.17. Соответствие различных схем реализации матричного умножения приведенным паттернам доступа.

Вычислительная схема	Паттерны доступа к элементам массивов		
	А	В	С
ijk	Паттерн 1	Паттерн 2	Паттерн 3
jik	Паттерн 4	Паттерн 5	Паттерн 6
ikj	Паттерн 3	Паттерн 4	Паттерн 1
kij	Паттерн 6	Паттерн 1	Паттерн 4
kji	Паттерн 5	Паттерн 3	Паттерн 2
jki	Паттерн 2	Паттерн 6	Паттерн 5

В данном случае необходимо анализировать совокупность используемых паттернов доступа. Введем более общий параметр на базе выражения (3.51).

$$k_n^b = \begin{cases} 1, \text{ если } n * b < (V - \partial) \\ 0, \text{ если } n * b \geq (V - \partial) \end{cases} . \quad (3.65)$$

где n – число элементов заданного типа, помещающихся в кеш-памяти..

При анализе паттернов доступа к одному массиву использовались параметры (k и l), влияющие на эффективность паттернов, В данном случае параметр будет зависеть от группы используемых паттернов (таблица 3.18).

Таблица 3.18.Количество обращений к памяти.

Вычислительная схема	Параметры		
	Паттерны	Кешируемые элементы	k
ijk	1, 2, 3	Строка А, 8 столбцов В	k_{9N}^8
jik	4, 5, 6	8 столбцов В	k_{8N}^8

Продолжение таблицы 3.18

Вычислительная схема	Параметры		
	Паттерны	Кешируемые элементы	k
ikj	3, 4, 1	Строка C	k_N^8
kij	6, 1, 4	Строка B	k_N^8
kji	5, 3, 2	8 столбцов A, 8 столбцов C	k_{16N}^8
jki	2, 6, 5	8 столбцов A, 8 столбцов C	k_{16N}^8

На следующем этапе необходимо посчитать количество обращений в память для всех трех массивов в соответствии с рассчитанными ранее параметрами (3.66-3.71).

$$N'_{mem\ ijk} = [(1 - k_{9N}^8)N^2 + k_{9N}^8]_A + \left[(1 - k_{9N}^8)N^2 c + k_{9N}^8 c \frac{N^2}{8} \right]_B + \left[\frac{N^2}{8} \right]_C \quad (3.66)$$

$$= \frac{N^2[(1 - k_{9N}^8)(8c + 8) + k_{9N}^8 c + 1] + 8k_{9N}^8}{8}$$

$$N'_{mem\ jik} = [N]_A + \left[(1 - k_{8N}^8)cN^2 + k_{8N}^8 c \frac{N^2}{8} \right]_B + \left[k_{8N}^8 \frac{N^2}{8} + (1 - k_{8N}^8)N^2 \right]_C \quad (3.67)$$

$$= \frac{N^2[(1 - k_{8N}^8)(8c + 8) + k_{8N}^8 c + k_{8N}^8] + 8N}{8}$$

$$N'_{mem\ ikj} = \left[\frac{N^2}{8} \right]_A + [N]_B + [(1 - k_N^8)N^2 + k_N^8]_C \quad (3.68)$$

$$= \frac{N^2[8(1 - k_N^8) + 1] + 8N + 8k_N^8}{8}$$

$$N'_{mem\ kij} = \left[k_N^8 \frac{N^2}{8} + (1 - k_N^8)N^2 \right]_A + [(1 - k_N^8)N^2 + k_N^8]_B + [N]_C \quad (3.69)$$

$$= \frac{N^2[16(1 - k_N^8) + k_N^8] + 8N + 8k_N^8}{8}$$

$$N'_{mem\ kji} = \left[(1 - k_{16N}^8)N^2c + k_{16N}^8c \frac{N^2}{8} \right]_A + \left[\frac{N^2}{8} \right]_B \quad (3.70)$$

$$\begin{aligned}
& + \left[(1 - k_{16N}^8)N^2c + k_{16N}^8c \frac{N^2}{8} \right]_C \\
& = \frac{N^2[16c(1 - k_{16N}^8) + 2k_{16N}^8c + 1]}{8} \\
N'_{mem\ jki} & = \left[(1 - k_{16N}^8)N^2c + k_{16N}^8c \frac{N^2}{8} \right]_A + \left[k_{16N}^8 \frac{N^2}{8} + (1 - k_{16N}^8)N^2 \right]_B \quad (3.71) \\
& + \left[(1 - k_{16N}^8)N^2c + k_{16N}^8c \frac{N^2}{8} \right]_C \\
& = \frac{N^2[(1 - k_{16N}^8)(16c + 8) + 2k_{16N}^8c + k_{16N}^8]}{8}
\end{aligned}$$

Далее, с учетом полученных выражений были определены значения снижения вероятности кеш-промаха для различных реализаций матричного умножения относительно базовой схемы ijk с учетом предвыборки, данные приведены в таблице 3.19.

Таблица 3.19. Значения снижения вероятности кеш-промаха для различных реализаций матричного умножения относительно базовой схемы ijk с учетом предвыборки.

Схема умножения	Снижение вероятности кеш-промаха, R_p
jik	$\frac{\left[N^2 \left[(1 - k_{9N}^8) (8c + 8) + k_{9N}^8c + 1 \right] + 8k_{9N}^8 \right]}{\left[N^2 \left[(1 - k_{8N}^8) (8c + 8) + k_{8N}^8c + k_{8N}^8 \right] + 8N \right]}$
ikj	$\frac{\left[N^2 \left[(1 - k_{9N}^8) (8c + 8) + k_{9N}^8c + 1 \right] + 8k_{9N}^8 \right]}{\left[N^2 \left[8 (1 - k_N^8) + 1 \right] + 8N + 8k_N^8 \right]}$
kij	$\frac{\left[N^2 \left[(1 - k_{9N}^8) (8c + 8) + k_{9N}^8c + 1 \right] + 8k_{9N}^8 \right]}{\left[N^2 \left[16 (1 - k_N^8) + k_N^8 \right] + 8N + 8k_N^8 \right]}$
kji	$\frac{\left[N^2 \left[(1 - k_{9N}^8) (8c + 8) + k_{9N}^8c + 1 \right] + 8k_{9N}^8 \right]}{\left[N^2 \left[16c (1 - k_{16N}^8) + 2k_{16N}^8c + 1 \right] \right]}$

Продолжение таблицы 3.19.

Схема умножения	Снижение вероятности кеш-промаха, R_p
jki	$\frac{\left[N^2 \left[(1 - k_{9N}^8) (8c + 8) + k_{9N}^8 c + 1 \right] + 8k_{9N}^8 \right]}{\left[N^2 \left[(1 - k_{16N}^8) (16c + 8) + 2k_{16N}^8 c + k_{16N}^8 \right] \right]}$

Параметр, представленный в таблице 3.19, характеризует снижение значения вероятности кеш-промаха. Значение больше единицы означает, что данная схема лучше базовой, меньше единицы – хуже.

Полученные выражения необходимо прокомментировать. У всех вычислительных схем наивысшая размерность в выражении равна N^2 , отличия заключаются в множителе при этом члене, которые при $k=0$ имеют следующие значения.

- ijk, jik соответственно $(8c + 9)$ и $(8c+8)$;
- ikj, kij соответственно 9 и 16;
- kji, jki – $(16c + 1)$ и $(16c + 8)$.

Параметр c учитывает меньшую эффективность столбцовых паттернов доступа. Если предположить, что предвыборка работает идеально, то значения данного параметра будет стремиться к 1. Однако в реальности значение данного параметра превышает 1, варьируется для различных типов МП в зависимости от эффективности блока предвыборки. Соответственно, лучшими должны быть вычислительные схемы, в которых отсутствуют столбцовые паттерны, в данном случае ikj, kij. Вычислительные схемы, содержащие столбцовые паттерны доступа, обладают худшими показателями эффективности, причем эффективность в данной группе будет зависеть от коэффициента при N^2 ,. Наименее эффективными оказываются вычислительные схемы kji, jki, средние показатели эффективности имеют ijk, jik.

Был проведен вычислительный эксперимент на вычислительном узле, оснащённом МП Xeon E5-2650v3(x2) 2.3 GHz, результаты приведены на рисунке 3.23.

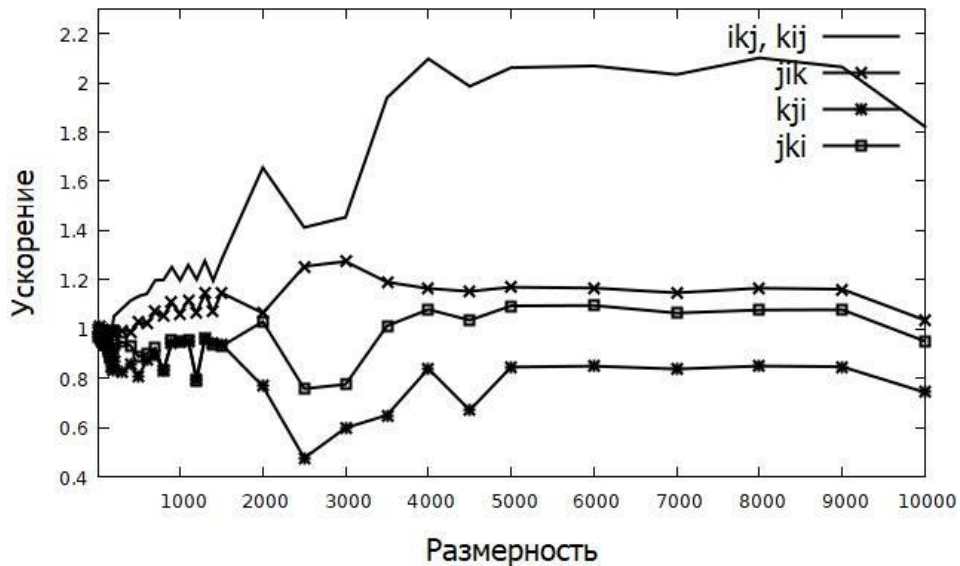


Рисунок 3.23 - Значения ускорения, полученного относительно базовой схемы ijk на МП Xeon E5-2650v3(x2) 2.3 GHz

На рисунке 3.23 показаны ускорения различных схем выполнения матричного умножения по отношению к ijk . Видно, что результаты эксперимента подтверждают сделанные ранее выводы. Однако сделанные выводы опирались на предположения, что во всех выражениях (таблица 3.19) $k=0$ для всех вычислительных схем, что соответствует большим размерностям матрицы.

Влияние параметра k на эффективность различных вычислительных схем матричного умножения зависит от эффективности блока предвыборки. В случае если МП не имел бы данного блока, влияние k было бы хорошо заметно для всех уровней кеш-памяти. Современные МП имеют достаточно эффективные блоки предвыборки, поэтому влияние данных параметров менее акцентировано и в большей степени проявляет себя на больших уровнях кеша. При этом размер $L3$ достаточно большой для того, чтобы оказывать влияние на кеширование отдельных строк матриц, поэтому наибольший эффект достигается на $L2$.

МП, использованный для моделирования, имеет следующие характеристики кеш-памяти:

- длина кеш-линии для всех уровней кеш-памяти равна 64 байта;
- L1 имеет объем 32Кб, степень ассоциативности 8;
- L2 имеет объем 256Кб, степень ассоциативности 8;
- L3 имеет объем 25Мб, степень ассоциативности 20.

Значения максимальной размерности матрицы, при которой соответствующие параметры эффективности принимают единичное значение для кеш-памяти уровней L1 и L2 приведены в таблице 3.20.

Таблица 3.20. Значения максимальной размерности матрицы, при которой соответствующие параметры эффективности принимают единичное значение

Параметр	Максимальная размерность для L1	Максимальная размерность для L2
k_N^8	4096	32768
k_{8N}^8	512	4096
k_{9N}^8	455	3640
k_{16N}^8	256	2048

Стоит также отметить, что проведенный анализ необходим в случае невозможности или неэффективности разработки блочных реализаций алгоритма решения задачи. В случае реализации блочного алгоритма важной проблемой является выбор размеров блоков, используемых в процессе вычислений.

3.3 Аналитический метод количественной оценки метрик использования подсистемы памяти

В области ВПВ большое значение имеет оптимизация ПС под конкретные вычислительные архитектуры. Существуют различные подходы решения данной задачи. Одним из подходов является разработка версий ПС, оптимизированных для конкретных вычислительных архитектур, как это сделано, например, в Intel MKL. Другим подходом к решению данной задачи является настройка ПС для получения наилучшей реализации на конкретной вычислительной архитектуре. Процедура настройки заключается в поиске наилучшей комбинации оптимизационных настроек или параметров. Целью подбора оптимального набора параметров является получение максимально эффективной ПС [44].

Важным приемом оптимизации ПС является разработка блочных алгоритмов реализации известных преобразований, примером здесь может служить матричное умножение. Размер блока необходимо выбирать таким образом, чтобы минимизировать количество кеш-промахов в процессе выполнения вычислений, что в свою очередь влияет на значение метрики ПС – снижение вероятности кеша-промаха. И важным здесь является следующий вопрос: «Каков максимальный размер блока, при котором генерируются только «холодные» промахи и не происходит вытеснения строк из кеш-памяти?». Ответ на данный вопрос является крайне важным для задания значений параметров блочных алгоритмов, которые могут меняться в зависимости от характеристики целевой ВС.

В современных МП кеш-память имеет множественно-ассоциативную организацию, при которой кеш-память делится на несколько банков, каждый из которых работает как кеш с прямым отображением. Строка основной памяти может быть отображена не в одну конкретную строку кеша, а в некоторое множество строк, т.е. в один из банков. При последовательном обращении по адресам основной памяти, такая организация кеш-памяти

приводит к равномерному заполнению множеств кеша. Однако, при использовании других шаблонов доступа, ситуация меняется.

Рассмотрим процесс отображения регулярного шаблона доступа в память на кеш-память. Т.к. попадание в то или иное множество зависит от адреса операнда (номера строки), условие попадания в i -ое множество может быть представлено следующим образом

$$R \% N = i, \quad (3.72)$$

где R – номер строки, N – число множеств. Пусть длина шаблона равна 7 кеш-линиям, обращения осуществляются к 5 из них, кеш-память содержит 4 множества. Схема отображения строк будет выглядеть следующим образом (рисунок 3.24).

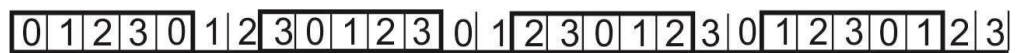


Рисунок 3.24 – Схема отображения строк основной памяти на множества кеш-памяти.

На рисунке для каждой строки указан номер множества кеш-памяти, в которое данная строка попадает. Видно, что заполнение множеств происходит неравномерно в отличие от непрерывного расположения данных. Однако в данном случае закономерность выявить достаточно просто, т.к. доступ регулярный. В ряде случаев это сделать затруднительно:

- сложная структура шаблона;
- нерегулярный доступ в память;
- размер шаблона превосходит размер анализируемого уровня кеш-памяти.

Предлагается метод определения максимальной длины блока данных, при которой генерируются только «холодные» промахи и не происходит вытеснения строк. Данный метод описывается выражением (3.73) и может быть применен при реализации любого шаблона доступа в память.

$$B_B: C_B \rightarrow C_M, \forall c \leq M, c \in C_M, \quad (3.73)$$

где B_B – максимальная длина блока, при которой генерируются только «холодные» промахи и не происходит вытеснения строк;

C_B - множество кешируемых элементов шаблона (паттерна);

C_M – множества кеш-памяти;

M – степень ассоциативности кеш-памяти.

Множества C_B , определенные на паттернах доступа, отображаются на множества C_M до тех пор, пока мощность любого c , принадлежащего C_M , не превышает степени ассоциативности кеш-памяти. Блок-схема алгоритма реализации описанного метода приведена на рисунке 3.25.

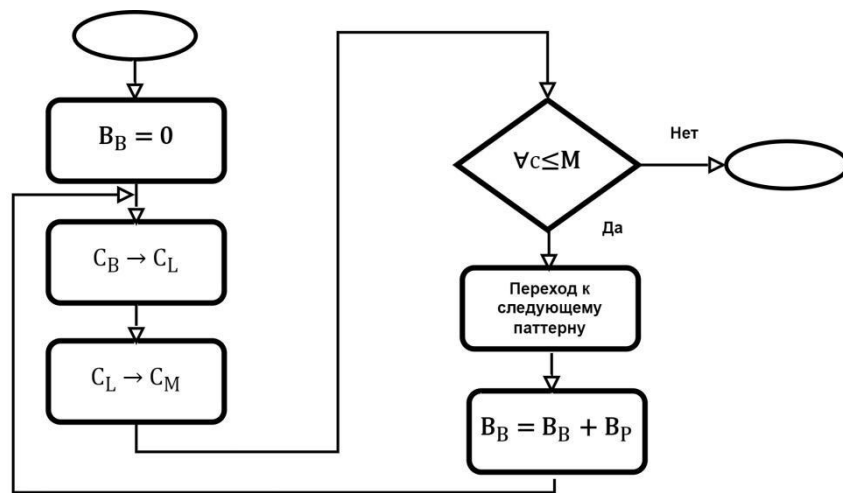


Рисунок 3.25 – Алгоритм определения максимальной длины блока

На рисунке 3.25 дополнительно используются следующие обозначения:

C_L - множество кеш-линий;

B_P – длина паттерна.

Блок-схема соответствует двум случаям применения метода: сложная структура шаблона и нерегулярный доступ в память. В случае большого размера шаблона данный алгоритм необходимо несколько видоизменить, сделав шагом итерации цикла не целый паттерн, а определенное количество строк паттерна, конкретное значение необходимо рассчитывать, исходя из характеристик кеш-памяти.

В случае регулярной структуры шаблона, размер которого не превосходит размера кеш-памяти, все количественные характеристики метрик ПС можно получить аналитическим способом, включая и максимальную длину блока данных.

Рассмотрим общий случай регулярного доступа к данным, которые не расположены в памяти непрерывно (рисунок 3.26).

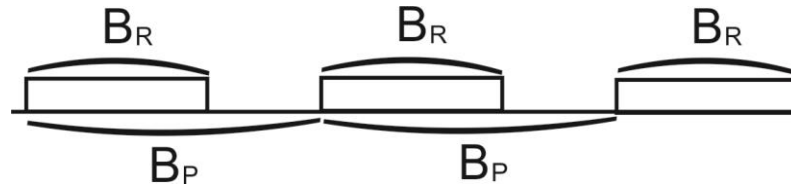


Рисунок 3.26 – Схема регулярного расположения данных

Данные расположены регулярно, но не непрерывно:

- длина каждого паттерна составляет B_P байт;
- из каждого паттерна считывается B_R байт.

Предварительно сформулируем и докажем несколько теорем.

Теорема о формировании рядов взаимно простых чисел (Теорема 3.1).

Если x и y – положительные взаимно простые числа, то числовой ряд, полученный как

$$L = (a * x) \% y, \quad a = \overline{0, y - 1} \quad (3.74)$$

будет содержать значения

$$L = \underline{\underline{\overline{0, y - 1}}} \quad (3.75)$$

и не содержать повторов.

Доказательство.

Используем доказательство от противного. Допустим, утверждение теоремы неверно, ряд L содержит другие значения. Значения элементов ряда также лежит в пределах от 0 до $(y-1)$, что очевидно (элементы формируются как остаток от деления на y), однако ряд содержит повторяющиеся значения. Тогда можно подобрать значения a, b, c, m, n , такие, что верно следующее

$$a * x = m * y + c, \quad (3.76)$$

$$b * x = n * y + c$$

и два элемента ряда будут равны c , т.к. появляются одинаковые значения остатков от деления $(a*x)\%y$ и $(b*x)\%y$ для разных a и b . Пусть $a > b$, вычтем из первого уравнения второе, получим

$$(a - b) * x = (m - n) * y. \quad (3.77)$$

В левой и правой части выражения (3.77) – общие кратные x и y . Но, т.к. они являются взаимно простыми числами, то их наименьшее общее кратное (НОК) равно $x*y$. Однако, по условию $a < y$, $b < y$, оба коэффициента являются положительными числами и, т.к. $a > b$, $(a-b)$ является положительным числом, меньшим y . Подобрать такие значения невозможно, следовательно, сделанное предположение неверно, а верно утверждение, сделанное в теореме.

Пример. Пусть $x = 3$, $y = 5$. Получим ряд $L = [0, 3, 1, 4, 2]$.

Пусть $x = 5$, $y = 3$. Получим ряд $L = [0, 2, 1]$.

Следствие 1 из теоремы 3.1.

Если x и y – положительные взаимно простые числа, то выводы теоремы 3.1 справедливы также для числового ряда

$$L = (a * x + b) \% y, \quad a = \overline{0, y-1}, \quad b = \overline{0, x-1}. \quad (3.78)$$

Например, пусть 2 и 3 – исходные взаимно простые числа. В соответствии с теоремой 1 получим числовой ряд

$$L = [(0 * 2) \% 3 = 0, (1 * 2) \% 3 = 2, (2 * 2) \% 3 = 1] \quad (3.79)$$

Пусть $b = 1$, тогда получим ряд

$$L = [(0 * 2 + 1) \% 3 = 1, (1 * 2 + 1) \% 3 = 0, (2 * 2 + 1) \% 3 = 2] \quad (3.80)$$

Пусть $b = 2$, тогда получим ряд

$$L = [(0 * 2 + 2) \% 3 = 2, (1 * 2 + 2) \% 3 = 1, (2 * 2 + 2) \% 3 = 0] \quad (3.81)$$

Если рассматривать полученные ряды как неупорядоченные множества, они идентичны.

Следствие 2 из теоремы 3.1.

Если x и y – положительные взаимно простые числа, то числовой ряд, полученный как

$$L = (a * x) \% y, \quad (3.82)$$

где a – ряд неотрицательных целых чисел $(0, 1, 2, 3, \dots)$, будет содержать конкатенацию рядов (3.75). Другими словами, в полученном ряду элементы, отстоящие друг от друга на y позиций, будут содержать одни и те же значения.

Данное утверждение можно доказать следующим образом:

$$\begin{aligned} [(a + y) * x] \% y &= (a * x + y * x) \% y = [(a * x) \% y + (y * x) \% y] \% y \quad (3.83) \\ &= [(a * x) \% y + 0] \% y = (a * x) \% y \end{aligned}$$

Аналогично можно доказать, что данное утверждение справедливо для рядов со смещением (3.78).

Пример. Пусть $x = 3$, $y = 5$. Получим ряд $L = [0, 3, 1, 4, 2, 0, 3, 1, 4, 2, \dots]$.

Пусть $x = 5$, $y = 3$. Получим ряд $L = [0, 2, 1, 0, 2, 1, \dots]$.

Пусть имеем ряд со смещением $x=5$, $y=3$, $b=1$. Получим ряд $L = [1, 0, 2, 1, 0, 2, \dots]$.

Следствие 3 из теоремы 3/1.

Максимальная длина ряда, сформированного с использованием (3.82) и не содержащего повторов, равна y .

Данное следствие является иначе сформулированным следствием 2 из теоремы 1. Действительно, ряд, сформированный в соответствии с условиями следствия 2 из теоремы 3.1, представляет собой конкатенацию рядов (3.75), содержащих уникальные значения. Элементы, отстоящие друг от друга на y позиций, будут содержать одни и те же значения. Поэтому максимальная длина ряда, не содержащего повторов, равна y .

Пример. Пусть $x = 3$, $y = 5$. Получим ряд $L = [0, 3, 1, 4, 2, 0, 3, 1, 4, 2, \dots]$. Максимальная длина ряда, не содержащего повторов, равна 5, т.е. равна y .

Теорема о заполняемости упорядоченных множеств, мощности которых являются взаимно простыми числами (Теорема 3.2).

Пусть L и M – положительные взаимно простые числа, есть последовательность чисел (список) A , формируемая как конкатенация последовательностей (списков) A' длины L со следующей структурой

$$A' = [0, \dots, L - 1] \quad (3.84)$$

Пусть также есть список B' длины L , содержащий одну единицу в позиции k , остальные нули. Пусть есть упорядоченное множество B мощности M , первоначально содержащее нулевые значения. Пусть осуществляется обход списка A , для каждого элемента вычисляется функция f , модифицирующая элементы B .

$$f: B(i \% M) = B(i \% M) + B'(i \% L), \quad (3.85)$$

где i – номер элемента A . Тогда существует конечное число S пройденных элементов A , после которых все элементы B будут содержать единицы.

Доказательство.

Пусть длина списка A равна НОК(L, M), т.е. $S=L*M$ в данном случае, тогда каждый элемент B модифицируется в соответствии с (3.85) L раз. В самом деле, в соответствии со следствием 3 из теоремы 3.1, максимальная длина ряда в списке A , сформированного с использованием (3.82) и не содержащего повторов, равна L . Значение i в выражении (3.85) принимает значения

$$i = \overline{0, L * M - 1} .. \quad (3.86)$$

При этом список индексов B в (3.85) представляет собой конкатенацию числовых рядов (3.75) для разных значений b , составленных следующим образом

$$i_B = (a * L + b) \% M, \quad a = \overline{0, M - 1}, \quad b = \overline{0, L - 1}.. \quad (3.87)$$

Здесь a – номер строки длиной M внутри числового ряда, b – номер элемента внутри строки, i_B – индекс внутри строки. Каждая такая строка в соответствии со следствием 3 из теоремы 3.1 содержит все индексы B и не содержит повторов. Число строк равно L по количеству элементов B' , следовательно, каждый элемент B модифицируется L раз.

Также можно показать, что в случае, если длина списка A равна $S=L*M$, к каждому элементу B в процессе модификации прибавляются значения всех элементов B' по одному разу. В самом деле, если принять в выражении (3.78) $x=M$, $y=L$, b – номер элемента B , получится числовой ряд длины L , не содержащий повторов, представляющий собой набор всех индексов в B' . Таким образом, к каждому элементу B (с номером b в (3.78)) в процессе модификации прибавляются все значения B' по одному разу. Т.к. только один элемент B' с индексом k содержит единицу, остальные содержат нули, после обхода $L*M$ элементов списка A все элементы B будут содержать единичные значения.

Проиллюстрируем теорему 3.2 на примере. Пусть $L=7$, $M=3$. Тогда при длине списка A , равной $7*3=21$, $k=0$ (номер элемента B' с единичным значением) получим следующие числовые ряды (рисунок 3/27).

A'	[	0	1	2	3	4	5	6	]														
i	[	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	]
A	[	0	<u>1</u>	2	<i>3</i>	<u>4</u>	5	6	<u>0</u>	1	<i>2</i>	<u>3</u>	4	5	<u>6</u>	0	1	<u>2</u>	3	<i>4</i>	<u>5</u>	6	]
B'	[	1	0	0	0	0	0	0	]														
B	[	1	1	1	]																		

Рисунок 3.27 – Числовые ряды, иллюстрирующие Теорему 3.2.

В множестве A элементы, использующиеся для модификации нулевого элемента B выделены жирным на сером фоне, использующиеся для модификации первого элемента B выделены подчеркиванием, использующиеся для модификации второго элемента B выделены курсивом. Видно, что каждый элемент модифицируется 7 раз, т.е. L раз. Значения списка A , выделенные для каждого элемента, представляют собой номера элементов B' , используемых при модификации B . Видно, что для каждого элемента B используются индексы от 0 до 6, т.е. все элементы B' по одному разу.

Следствие из теоремы 3.2..

Если в условии к теореме 3.2 длина списка равна $S=L*M$, список B' содержит единицы в v позициях, в остальных содержит нули, $v \leq L$, тогда после прохода A все элементы B будут содержать v .

Доказать данное утверждение можно, опираясь на доказательство теоремы 3.2. В процессе доказательства было показано, что к каждому элементу B в процессе модификации прибавляются значения всех элементов B' по одному разу. Т.е. после окончания обхода A в каждом элементе B будет накоплена сумма элементов B' , в данном случае равная v .

Проиллюстрируем данное следствие на примере. Пусть $L=7$, $M=3$, список B' содержит единицы в 3 позициях. Тогда при длине списка A , равной $7*3=21$, получим следующие числовые ряды (рисунок 3.28).

A'	[	0	1	2	3	4	5	6	]														
i	[	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	]
A	[	<u>0</u>	<u>1</u>	2	<u>3</u>	<u>4</u>	5	<u>6</u>	<u>0</u>	1	<u>2</u>	<u>3</u>	4	<u>5</u>	<u>6</u>	0	<u>1</u>	<u>2</u>	3	<u>4</u>	<u>5</u>	6	]
B'	[	1	1	1	0	0	0	0	]														
B	[	3	3	3	]																		

Рисунок 3.28 – Числовые ряды, иллюстрирующие Следствие из Теоремы 3.2.

Теорема о формировании рядов чисел, не являющихся взаимно-простыми (Теорема 3.3).

Если x и y не являются взаимно простыми числами, при этом являются положительными числами и $x=m*b$, $y=n*b$, m и n – взаимно простые числа тогда числовой ряд, полученный как

$$L = (a * x) \% y, \quad (3.88)$$

будет иметь длину n и содержать значения в интервале

$$L = \overline{0, y - 1} \quad (3.89)$$

и не содержать повторов при $a < n$.

Доказательство

Доказательство можно построить аналогично доказательству теоремы 3.1. Используем доказательство от противного. Допустим, утверждение теоремы неверно, ряд L содержит другие значения. Значения элементов ряда также лежат в пределах от 0 до $(y-1)$, что очевидно (элементы формируются как остаток от деления на y), однако ряд содержит повторяющиеся значения. Тогда можно подобрать значения a_1, a_2, w_1, w_2, c , такие, что верно

$$a_1 * x = w_1 * y + c, \quad (2.90)$$

$$a_2 * x = w_2 * y + c$$

и два элемента ряда будут равны c , т.к. появляются одинаковые значения остатков от деления $(a_1 * x) \% y$ и $(a_2 * x) \% y$ для разных a_1 и a_2 . Пусть $a_1 > a_2$, вычтем из первого уравнения второе, получим

$$(a_1 - a_2) * x = (w_1 - w_2) * y. \quad (3.91)$$

С учетом того, что $x = m * b$, $y = n * b$, получим

$$(a_1 - a_2) * m = (w_1 - w_2) * n. \quad (3.92)$$

В левой и правой части выражения (3.92) – общие кратные m и n . Но, т.к. они являются взаимно простыми числами, то их НОК равно $m * n$. Однако, по условию $a_1 < n$, $a_2 < n$, оба коэффициента являются положительными числами и, т.к. $a_1 > a_2$, $(a_1 - a_2)$ является положительным числом, меньшим n . Подобрать такие значения невозможно, следовательно, сделанное предположение неверно, а верно утверждение, сделанное в теореме.

Пример. Пусть $x = 4$ ($m = 2, b = 2$), $y = 6$ ($n = 3, b = 2$). Получим ряд $L = [0, 4, 2]$. Длина ряда равна 3, т.е. равно n .

Пусть $x = 6$ ($m = 3, b = 2$), $y = 4$ ($n = 2, b = 2$). Получим ряд $L = [0, 2]$. Длина ряда равна 2, т.е. равно n .

Из данной теоремы можно сформулировать следствия, аналогичные следствиям из теоремы 3.1.

Следствие 1 из теоремы 3.3.

Если x и y не являются взаимно простыми числами, при этом являются положительными числами и $x = m * b$, $y = n * b$, m и n – взаимно простые числа,

то выводы теоремы 3.3 справедливы также для числовых рядов с разным значением k

$$L = (a * x + k) \% y, \quad a = \overline{0, n-1}, \quad k = \overline{0, x-1}. \quad (3.93)$$

При этом в отличие от Следствия 1 из Теоремы 3.1, полученные ряды, представленные в виде неупорядоченных множеств, для разных значений k совпадать не будут. В самом деле, пусть $x=4$ ($m=2$, $b=2$) и $y=6$ ($n=3$, $b=2$) – исходные числа, причем $m=2$ и $n=3$ являются взаимно простыми. Тогда

$$\begin{aligned} & [((0 * 4 + 3) \% 6) = 3, ((1 * 4 + 3) \% 6) = 1, ((2 * 4 + 3) \% 6) = 5], \\ & k=3 \end{aligned} \quad (3.94)$$

$$\begin{aligned} & [((0 * 4 + 2) \% 6) = 2, ((1 * 4 + 2) \% 6) = 0, ((2 * 4 + 2) \% 6) = 4], \\ & k=2 \end{aligned}$$

При этом, количество различных полученных рядов меньше x , полученные ряды повторяются при k , формируемом в соответствии с (3.95) для разных v

$$k = v * b + w, \quad k < x, \quad w < b. \quad (3.95)$$

Пусть $w=0$, тогда для тех же значений x и y получим

$$[((0 * 4) \% 6) = 0, ((1 * 4) \% 6) = 4, ((2 * 4) \% 6) = 2], \quad v=0 \quad (3.96)$$

$$\begin{aligned} & [((0 * 4 + 2) \% 6) = 2, ((1 * 4 + 2) \% 6) = 0, ((2 * 4 + 2) \% 6) = 4], \\ & v=1 \end{aligned}$$

Если рассматривать полученные ряды как неупорядоченные множества, они идентичны $([0, 2, 4])$. Аналогично для $w=1$ получим

$$\begin{aligned} & [((0 * 4 + 1) \% 6) = 1, ((1 * 4 + 1) \% 6) = 5, ((2 * 4 + 1) \% 6) = 3], \\ & v=0 \end{aligned} \quad (3.97)$$

$$\begin{aligned} & [((0 * 4 + 3) \% 6) = 3, ((1 * 4 + 3) \% 6) = 1, ((2 * 4 + 3) \% 6) = 5], \\ & v=1 \end{aligned}$$

Полученные множества также идентичны $([1, 3, 5])$. Очевидно, что множества, полученные в соответствии с (3.96) и (3.97) не имеют общих элементов (их пересечение соответствует пустому множеству). При этом объединение этих множеств дает множество чисел от 0 до y .

Следствие 2 из теоремы 3.3.

Если x и y не являются взаимно простыми числами, при этом являются положительными числами и $x=m*b$, $y=n*b$, m и n – взаимно простые числа, то числовой ряд, полученный как

$$L = (a * x) \% y, \quad (3.98)$$

где a – ряд неотрицательных целых чисел $(0, 1, 2, 3, \dots)$, будет содержать конкатенацию рядов из теоремы 3.3. Другими словами, в полученном ряду элементы, отстоящие друг от друга на n позиций, будут содержать одни и те же значения.

Данное утверждение можно доказать следующим образом:

$$\begin{aligned} [(a + n) * x] \% y &= (a * x + n * x) \% y & (3.99) \\ &= [(a * x) \% (y) + (n * m * b) \% (n * b)] \% (y) \\ &= [(a * x) \% y + 0] \% y = (a * x) \% y \end{aligned}$$

Пример. Пусть $x = 4$ ($m = 2$, $b = 2$), $y = 6$ ($n = 3$, $b = 2$). Получим ряд $L=[0, 4, 2, 0, 4, 2, \dots]$.

Пусть $x = 6$ ($m = 3$, $b = 2$), $y = 4$ ($n = 2$, $b = 2$). Получим ряд $L=[0, 2, 0, 2, \dots]$.

Аналогично можно доказать, что данное утверждение справедливо для рядов со смещением для Следствия 1 из теоремы 3.3..

Следствие 3 из теоремы 3/3.

Максимальная длина ряда, сформированного с использованием (3.98) и не содержащего повторов, равна n .

Данное следствие является иначе сформулированным следствием 2 из теоремы 3.3. Действительно, ряд, сформированный в соответствии с условиями следствия 2 из теоремы 3.3, представляет собой конкатенацию рядов 3.88, содержащих уникальные значения. Элементы, отстоящие друг от друга на n позиций, будут содержать одни и те же значения. Поэтому максимальная длина ряда, не содержащего повторов, равна n .

Пример. Пусть $x = 4$ ($m = 2$, $b = 2$), $y = 6$ ($n = 3$, $b = 2$). Получим ряд $L=[0, 4, 2, 0, 4, 2, \dots]$. Максимальная длина без повторов равна 3 ($n=3$).

Пусть $x = 6$ ($m = 3, b = 2$), $y = 4$ ($n = 2, b = 2$). Получим ряд $L=[0, 2, 0, 2, \dots]$. Максимальная длина без повторов равна 2 ($n=2$).

Следствие 4 из теоремы 3.3.

Объединение множеств из Следствия 1 для теоремы 3.3 дает множество, элементы которого принадлежат ряду $0 < s < y$, мощность множества равна y , повторяющихся элементов нет. В примере выше для $x = 4, y = 6$ были получены два ряда

$$L=[0, 2, 4] \mid w=0, \quad (3.100)$$

$$L=[1, 3, 5] \mid w=1$$

Объединение этих рядов дает ряд

$$L=[0, 1, 2, 3, 4, 5], \quad (3.101)$$

элементы которого удовлетворяют сформулированному условию.

Теорема о заполнении числами упорядоченных множеств, мощности которых не являющимися взаимно-простыми (Теорема 3.4).

Пусть L и M не являются взаимно простыми числами, тогда сформировать равномерно заполненное одинаковыми значениями множество V в соответствии с условием теоремы 3.2 невозможно.

Доказательство.

В соответствии с Теоремой 3.3, если L и M не являются взаимно простыми числами, причем $L = m*b$, $M = n*b$, то числовой ряд, полученный в соответствии с условиями теоремы, не будет содержать повторов на длине, не превышающей n . Поэтому, длину списка A примем равной $S=L*n$, т.е. $\text{НОК}(L, n)$. Значение i в выражении 3.85 принимает значения

$$i = \overline{0, L * n - 1} .. \quad (3.102)$$

В данном случае ряды

$$i_B = (a * L + k) \% M, \quad a = \overline{0, n - 1}, \quad k = \overline{0, L - 1} .. \quad (3.103)$$

в соответствии со Следствием 1 из Теоремы 3.3 для разных k дадут несколько одинаковых групп рядов, число которых равно w в выражении для k . Т.к. $0 < w < b$, то и число групп рядов будет равно b . Общее число рядов равно k ($0 < k < L$), каждый ряд имеет длину n .

Пример. Пусть $L = 4$ ($m = 2, b = 2$), $M = 6$ ($n = 3, b = 2$). Получаем

$$\begin{aligned} k = 0 &- [0, 4, 2] \\ k = 1 &- [1, 5, 3] \\ k = 2 &- [2, 0, 4] \\ k = 3 &- [3, 1, 5] \end{aligned} \quad (3.104)$$

Т.к. в данном случае $b = 2$, получаем 2 группы рядов – $[0, 2, 4]$ и $[1, 3, 5]$. Другими словами, элементы множества B с индексами 0, 2 и 4 формируются суммой значений множества B' с индексами 0 и 2, а элементы множества B с индексами 1, 3 и 5 формируются суммой значений множества B' с индексами 1 и 3. Таким образом, если только один элемент множества B' установлен в 1, после выполнения процедуры, описанной в условии теоремы, только M/b , т.е. n элементов множества B будут установлены в 1.

Следствие 1 из Теоремы 3.4.

При заполнении множеств в соответствии с условиями Теоремы 3.4 в 1 будут установлены M/b (b – наибольший общий делитель $\text{НОД}(L, M)$) элементов множества B .

Проиллюстрируем данное утверждение на примере. Пусть $L=6, M=4$. Тогда при мощности множества A , равном $S=\text{НОК}(6,4)=12$, $k=0$, получим следующие числовые ряды (рисунок 3.29).

A'	[	0	1	2	3	4	5	]						
i	[	0	1	2	3	4	5	6	7	8	9	10	11	]
A	[	<u>0</u>	<u>1</u>	<u>2</u>	3	<u>4</u>	5	<u>0</u>	1	<u>2</u>	3	<u>4</u>	5	]
B'	[	1	0	0	0	0	0	]						
B	[	1	0	1	0	]								

Рисунок 3.29 – Числовые ряды, иллюстрирующие Следствие 1 из Теоремы 3.4.

Тогда элемент $B[0] = B'[0] + B'[4] + B'[2] = 1$, $B[1] = B'[1] + B'[5] + B'[3] = 0$, $B[2] = B'[2] + B'[0] + B'[4] = 1$, $B[3] = B'[3] + B'[1] + B'[5] = 0$. Т.е. два элемента равны единице, два равны нулю. В процессе формирования

Следствие 3 из Теоремы 3.4.

Соответствие элементов проецируемого B' и результирующего множеств B при заполнении множеств в соответствии с условиями Теоремы 3.4.

Как было показано выше, число групп рядов при заполнении результирующего множества B равно b , каждый ряд содержит n элементов.

На элементы групп результирующего множества оказывают влияние элементы проецируемого множества B' , которые также разделяются на b групп по m элементов.

Элементы i -ой группы проецируемого множества B' оказывают влияние на элементы i -ой группы результирующего множества.

Разделение на группы осуществляется следующим образом:

- множества делятся на подмножества S_i равной длины b (m подмножеств для множества B' и n подмножеств для множества B);
- элементы с равным смещением в каждом подмножестве попадают в одну группу.

Поясним сказанное на примере.

Пусть $L = 10$ ($m = 2, b = 5$), $M = 15$ ($n = 3, b = 5$).

Множества делятся на подмножества длины $b=5$. Тогда проецируемое множество B' будет разбито на $m=2$ подмножества, множество B будет разбито на $n=3$ подмножеств. В каждом подмножестве выделяем элементы с равными смещениями, помещаем их в одну группу, устанавливаем соответствие между группами.

Таблица 3.21. Соответствие групп проецируемого и результирующего множеств.

B'	B
[0, 5]	[0, 5, 10]
[1, 6]	[1, 6, 11]

Продолжение таблицы 3.21

В'	В
[2, 7]	[2, 7, 12]
[3, 8]	[3, 8, 13]
[4, 9]	[4, 9, 14]

К такому же выводу можно прийти, если рассуждать в контексте Следствия 1 из Теоремы 3.3. Построим числовые ряды

$$L = (a * x + k) \% y, \quad a = \underline{0, n-1}, \quad k = v * b + w, \quad k < x,$$

$w < b$.

$$k = 0 - [0, 10, 5]$$

$$k = 1 - [1, 11, 6]$$

$$k = 2 - [2, 12, 7[\quad (3.105)$$

$$k = 3 - [3, 13, 8]$$

$$k = 4 - [4, 14, 9]$$

$$k = 5 - [5, 0, 10]$$

$$k = 6 - [6, 1, 11[$$

$$k = 7 - [7, 2, 12]$$

$$k = 8 - [8, 3, 13[$$

$$k = 9 - [9, 4, 14]$$

Видно, что получено то же соответствие групп.

Рассмотрим пример заполняемости результирующего множества (рисунок 3.31)

A'	[	0	1	2	3	4	5	6	7	8	9	]																				
i	[	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	]
A	[	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	]
B'	[	1	0	1	0	0	1	0	0	0	0	]																				
B	[	2	0	1	0	0	2	0	1	0	0	2	0	1	0	0	]															

Рисунок 3.31 – Числовые ряды, иллюстрирующие Следствие 3 из Теоремы 3.4.

В проецируемом множестве B' единицы установлены в позициях 0, 2 и 5. Позиция 2 принадлежит группе $[2, 7]$, которая соответствует группе $[2, 7, 12]$ результирующего множества B . Соответственно, каждый элемент этой группы устанавливается в 1. Позиции 0 и 5 принадлежат одной группе $[0, 5]$, которая соответствует группе $[0, 5, 10]$ результирующего множества. Каждая из 1 приводит к инкременту элемента результирующего множества, поэтому соответствующие элементы результирующего множества равны 2.

Следствие 4 из Теоремы 4/4.

Каждая из единиц в группе проецируемого множества приводит к инкременту всех элементов соответствующей группы результирующего множества.

Иллюстрация данного утверждения приведена выше. Единицы в позициях 0 и 5 принадлежат одной группе проецируемого множества $[0, 5]$, что привело к двум операциям инкремента для каждого элемента соответствующей группы результирующего множества $[0, 5, 10]$.

Следствие 5 из Теоремы 4/4.

Для равномерного заполнения единицами множества B в соответствии с условиями теоремы 3.4, необходимо, чтобы в единицу были установлены все элементы одного из подмножеств S_i проецируемого множества.

В самом деле, установка одного из элементов подмножества в 1 приведет к установке в 1 всех элементов соответствующей группы результирующего множества. Установка всех элементов подмножества в 1

приведет к установке в 1 всех элементов соответствующих групп результирующего множества. В соответствии со Следствием 4 из Теоремы 3.3 объединение всех групп результирующего множества дает множество всех индексов В, таким образом, все элементы множества В установлены в 1.

Следствие 6 из Теоремы 3.4.

Для равномерного заполнения множества В в соответствии с условиями теоремы 3.4, необходимо, чтобы в единицу были установлены все элементы одного или более подмножеств S_i проецируемого множества. Значения элементов множества В будут при этом равны числу заполненных подмножеств проецируемого множества В'.

Доказательство данного утверждения вытекает из доказательства предыдущего (Следствие 5).

Теоремы, доказанные выше, а также следствия из данных теорем позволяют аналитическим путем определить степень заполнения множественно-ассоциативной кеш-памяти при доступе к данным, расположенным в памяти в соответствии с регулярным паттерном доступа, однако, при этом данные считываются не непрерывно, в паттернах присутствуют данные, к которым обращение не осуществляется.

Интерпретация параметров, использованных в формулировках теорем 3.1 – 3.4 для подсистемы памяти ВС. Интерпретация теоремы 3.1 и 3.3.

Y – число множеств в множественно-ассоциативном кеше;

X – длина шаблонов (шаг в памяти в элементах кеш-линий).

Например, 128 множеств. Шаг 7. Заполняться будут: 0 множество, 7, 14, и т.д. Т.к. числа взаимно простые, заполняться будут равномерно все множества.

Интерпретация теоремы 3.2 и 3.4.

L – длина блоков (шаг в памяти в элементах кеш-линий);

M – число множеств в множественно-ассоциативном кеше;

B' – множество, определяющее кешируемые элементы шаблонов;

B – множества кеша: набор значений, каждое из которых определяет количество заполненных банков в соответствующем множестве;

A – память.

Предлагаемый метод реализуется для определения количественных характеристик метрик использования подсистемы памяти, в частности вероятности кеш-промаха при обработке блока данных, например, массива. Соответственно, основным в данном случае является вопрос: «Какова вероятность кеш-попаданий P_{ch} при обработке массива заданного размера N_B ?». Как было показано выше, данный вопрос тесно связан с вопросом: «Каков максимальный размер N_B , при котором генерируются только «холодные» промахи и не происходит вытеснения строк?». Приведенные выше теоремы позволяют ответить на второй вопрос (соответственно и на первый) для регулярных паттернов доступа к данным. На рисунке 3.32 представлен оптимизированный алгоритм определения максимальной длины блока, при которой не происходит вытеснения кеш-строк, разработанный на основе приведенных теорем и следствий из них.

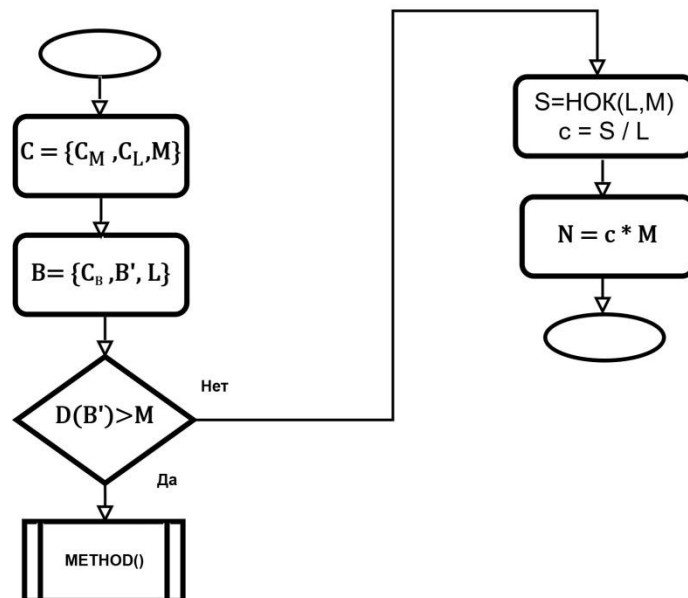


Рисунок 3.32 - Оптимизированный алгоритм определения максимальной длины блока, при которой не происходит вытеснения кеш-строк

Для определения максимального размера блока, не вызывающего кеш-промахов, необходимо:

- определить характеристики анализируемого уровня кеш-памяти (в частности, определить количество множеств в каждом банке M);
- определить характеристики шаблона доступа к памяти (длину шаблона в кеш-линиях L , количество кеш-линий в шаблоне, к которым производится обращение – количество единиц в множестве B');
- в случае, если количество заполненных элементов в множестве B' превышает степень ассоциативности анализируемого уровня кеш-памяти, реализовать итерационный вариант метода (рисунок 3.25);
- в противном случае определить $S = \text{НОК}(L, M)$;
- с учетом степени ассоциативности определить максимальный размер блока.

3.3.1 Экспериментальная проверка метода количественной оценки метрик использования подсистемы памяти

Был проведен вычислительный эксперимент, который подтверждает сделанные ранее выводы. Эксперимент проводился на вычислительной системе, в состав которой входит МП 11th Gen Intel(R) Core(TM) i9-11900 @ 2.50GHz/

Для проведения вычислительного эксперимента использовался язык программирования C, тип данных double (размер переменной составляет 8 байт).

Были выбраны разные длины шаблона, проведен аналитический расчет максимальной длины блока, далее проведен вычислительный эксперимент для проверки результатов. Количество множеств в банках кеш-памяти МП составляет 64 в L1D и 1024 в L2, менялось значение длины блока L , в опыте использовалась одна кеш-строка из всего шаблона (рисунок 3.33).

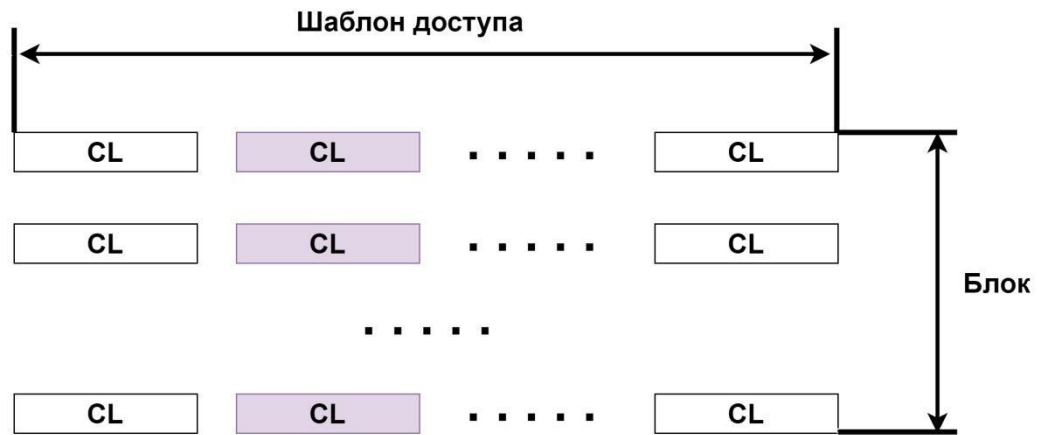


Рисунок 3.33 – Структура блока данных

Для этого при выделении памяти под массив учитываем выравнивание, также учитываем, что в одну строку кеша помещается 8 элементов double.

В таблице 3.22 приведены характеристики двух шаблонов доступа в память, рассчитанные для кеша L2 в соответствии с предложенным методом.

Таблица 3.22 – Характеристики шаблонов доступа в память

L, cl (d / b)	S=НОК(L,M)	c=S / L	N	S _В (d / b)	%
1024 (8192 / 64 кБ)	1024	1	8	65536 / 512Кб	0,098
512 (4096 / 32кБ)	1024	2	16	65536 / 512Кб	0,196

В первом столбце приведен размер шаблонов в кеш-линиях (cl), а также в элементах double (d) и байтах (b). Во втором столбце приводится значение наименьшего общего кратного для значений L и M (L2), которое в дальнейшем используется для расчета максимальной длины блоков. Длины шаблонов выбирались таким образом, чтобы заполнялось только одно множество в кеш-памяти каждого уровня. В третьем столбце приводится количество шаблонов, которые будут использоваться для размещения данных в одном банке L2. В четвертом столбце приводится количество шаблонов, непрерывно расположенных в памяти, которые помещаются в кеш-память L2, не вызывая промахов. В пятом столбце приводится общий размер блока в элементах double и байтах, В последнем столбце приводится

процент заполнения кеш-памяти при использовании данного шаблона доступа до начала вытеснения строк.

Необходимо пояснить, как заполнялась таблица. Ранее приводились характеристики подсистемы памяти МП, на котором выполнялся вычислительный эксперимент:

- кеш L1D имеет объем 48 кБ и степень ассоциативности 12;
- кеш L2 имеет объем 512 кБ и степень ассоциативности 8;
- длина кеш-линии составляет 64 байта.

Реализуем метод определения максимального размера блока, не вызывающего кеш-промахов в соответствии с оптимизированным алгоритмом, представленным на рисунке 3.32.

На первом этапе определяем характеристики анализируемого уровня кеш-памяти (в частности, необходимо определить количество множеств в каждом банке M).

$$M_{L1D} = 48 * 1024 / 64 / 12 = 64 \quad (3.106)$$

$$M_{L2} = 512 * 1024 / 64 / 8 = 1024$$

Далее определяем характеристики шаблона доступа к памяти (длину шаблона в кеш-линиях L, количество кеш-линий в шаблоне, к которым производится обращение – количество единиц в множестве B').

Длину шаблона выбираем равной 1024 кеш-линиям. Это приведет к заполнению только одного множества кеш-памяти в L2, т.к. число множеств там равно 1024. Однако это также приведет к заполнению только одного множества в L1D, т.к. данное значение кратно числу множеств в L1D. При расчете длины шаблона в элементах double и байтах учитывается длина кеш-линии (64 байта) и размер типа double (8 байтов).

Далее определяем $S = \text{HOK}(L, M)$ и количество шаблонов, помещающихся в S (с).

На последнем этапе определить максимальный размер блока с учетом степени ассоциативности как произведение количества шаблонов, которые размещаются в одном банке L2 и степени ассоциативности. Определяется

количество шаблонов, непрерывно расположенных в памяти, которые помещаются в кеш-память, не вызывая промахов. Т.к. анализируем уровень L2, учитываем его степень ассоциативности, равный 8. Далее рассчитываем общую длину блока данных, а также долю использованных данных в шаблоне (одна кеш-строка из 1024 и из 512).

В результате были получены результаты, представленные на рисунках 3.34 и 3.35. На рисунке 3.34 представлен результат моделирования для $L=8192d$, на рисунке 3.35 – $L=4096d$.

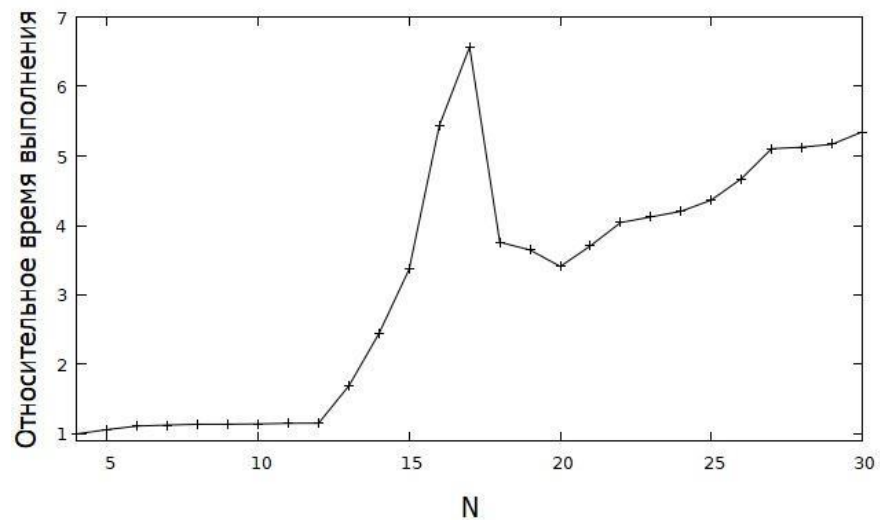


Рисунок 3.34 – Результат моделирования для $L=8192d$,

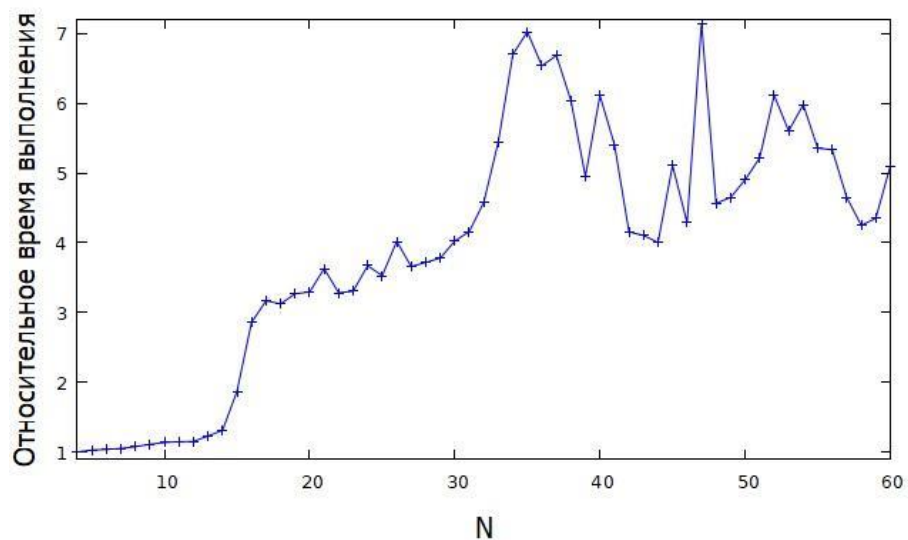


Рисунок 3.35 - Результат моделирования для $L=4096d$,

Измерения осуществлялись следующим образом. На первом этапе осуществлялся многократный доступ к элементам массива, размер которого

совпадал с размером блока, состоящим из N шаблонов. Далее для каждого измерения вычислялось среднее время обработки одного шаблона. На последнем этапе для каждого размера блока вычислялось относительное время выполнения, которое определялось как отношение среднего времени обработки шаблона к среднему времени обработки шаблона при $N=4$, с этого значения осуществлялось построение графика. При значении N , меньшем четырех, результаты оказываются зашумленными в связи со снижением вычислительной нагрузки и увеличении влияния длительности вспомогательных операций, например, операторов цикла.

Относительно полученных результатов можно сделать вывод, что они подтверждают сделанные аналитические оценки, при этом стоит отметить следующее. Промахи в L2 оказывают гораздо большее влияние на время выполнения программы, чем промахи в L1. Для кеша L2 промахи начинаются, начиная с N , приведенного в таблице 3.22, достигая максимума при размере блока, равным двойному размеру кеш-памяти ($N = 16$ на рисунке 3.34 и $N = 32$ на рисунке 3.35). Особенно хорошо данный эффект заметен на рисунке 3.35. Т.к. используемые длины шаблонов подбирались таким образом, чтобы демонстрировать промахи в кешах обоих уровней, на рисунках можно наблюдать и промахи L1 при $N=12$ (степень ассоциативности кеша L1D). На рисунке 3.34 эффект более заметен, т.к. на рисунке 3.35 замедление при $N=12$ переходит в более значимое замедление при $N=16$, вызванное промахами в L2.

Третий эксперимент связан с анализом шаблонов, размер которых является взаимно простым числом с количеством множеств кеш-памяти (1024 в L2). Было выбрано значение 17 кеш-строк (136 в элементах double). Результат эксперимента приведен на рисунке 3.36.

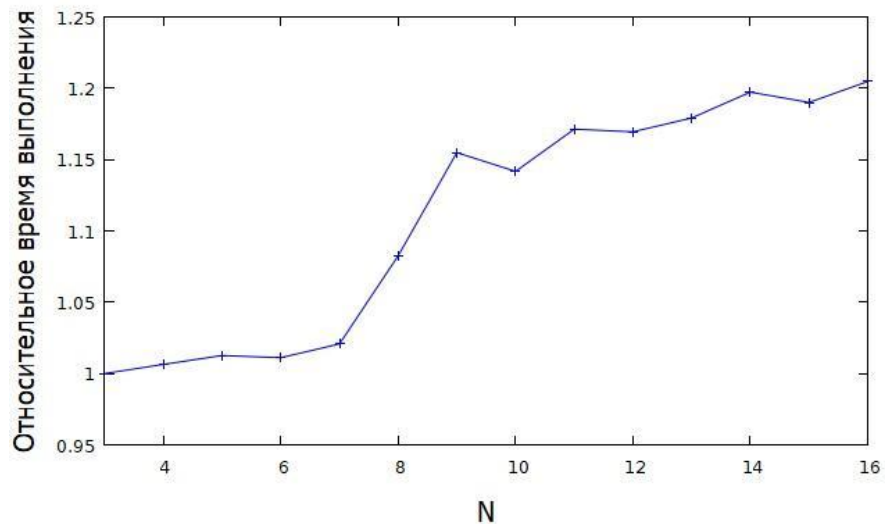


Рисунок 3.36 – Результат эксперимента для размера шаблона, взаимно простого с размером множества L2

Эксперимент проводился следующим образом. Фиксировался размер шаблона (17 кеш-линий или 136 double). Далее варьировалось заполнение шаблона, параметр N на графике соответствует текущему заполнению (от 1 до 16 кеш-линий). Относительное время выполнения считалось аналогично графикам на рисунках 3.34 и 3.35. В соответствии со следствиями из приведенных теорем, при подобном размере шаблона кеш-промахи не будут генерироваться при заполнении, не превышающем степень ассоциативности кеш-памяти. На графике видно, что рост времени выполнения наблюдается при $N=8$, что соответствует степени ассоциативности L2. Результат эксперимента также полностью подтверждает сделанные ранее выводы.

Зависимости, представленные на графиках, в большей степени отражают процессы, происходящие в кеш-памяти L2, чем в L1D. Это связано с большей разницей в скорости работы между L3 и L2, чем между L2 и L1D, таким образом, промахи в L2 приводят к большим временным задержкам, чем промахи в L1.

Кроме того, в процессе анализа использовался регулярный доступ к памяти: шаблоны обрабатывались последовательно друг за другом, внутри шаблона данные также обрабатывались последовательно, Это приводило к тому, что механизм предвыборки данных, существующий во всех

современных МП, сглаживал проблемы, связанные с кеш-промахами (например, на рисунках 3.34 и 3.35 не наблюдается резкое увеличение времени обработки при генерации кеш-промахов в L2 при N=8, хотя увеличение времени обработки имеет место). При другом порядке обращения к элементам блоков, например, при случайном порядке обработки, эффекты были бы более акцентированными. Однако был выбран регулярный порядок обработки, так как именно он используется в большинстве реальных задач.

3.4 Имитационная модель исполнения программы в подсистеме памяти

Для определения характеристик ПС на конкретной ЦВА, а также характеристик подсистемы памяти для проектируемой ПС предложена имитационная модель исполнения программы (ИМИП) в ГВС с иерархической структурой памяти (рисунок 3.37).

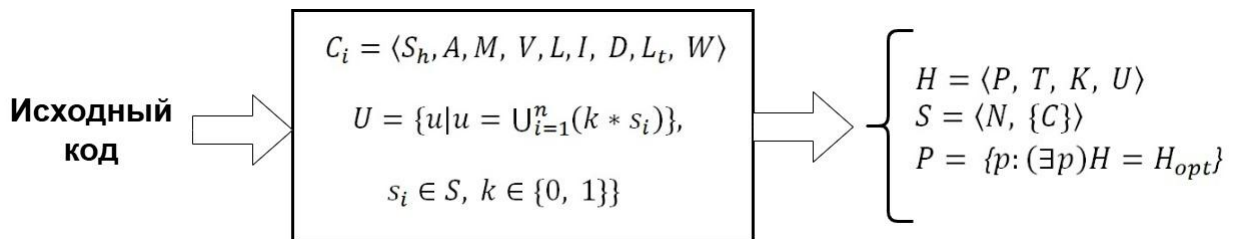


Рисунок 3.37 - Имитационная модель исполнения программы в ГВС с иерархической структурой памяти

ИМИП представляет собой детерминированную модель, описывающую ГВС с иерархической структурой памяти, содержащей N уровней, при этом каждый уровень реализован в виде кеш-памяти, которая описывается кортежем

$$C = \langle S_h, A, S, L_s, I, D, L, W_t, W_a \rangle \quad (3.107)$$

где S_h - разделяемость (0/1) (используется только в многопоточной модели);

A - ассоциативность (1, 2, 4...);

S - размер;

L_s - длина кеш-линии;

I - инклюзивный / эксклюзивный (0/1);

D - алгоритм вытеснения (LRU, pseudoLRU...);

L - латентность (используется для оценки времени выполнения программы, не применяется для анализа вероятности кеш-промахов);

W_t – политика записи: write-through (при изменении данных в кеше эти данные немедленно и одновременно записываются как в кеш, так и в основную память) и write-back (при изменении данных в кеше эти данные сначала записываются только в кеш, запись в основную память осуществляется только при вытеснении строки);

W_a – поведение при промахе по записи: write-allocate (при промахе по записи, блок памяти, содержащий целевой адрес, сначала загружается в кеш, после чего операция записи осуществляется в кеше) и write-around (при промахе по записи операция записи осуществляется напрямую в следующий уровень иерархии памяти, содержащий блок памяти с целевым адресом).

В ИМИП определены операции двух типов (рисунок 3.38):

- пользовательские операции (user defined operations), описываются множеством U ;
- системные операции (system defined operations), описываются множеством S .

Пользовательские операции – операции, непосредственно инициируемые кодом программы (на рисунке 3.38 обозначены сплошными стрелками):

- чтение из памяти с кешированием;
- чтение из памяти без кеширования;
- запись в память с кешированием;
- запись в память без кеширования;
- предвыборка в кеш L_i ;
- предвыборка без кеширования.

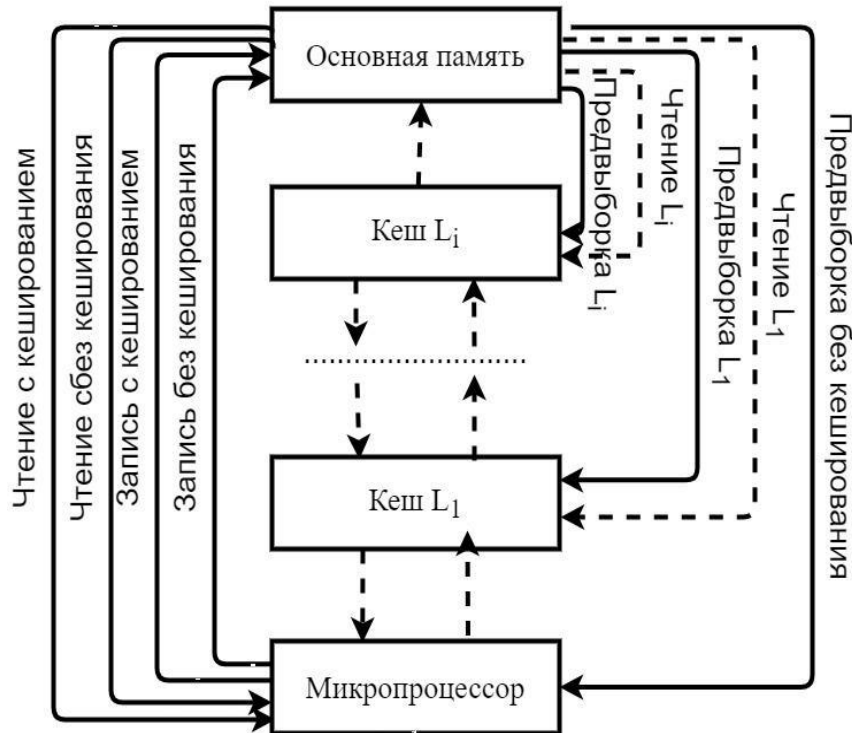


Рисунок 3.38 - Модель исполнения программы в подсистеме памяти вычислительной системе с общей памятью

Системные операции (таблица 3.23) представляют собой операции, напрямую пользователем не инициируемые, однако выполняемые ВС для реализации пользовательских операций (на рисунке 3.38 обозначены штриховыми стрелками).

Таблица 3.23 - Атомарные системные операции

Обозначение	Описание
$\text{data}[I][\text{tag}] = d$	запись в строку кеш-памяти уровня I с тегом tag данных d ;
$d = \text{data}[I][\text{tag}]$	считывание строки кеш-памяти уровня I с тегом tag
$\text{reg} = \text{data}[1][\text{tag}]$	считывание данных из $L1$ в регистр МП
$\text{data}[1][\text{tag}] = \text{reg}$	запись содержимого регистра МП в $L1$
$d = \text{mem}[\text{tag}]$	считывание строки, идентифицируемой тегом tag , из памяти
$\text{mem}[\text{tag}] = d$	запись строки, идентифицируемой тегом tag , в память
$c = \text{HIT}(I, \text{tag})$	проверка, находится ли строка с тегом tag в уровне кеш-памяти I

Продолжение таблицы 3.23

Обозначение	Описание
$\text{tag} = \text{ALG}(I)$	определение тега вытесняемой строки с уровня I в соответствии с используемым алгоритмом замещения строк

Пользовательские операции представляют собой суперпозицию системных операций.

$$U = \{u | u = \bigcup_{i=1}^n (k * s_i), s_i \in S, k \in \{0, 1\}\} \quad (3.108)$$

Другими словами, элементы множества пользовательских операций U формируются в виде суперпозиции элементов множества S системных операций. Для реализации одной пользовательской операции необходимо от одной и более системных операций, выполняемых в определенной последовательности.

Операция суперпозиции (3.108) формирует последовательность, в которой порядок следования элементов имеет значение, при этом индекс i представляет собой номер в данной последовательности.

На основе системных операций при имитационном моделировании формируется множество пользовательских операций. Ниже приведен псевдокод операции считывания данных из памяти.

```

read (tag, N){
    curI = 1;
    while ((! HIT(curI, tag)) && (curI <= N)) curI++;
    if (curI <= N) return read_cache(tag, curI, N);
    else{ return read_mem(tag, N);
}

```

При моделировании исполнения программы осуществляется поиск строки в кеш-памяти, начиная с уровня $L1$, и ее считывание `read_cache()`. Если строка отсутствует в кеш-памяти, она считывается из основной памяти. Реализация системных операций считывания из заданного уровня кеш-памяти (`read_cache()`) и считывания из основной памяти (`read_mem()`) зависит от способа организации кеш-памяти. Ниже приведен псевдокод реализации в

ИМИП системных операций для кеш-памяти с инклюзивной организацией (таблица 3.24).

Таблица 3.24 – Реализация системных операции считывания данных для кеш-памяти с инклюзивной организацией

Считывание из кеш-памяти	Считывание из памяти
<pre>read_cache(tag, I, N){ curI = 1; tag_pushed = ALG(1); while (curI < I){ data[curI][tag_pushed] = (data[I][tag]); curI++;} return data[1][tag]; }</pre>	<pre>read_mem(tag, N){ tag_pushed = ALG(1); t = mem[tag] curI = 1; while (curI < N){ data[curI][tag_pushed] = t; curI++;} return t; }</pre>

На вход ИМИП подается исходный код проектируемой ПС, который преобразуется в последовательность пользовательских операций (3.108), выполнение которых моделируется в ГВС с характеристиками (3.107). В результате формируются выходные параметры. ИМИП применяется в случае невозможности получения значений метрик ПС аналитическим способом.

Выходные параметры модели.

1. Характеристики ПС на ЦВА.

$$H = \langle M, T, K, P \rangle \quad (3.109)$$

где M – вероятность кеш-промахов для кеш-памяти различных уровней;

T – оценка времени выполнения программы;

K – коэффициент повторного использования данных для кеш-памяти различных уровней;

P – процент неэффективных обращений к памяти (штрафы).

Первые два параметра совпадают с соответствующими метриками ПС. Третий и четвертый параметр дополняют их для комплексной оценки проектируемой ПС:

- коэффициент повторного использования данных тесно связан с вероятностью кеш-промаха; при этом можно формировать различные

варианты данного параметра: в целом для ПС, отдельно для массивов, обрабатываемых в анализируемом коде;

- процент неоптимальных обращений к памяти тесно связан с метрикой, вычисляемой на основе числа невыровненных обращений к памяти; можно формировать различные варианты данного параметра; штраф за невыровненный переход между кеш-линиями, штраф за невыровненный переход между страницами памяти, штраф за использование выровненных операций при обращении к невыровненным участкам памяти и другие штрафы, которые могут зависеть от архитектуры МП; совместно с параметром латентности позволяет оценивать увеличение времени работы программы из-за наличия штрафов и принятия решений о необходимости новой итерации в рамках модели улучшения показателей эффективности ПС.

Данный набор выходных параметров необходим для оценки эффективности ПС на конкретной ЦВА.

2. Определение аппаратурной конфигурации подсистемы памяти для проектируемой программы

$$S = \langle N, \{C\} \rangle \quad (3.110)$$

где N – число уровней кеш-памяти;

$\{C\}$ – параметры кеш-памяти различных уровней.

3. Определение параметров алгоритмов.

$$P = \langle p: (\exists p) H = H_{opt} \rangle \quad (3.111)$$

где P – множество параметров.

Таким образом, задача определения параметров алгоритмов сводится к поиску набора параметров p , таких, что характеристики ПС (3.109) принимают оптимальные значения:

- M , T и P принимают минимально возможные значения;
- K принимает максимально возможное значение.

В ИМИП используется единое виртуальное адресное пространство. Переменные и массивы размещаются в нём последовательно с учётом выравнивания и выбранной гранулярности. Соответственно, кеш-промахи в

ИМИП определяются по виртуальным адресам, тогда как в реальной ГВС виртуальные адреса транслируются в физические с помощью страничной адресации, и промахи кеша формируются уже относительно физических адресов. Оценим эквивалентность оценок кеш-промахов в ИМИП и реальных вычислительных системах. Введем основные определения.

Рассмотрим вычислительную систему, включающую процессор с множественно-ассоциативной кеш-памятью и основную память, организованную постранично. В памяти размещается непрерывный массив данных (линейный буфер), к элементам которого производится доступ в ходе выполнения некоторой вычислительной процедуры (программы).

1. Пусть $\mathcal{A} \subset \mathbb{N}_0$ — множество всех возможных адресов памяти в рассматриваемой системе.

2. Параметры подсистемы памяти задаются константами:

- $L \in \mathbb{N}$ — размер кеш-строки в байтах;
- $S \in \mathbb{N}$ — число множеств кеша;
- $P \in \mathbb{N}$ — размер страницы виртуальной памяти в байтах.

3. Порядок обращений к элементам массива задается детерминированной последовательностью смещений (*трассой*):

$$\mathcal{T} = \langle \tau_1, \tau_2, \dots, \tau_m \rangle, \quad \tau_k \in \mathbb{N}_0 \quad (3.112)$$

где τ_k есть смещение k -го обращения относительно начала массива. Данная последовательность рассматривается как инвариантная относительно физического размещения массива в памяти.

4. Пусть $A^{model} \in \mathcal{A}$ и $A^{real} \in \mathcal{A}$ — базовые адреса начала массива соответственно в имитационной модели и в реальной системе. Абсолютные адреса k -го обращения в этих двух случаях обозначим через $a_k^{model} \in \mathcal{A}$ и $a_k^{real} \in \mathcal{A}$.

В имитационной модели массив располагается в непрерывной области памяти, поэтому:

$$a_k^{model} = A^{model} + \tau_k. \quad (3.113)$$

В реальной системе адрес $A^{real} + \tau_k$ рассматривается как виртуальный, а a_k^{real} — как соответствующий ему физический адрес. Предполагается, что внутристраничное смещение сохраняется:

$$a_k^{real} \equiv A^{real} + \tau_k \pmod{P}. \quad (3.114)$$

Таким образом, далее под адресами a_k^{model} понимаются адреса в абстрактном линейном адресном пространстве имитационной модели, тогда как a_k^{real} трактуются как физические адреса реальной системы, по которым определяется отображение адресов на строки и множества кеша.

Для страниц памяти, занятых рассматриваемым массивом, отображение идентификаторов страниц в модели на идентификаторы физических страниц реальной системы является корректно определённым и инъективным. То есть, существует инъективная функция $\pi: \mathbb{Z} \rightarrow \mathbb{Z}$ такая, что:

$$\forall k \in \{1, \dots, m\} \quad \left\lfloor \frac{a_k^{real}}{P} \right\rfloor = \pi \left(\left\lfloor \frac{a_k^{model}}{P} \right\rfloor \right). \quad (3.115)$$

5. Определим идентификатор блока памяти, отображающийся в одну кеш-строку:

$$B(a) = \left\lfloor \frac{a}{L} \right\rfloor, \quad (3.116)$$

и тег строки кеш-памяти:

$$\text{Tag}(a) = \left\lfloor \frac{B(a)}{S} \right\rfloor. \quad (3.117)$$

6. Пусть $\mathcal{M}_s(A, \mathcal{T})$ — число кеш-промахов, возникающих в s -м множестве кеша при выполнении трассы $\mathcal{T}$ с базовым адресом A .

7. Пусть $\mathcal{M}(A, \mathcal{T})$ — общее число кеш-промахов при последовательной обработке трассы $\mathcal{T}$ в системе, в которой массив размещен по базовому адресу A .

Для дальнейшего изложения принимаются следующие предположения:

1. При исполнении программы в реальной системе отсутствуют внешние воздействия на состояние кеш-памяти со стороны посторонних вычислений; не учитываются также эффекты прерываний, DMA и иных

механизмов, изменяющих содержимое кеша. Тем самым, эволюция состояния кеша определяется только трассой $\mathcal{T}$.

2. Кеш-память рассматривается как детерминированная система с фиксированным алгоритмом замещения; не учитываются аппаратная предвыборка, эффекты TLB и спекулятивное исполнение. Предполагается, что теги используются только при проверке попаданий / промахов (сравнении на равенство / неравенство), а политика замещения не зависит от их численных значений.

3. Перед началом обработки трассы кеш находится в одинаковом состоянии в ИМИП и в реальной системе (например, пуст).

Теорема об эквивалентности количества кеш-промахов (Теорема 3.5).

Пусть при сделанных выше модельных предположениях для рассматриваемой вычислительной системы и моделируемой задачи выполнены следующие условия:

1. Все биты адреса, используемые для вычисления индекса множества кеша, целиком принадлежат полю смещения внутри страницы: $S \cdot L \leq P$. В силу этого, индекс множества инвариантен относительно номера страницы и зависит только от смещения внутри неё:

$$J(a) = \left\lfloor \frac{a \bmod P}{L} \right\rfloor \bmod S. \quad (3.118)$$

2. Число строк кеша, полностью помещающихся в одну страницу, является кратным числу его множеств:

$$P \equiv 0 \pmod{S \cdot L}. \quad (3.119)$$

3. Базовые адреса обрабатываемого массива данных в ИМИП и реальной системе имеют одинаковое смещение относительно начала кеш-строки:

$$A^{model} \equiv A^{real} \pmod{L}. \quad (3.120)$$

4. Отображение виртуальных страниц, занятых рассматриваемым массивом, в физические страницы является инъективным (две различные

виртуальные страницы массива не отображаются в одну и ту же физическую страницу).

Тогда для любой трассы обращений $\mathcal{T}$ к элементам массива число кеш-промахов в ИМИП совпадает с числом кеш-промахов в реальной системе:

$$\mathcal{M}(A^{model}, \mathcal{T}) = \mathcal{M}(A^{real}, \mathcal{T}). \quad (3.121)$$

Доказательство. Из (3.120) следует, что $\exists K \in \mathbb{Z}: A^{real} - A^{model} = K \cdot L$. Из (3.119) следует, что $\exists N_{lines} \in \mathbb{N}: P = N_{lines} \cdot L$, причём N_{lines} кратно S . В дальнейшем N_{lines} интерпретируется как количество кеш-строк в странице памяти.

По теореме о делении с остатком представим K в виде:

$$K = q \cdot N_{lines} + \sigma. \quad (3.122)$$

где $q \in \mathbb{Z}$, а $\sigma \in \{0, 1, \dots, N_{lines} - 1\}$. Тогда:

$$\begin{aligned} A^{real} - A^{model} &= (q \cdot N_{lines} + \sigma) \cdot L = q \cdot (N_{lines}L) + \sigma L \\ &= q \cdot P + \sigma \cdot L. \end{aligned} \quad (3.123)$$

Отсюда:

$$A^{real} = A^{model} + q \cdot P + \sigma \cdot L. \quad (3.124)$$

Переходя к сравнению выражения по модулю P , получаем:

$$A^{real} \equiv A^{model} + \sigma L \pmod{P}. \quad (3.125)$$

Из (3.113), (3.114) и (3.125) следует, что для $\forall i \in \{1, \dots, m\}$ справедливо:

$$a_i^{real} \equiv a_i^{model} + \sigma L \pmod{P}. \quad (3.126)$$

Поскольку $P = N_{lines}L$, и N_{lines} кратно S , то согласно определению (3.118) и соотношению (3.126):

$$\begin{aligned} \mathcal{J}(a_i^{real}) &= \left\lfloor \frac{a_i^{real} \bmod P}{L} \right\rfloor \bmod S = \left\lfloor \frac{(a_i^{model} + \sigma L) \bmod P}{L} \right\rfloor \bmod S = \\ &= \left(\left\lfloor \frac{a_i^{model} \bmod P}{L} \right\rfloor + \sigma \right) \bmod N_{lines} \bmod S = \\ &= (\mathcal{J}(a_i^{model}) + \sigma) \bmod S. \end{aligned} \quad (3.127)$$

Кеш-память является множественно-ассоциативной, причём состояние каждого множества изменяется независимо от остальных. Поэтому обработку

общей трассы $\mathcal{T}$ можно рассматривать как независимую обработку S подпоследовательностей, соответствующих множествам кеша. Для каждого $k \in \{0, \dots, S-1\}$ определим множества номеров обращений к памяти, которые отображаются в k -е множество кеша, отдельно для ИМИП и для реальной системы:

$$\begin{aligned} T_k^{model} &:= \{i \in \{1, \dots, m\} : J(a_i^{model}) = k\}, \\ T_k^{real} &:= \{i \in \{1, \dots, m\} : J(a_i^{real}) = k\}. \end{aligned} \quad (3.128)$$

Из (3.127) следует, что для любого $s \in \{0, \dots, S-1\}$ выполнено

$$i \in T_s^{model} \Leftrightarrow i \in T_{(s+\sigma) \bmod S}^{real}, \quad (3.129)$$

то есть

$$T_s^{model} = T_{(s+\sigma) \bmod S}^{real}. \quad (3.130)$$

Пусть $\mathcal{T}_k^{model}$ (соответственно, $\mathcal{T}_k^{real}$) обозначает подпоследовательность элементов исходной трассы $\mathcal{T}$, составленную из тех τ_i , для которых $i \in T_k^{model}$ (соответственно, $i \in T_k^{real}$), с сохранением их хронологического порядка. Из (3.130) следует, что подпоследовательности $\mathcal{T}_s^{model}$ и $\mathcal{T}_{(s+\sigma) \bmod S}^{real}$ состоят из одних и тех же элементов τ_i в одном и том же порядке, а потому

$$\mathcal{T}_s^{model} = \mathcal{T}_{(s+\sigma) \bmod S}^{real}. \quad (3.131)$$

Тем самым, соответствующие множества кеша в ИМИП и в реальной системе обрабатывают согласованные подпоследовательности обращений. Так как кеш детерминирован, для эквивалентности попаданий и промахов достаточно, чтобы на этих подпоследовательностях совпадали результаты сравнения тегов.

При этом из $P = N_{\text{lines}}L$ и кратности N_{lines} числу S следует, что для любого адреса a выполнено:

$$J(a) = \left\lfloor \frac{a \bmod P}{L} \right\rfloor \bmod S = \mathcal{B}(a) \bmod S, \quad (3.132)$$

и, следовательно, в силу определения тега (3.117) имеем:

$$\mathcal{B}(a) = S \cdot \text{Tag}(a) + J(a). \quad (3.133)$$

Для адресов, отображающихся в одно и то же множество, $\mathcal{I}(a)$ фиксирован. Из соотношения (3.133) видно, что при фиксированном $\mathcal{I}(a)$ равенство тегов эквивалентно равенству идентификаторов блоков. Следовательно, для фиксированного s достаточно, чтобы для любых $i, j \in T_s^{model}$ (или, что то же самое, $i, j \in T_{(s+\sigma) \bmod s}^{real}$) выполнялось

$$\mathcal{B}(a_i^{model}) = \mathcal{B}(a_j^{model}) \Leftrightarrow \mathcal{B}(a_i^{real}) = \mathcal{B}(a_j^{real}). \quad (3.134)$$

Доказательство последнего факта опирается на разложение идентификатора блока по номеру страницы и внутрестраничному смещению. Поскольку $P = N_{lines}L$, имеем

$$\mathcal{B}(a) = \left\lfloor \frac{a}{L} \right\rfloor = \left\lfloor \frac{a}{P} \right\rfloor \cdot N_{lines} + \left\lfloor \frac{a}{L} \right\rfloor \bmod N_{lines}. \quad (3.135)$$

Отсюда следует критерий равенства блоков

$$\mathcal{B}(a_i^{real}) = \mathcal{B}(a_j^{real}) \Leftrightarrow \begin{cases} \left\lfloor \frac{a_i^{real}}{P} \right\rfloor = \left\lfloor \frac{a_j^{real}}{P} \right\rfloor, \\ \mathcal{B}(a_i^{real}) \equiv \mathcal{B}(a_j^{real}) \pmod{N_{lines}}. \end{cases} \quad (3.136)$$

Рассмотрим сравнение блоков по модулю N_{lines} . Так как $P = N_{lines}L$, в силу (3.126) и определения (3.116) справедлива цепочка преобразований:

$$\begin{aligned} \mathcal{B}(a_i^{real}) \bmod N_{lines} &= \left\lfloor \frac{a_i^{real}}{L} \right\rfloor \bmod N_{lines} = \left\lfloor \frac{a_i^{real} \bmod P}{L} \right\rfloor = \\ &= \left\lfloor \frac{(a_i^{model} + \sigma L) \bmod P}{L} \right\rfloor = \\ &= \left(\left\lfloor \frac{a_i^{model}}{L} \right\rfloor + \sigma \right) \bmod N_{lines} = \\ &= (\mathcal{B}(a_i^{model}) + \sigma) \bmod N_{lines}. \end{aligned} \quad (3.137)$$

Следовательно, учитывая инвариантность сравнения по модулю относительно прибавления (или вычитания) одного и того же целого числа, получаем:

$$\begin{aligned} \mathcal{B}(a_i^{real}) \equiv \mathcal{B}(a_j^{real}) \pmod{N_{lines}} &\Leftrightarrow \\ \Leftrightarrow \mathcal{B}(a_i^{model}) \equiv \mathcal{B}(a_j^{model}) \pmod{N_{lines}}. \end{aligned} \quad (3.138)$$

Осталось связать равенство номеров страниц. Из условия постраничного отображения (3.115) следует эквивалентность:

$$\left\lfloor \frac{a_i^{real}}{P} \right\rfloor = \left\lfloor \frac{a_j^{real}}{P} \right\rfloor \Leftrightarrow \left\lfloor \frac{a_i^{model}}{P} \right\rfloor = \left\lfloor \frac{a_j^{model}}{P} \right\rfloor. \quad (3.139)$$

Совмещая полученное соотношение с (3.138) и критерием (3.136), заключаем, что выполняется эквивалентность (3.134). Поэтому для обращений, относящихся к одному множеству кеша, число промахов в модельной и реальной системах совпадает:

$$\forall s \in \{0, \dots, S-1\} \quad \mathcal{M}_{(s+\sigma) \bmod S}(A^{real}, \mathcal{T}) = \mathcal{M}_s(A^{model}, \mathcal{T}). \quad (3.140)$$

Общее число промахов при базовом адресе A равно их сумме по всем множествам кеша:

$$\mathcal{M}(A, \mathcal{T}) = \sum_{s=0}^{S-1} \mathcal{M}_s(A, \mathcal{T}). \quad (3.141)$$

Поскольку отображение

$$s \mapsto (s + \sigma) \bmod S \quad (3.142)$$

является биекцией множества $\{0, \dots, S-1\}$ на себя, то из (3.140) следует

$$\begin{aligned} \mathcal{M}(A^{real}, \mathcal{T}) &= \sum_{s=0}^{S-1} \mathcal{M}_{(s+\sigma) \bmod S}(A^{real}, \mathcal{T}) = \sum_{s=0}^{S-1} \mathcal{M}_s(A^{model}, \mathcal{T}) \\ &= \mathcal{M}(A^{model}, \mathcal{T}). \end{aligned} \quad (3.143)$$

Таким образом, число кеш-промахов в ИМИП и реальных вычислительных системах, соответствующих условиям теоремы 3.5 и при выполнении сделанных выше модельных предположений, совпадают. В реальности будет наблюдаться высокая степень соответствия результатов. Условие принадлежности битов адреса, используемых для вычисления индекса множества кеша, полю смещения внутри страницы справедливо для большинства современных кешей L1. Таким образом, ИМИП позволяет достоверно предсказывать число промахов в кеше L1, использующем данное архитектурное решение.

Были сформулированы три теоремы, показывающие влияние нарушения отдельных условий 1–4 теоремы 3.5 на достоверность моделирования.

Таблица 3.25 - Теоремы 3.6–3.8 об эквивалентности числа кеш-промахов.

Теорема	Нарушенные условия
3.6	Условие 3 (смещения базовых адресов массива относительно начала кеш-строки в ИМИП и реальной системе различаются)
3.7	Условие 1 (часть битов индекса множества выходит за пределы смещения страницы)
3.8	Условия 1 и 3 (нарушены одновременно)

Теоремы 3.6–3.8 показывают, что при нарушении отдельных условий теоремы 3.5 равенство числа кеш-промахов между ИМИП и реальной системой в общем случае не гарантируется (существуют конфигурации кеша, отображения страниц и трасс обращений, при которых возникает существенное расхождение по числу промахов). Вместе с тем, для типичных конфигураций современных процессоров и при обычных режимах размещения (окраски) страниц операционной системой, не создающих искусственно неблагоприятных конфликтов, можно ожидать, что поведение модели и реальной системы будет близким, а получаемые в ИМИП оценки числа кеш-промахов окажутся достаточно точными для практического анализа локальности и сравнения вариантов кода.

Модель исполнения программы в подсистеме памяти используется для оценки эффективности использования подсистемы памяти проектируемой ПС методами статического анализа. При этом ограничение модели заключается в концепции «ядро МП – подсистема памяти» или «однойдерный МП – подсистема памяти». Она может использоваться не только для проверки гипотез на существующих ВС, но и применяться для оценки эффективности новых и перспективных ВС с иерархической

структурой памяти, различным числом уровней кеш-памяти и различными алгоритмами взаимодействия уровней кеша друг с другом. Так предлагаемая модель может использоваться в процессе проектирования МП, в т.ч. на базе открытых и свободных архитектур, таких, как, например, RISC-V для определения характеристик подсистемы памяти (число уровней кеш-памяти, их объем и параметры).

Более того, модель может быть использована для оценки необходимости оптимизации ПС для различных уровней кеш-памяти. В отличие от модели оценки эффективности проектируемых ПС может дать более точное представление о проектируемой ПС на целевой ГВС, т.к. ИМИП избавлена от ее ограничений, связанных с реализацией «кеш-независимого» подхода.

Предложенная модель может использоваться для исследования эффективности архитектур МП для решения задач определенных классов, т.к. можно разрабатывать и отлаживать любой алгоритм вытеснения строк, вводить любые системные и пользовательские операции для новых архитектур, выполнять другие настройки модели.

3.5 Комплекс методов повышения эффективности ПС

Основываясь на предложенных моделях, предложен комплекс новых методов повышения эффективности ПС. В методах впервые реализован аналитический подход к оценке проектируемых программных систем на базе совокупности метрик ПС, что позволяет осуществлять принятие проектных решений по повышению эффективности ПС.

Данные методы реализуются в рамках моделей улучшения показателей эффективности ПС и оценки эффективности проектируемых ПС на ГВС с различной архитектурой. Последовательность использования предлагаемых методов следующая:

- в рамках модели улучшения показателей эффективности ПС на соответствующей стадии этапа разработки или модификации ПС выполняется статический анализ исходного кода ПС;

- анализ осуществляется в рамках модели оценки эффективности проектируемых ПС для ГВС с различной архитектурой, определяются значения необходимых метрик ПС;

- в случае принятия решения о возможности улучшения показателей эффективности ПС применяется соответствующий метод повышения эффективности ПС;

- осуществляется переход на следующую итерацию разработки или модификации ПС в рамках модели улучшения показателей эффективности ПС.

Т.к. в качестве ЦВА выбрана ГВС с иерархической структурой памяти, векторным расширением системы команд и GPU, предложено три метода повышения эффективности ПС:

- метод повышения эффективности использования подсистемы памяти;
- метод повышения эффективности векторизации вычислений;
- метод повышения эффективности использования GPU.

3.5.1 Метод повышения эффективности использования подсистемы памяти

В модели оценки эффективности проектируемых ПС для ГВС с различной архитектурой введены метрики ПС, используемые для оценки эффективности использования подсистемы памяти:

- ускорение программы;
- ускорение работы с памятью;
- снижение вероятности кеш-промаха.

Данные метрики используются для оценки эффективности реализации ПС в рамках текущей итерации модели улучшения показателей

эффективности ПС. В зависимости от специфики задачи, могут использоваться отдельные метрики или совокупности нескольких метрик:

- ускорение программы в качестве метрики ПС целесообразно использовать в том случае, если корректировка исходного алгоритма приводит к изменению числа арифметико-логических операций;
- ускорение работы с памятью в качестве метрики ПС целесообразно использовать в том случае, когда число арифметико-логических операций не меняется, однако, число операций обращения к памяти изменяется;
- снижение вероятности кеш-промаха в качестве метрики ПС целесообразно использовать в том случае, когда не изменяется ни число арифметико-логических операций, ни число операций обращения к памяти.

Предлагаемый метод (рисунок 3.39), называемый далее методом улучшения стратегии кеширования (МУСК), можно описать следующим образом.

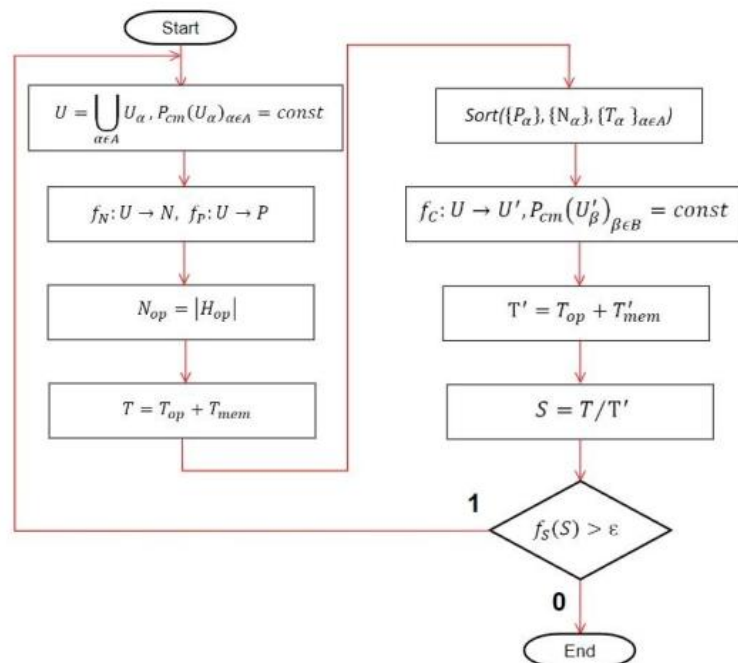


Рисунок 3.39 –Метод улучшения стратегии кеширования

На первом этапе осуществляется оценка выбранного параметра эффективности проектируемой программы, для этого выполняются следующие шаги:

- разбиение множества операций доступа к памяти U на непересекающиеся подмножества U_α в соответствии со способом доступа к памяти, который влияет на вероятность кеш-промаха; получение подмножеств операций с равными значениями вероятности промаха кеша;
- оценка числа элементов N и вероятностей кеш-промахов P в каждом подмножестве;
- оценка числа операций арифметико-логической обработки данных N_{op} для всего алгоритма в целом;
- оценка требуемого параметра (времени выполнения программы, времени выполнения операций работы с памятью и др.).

На втором этапе анализируются полученные результаты, и предпринимается попытка улучшения исходного алгоритма кеширования, для чего необходимо:

- на основе анализа полученных подмножеств операций доступа к памяти и соответствующих значений числа элементов, а также вероятностей кеш-промаха, найти участки кода, оказывающие негативное влияние на эффективность стратегии кеширования (под стратегией кеширования понимается организация доступа к данным, влияющая на эффективность использования кеш-памяти);
- выполнить коррекцию исходного алгоритма, чтобы получить новое разделение набора операций доступа к памяти на подмножества с другими характеристиками $U \rightarrow U^{\wedge'}$;
- оценить значение требуемого параметра для скорректированного алгоритма.

На рисунке 3.39 в качестве метрики ПС используется ускорение программы, которое может быть заменено на другую метрику в соответствии со спецификой задачи. Метод разбиения множества операций доступа к памяти U на непересекающиеся подмножества в соответствии со способом доступа к памяти описан в разделе 3.2.

После выполнения всех необходимых действий принимается решение о повторном применении описанного метода. Приведенный выше метод не позволяет осуществить точную оценку времени выполнения программ, полученного ускорения и других параметров, но дает возможность исследовать особенности базового алгоритма с точки зрения улучшения стратегии кеширования. Целью его применения является выполнение корректировки исходного алгоритма с целью уменьшения вероятности кеш-промахов и времени выполнения программы.

Данный метод, как и многие другие методы, связанные с кеш-оптимизацией, является эффективным в случае реализации циклового доступа к одному или нескольким массивам. При разбиении операций доступа к памяти на подмножества в описанном выше методе, а также при оценке вероятностей кеш-промахов в полученных подмножествах необходимо ответить на несколько вопросов, связанных с определением вероятности «холодных» промахов и максимального размера блока данных, при котором генерируются только «холодные» промахи. Методы, позволяющие ответить на данные вопросы, были представлены ранее в главе 3. В результате будет получена оценка вероятности кеш-промаха при обработке массива заданного размера.

Современные ВС имеют несколько уровней кеш-памяти. С увеличением уровня при приближении к МП уменьшается размер кеш-памяти и увеличивается скорость доступа к данным. Вследствие этого оптимизацию необходимо производить таким образом, чтобы наиболее используемые данные как можно дольше оставались в самой быстрой кеш-памяти.

Стоит также отметить, что оценка вероятности кеш-промахов только с использованием представленных методов не позволяет осуществить оптимизацию для конкретного уровня кеш-памяти, например L2. Данные выражения позволяют выбрать стратегию для доступа к данным. Подобный

подход можно назвать «высокоуровневой» оптимизацией алгоритма кеширования данных

3.5.2 Метод повышения эффективности векторизации вычислений

В модели оценки эффективности проектируемых ПС для ГВС с различной архитектурой введены метрики ПС, используемые для оценки эффективности векторизации вычислений:

- потенциальное ускорение;
- снижение доли невыровненных обращений.

Данные метрики используются для оценки эффективности реализации ПС в рамках текущей итерации модели улучшения показателей эффективности ПС.

Многие задачи, например преобразования линейной алгебры, достаточно хорошо векторизуются. В большинстве случаев может быть получен достаточно эффективный код и после автоматической векторизации, выполненной компилятором. Однако компилятор не может преодолеть некоторых ограничений. В частности, это касается ограничений, связанных с выравниванием данных, подробно описанных в [162].

В процессе автоматической векторизации компилятор выполняет векторизацию циклов, в рамках которых осуществляется доступ к массивам данных. В общем случае, данные загружаемые и обрабатываемые в цикле, могут располагаться в памяти по произвольным адресам. Если данные располагаются по невыровненным адресам, и не прилагается никаких усилий по преодолению этой проблемы, процессор во время выполнения программы будет вынужден постоянно обращаться по невыровненным адресам, что в итоге приводит к падению производительности. Уровень снижения производительности при работе в каждом конкретном случае зависит от типа процессора, однако, для любой ВС более эффективным вариантом является работа с выровненными адресами.

Разработчики компиляторов учитывают это обстоятельство, обработка циклов в общем случае выполняется в три этапа:

- пролог цикла (peeled loop); несколько первых элементов, которые предшествуют первому элементу, расположенному по выровненному адресу;
- выровненная основная часть цикла;
- эпилог цикла (remainder loop); последние несколько элементов цикла, следующие за последним вектором, расположенным по выровненному адресу.

Схема обработки цикла блоком автовекторизации оптимизирующего компилятора представлена на рисунке 3.40.

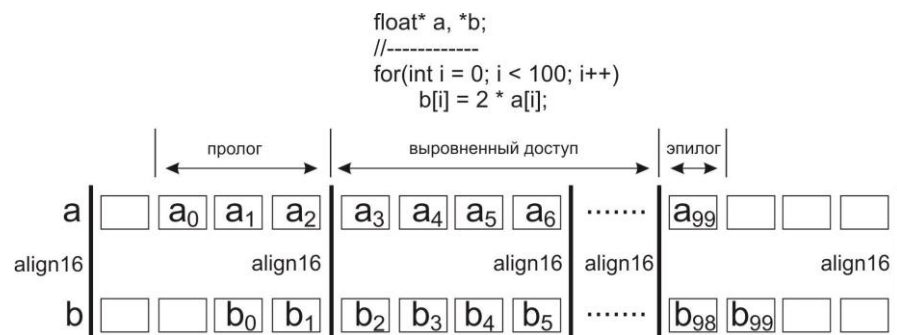


Рисунок 3.40 - Схема обработки цикла компилятором

Необходимость обработка прологов и эпилогов цикла снижают производительность полученной программы. Более того, т.к. компилятор не делает предположений о выравнивании данных, которые будут обрабатываться в процессе работы программы, он в любом случае формирует код пролога и эпилога. В примере на рисунке 3.40 используется векторизация SSE, поэтому данные должны быть выровнены таким образом, чтобы адрес начального элемента вектора был кратен 16 (align16).

Кроме того, при выполнении автоматической векторизации существует еще одно ограничение – доступ к нескольким массивам. Если в цикле используются несколько массивов, выровнены они по-разному, компилятор выбирает один из них, обработка этого массива будет осуществляться с использованием выровненных считываний из памяти, обработка остальных – с использованием невыровненных обращений; пролог и эпилог цикла

формируется на основе выбранного массива. В примере на рисунке 3.40 в качестве основного используется массив *a*, при обращении к элементам массива *b* используется невыровненный доступ.

Для понимания того, какие циклы компилятор векторизовал, а какие нет, и по какой причине, можно использовать отчеты компилятора. Отчеты компилятора являются мощным инструментом анализа работы компилятора. Отчеты компилятора представляют собой текстовые файлы, которые в зависимости от степени детализации содержат ту или иную информацию. Например, отчет может сообщить о том, какие циклы были векторизованы, а какие не были векторизованы. Отчет может сообщить о причинах, по которым цикл не был векторизован, о наличии различных зависимостей по данным. Отчеты также предоставляют программистам массу другой дополнительной полезной информации, которая может быть использована для повышения качества программного обеспечения.

В работе [33] описывается процесс повышения эффективности реализации одной из операций BLAS. Ядром преобразования является функция вычисления скалярных произведений.

```
for (int i = 0; i < Arow_Y; i++)
{
    Y[i] = 0;
    for (int j = 0; j < Acol_X; j++)
        Y[i] += cur[j] * X[j];
    cur = cur + N;
}
```

Компиляция выполнялась с использованием следующих ключей.

```
icrc *.cpp -o vect -qopenmp -xAVX -qopt-report=5
```

Был сгенерирован отчет компилятора. Компилятор создал две версии кода: векторизованную и не векторизованную. В векторизованной версии был сгенерирован пролог.

```

LOOP BEGIN at avx_0.cpp(91,5)
  LOOP BEGIN at avx_0.cpp(94,9)
    <Peeled loop for vectorization, Multiversiоned v1>
    remark #25015: Estimate of max trip count of loop=7
  LOOP END

```

Максимальная длина пролога равна 7, т.к. используется технология AVX и тип данных float. Далее был сгенерирован основной векторизованный цикл.

```

LOOP BEGIN at avx_0.cpp(94,9)
<Multiversiоned v1>
remark #25228: Loop multiversiоned for Data Dependence
remark #15389: vectorization support: reference cur[j] has unaligned access
[ avx_0.cpp(95,21) ]
remark #15388: vectorization support: reference X[j] has aligned access  [
avx_0.cpp(95,30) ]
remark #15381: vectorization support: unaligned access used inside loop
body
// -----
LOOP WAS VECTORIZED
// -----
LOOP END

```

Цикл векторизован, но был использован невыровненный доступ. Далее был сформирован эпилог цикла.

```

LOOP BEGIN at avx_0.cpp(94,9)
<Remainder loop for vectorization, Multiversiоned v1>

```

remark #15389: vectorization support: reference cur[j] has unaligned access
[avx_0.cpp(95,21)]

remark #15388: vectorization support: reference X[j] has aligned access [avx_0.cpp(95,30)]

remark #15381: vectorization support: unaligned access used inside loop body

remark #15335: remainder loop was not vectorized: vectorization possible but seems inefficient.

LOOP END

Видно, что в эпилоге используется невыровненный доступ, кроме того, эпилог не был векторизован, т.к. компилятор посчитал, что векторизация в данном случае неэффективна. Помимо векторизованной версии цикла компилятор создал не векторизованную версию.

LOOP BEGIN at фмч_0.cpp(94,9)

<Multiversiоned v2>

remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning

remark #25439: unrolled with remainder by 2

LOOP END

LOOP BEGIN at avx_0.cpp(94,9)

<Remainder, Multiversiоned v2>

LOOP END

LOOP END

Видно, что компилятор частично векторизовал цикл, однако:

- не были векторизованы пролог и эпилог цикла;
- в векторизованной части цикла используется невыровненный доступ к данным.

Предлагаемый метод, называемый далее методом ведущих векторов (MBV) может существенно повысить эффективность ПС, в процессе выполнения которой осуществляется регулярный доступ к данным. Подобный доступ к данным характерен для большого количества преобразований, примером которых могут служить операции BLAS, поэтому MBV существенно улучшает характеристики подобных программных реализаций, его применение позволяет:

- избежать формирования прологов и эпилогов;
- обеспечить одновременный выровненный доступ к нескольким массивам, обрабатываемым в цикле, что по умолчанию в общем случае не представляется возможным.

MBV содержит два этапа (рисунок 3.41).

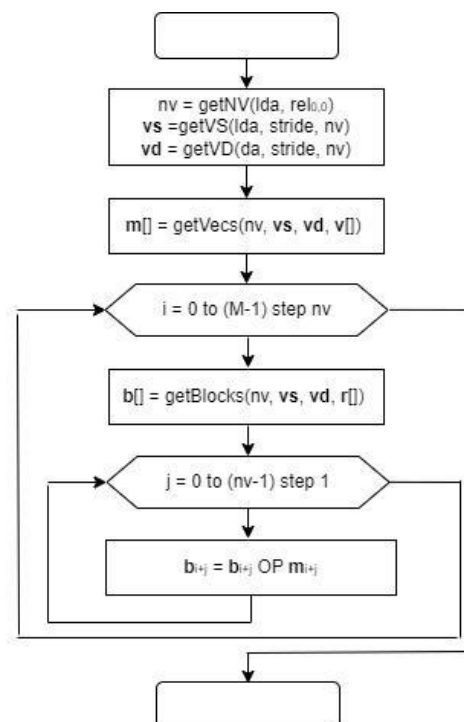


Рисунок 3.41 - Метод ведущих векторов

На первом этапе осуществляется формирование набора необходимых ведущих векторов с нужным выравниванием $m[]$ на основе исходных векторов $v[]$, при этом выполняется:

- определение размерности используемых векторов, их количества nv , смещения их начальных элементов vd , размерности обрабатываемых

массивов данных в направлении обработки `lda` и других характеристик векторизуемого цикла;

- формирование выровненных ведущих векторов в соответствии с определенными ранее параметрами **`m[]`**;
- определение длин полученных векторов **`vs`**, смещений проекций векторов относительно обрабатываемых блоков данных **`r[]`**, под проекцией в данном случае понимается соответствие элементов ведущих и обрабатываемых векторов, которое не совпадает для различных ведущих векторов.

На втором этапе на основе полученных результатов осуществляется изменение циклов доступа к данным:

- развертка цикла в соответствии с числом ведущих векторов;
- изменение параметров цикла (стартовый элемент, количество итераций и т.д.) в соответствии с параметрами ведущих векторов.

На рисунке 3.41 дополнительно использованы следующие обозначения:

- **`b[]`** – блоки обрабатываемых данных с нужным выравниванием, полученные на основе исходных блоков обрабатываемых данных **`r[]`**;
- **OP** – векторная операция, применяемая к блокам данным **`r[]`** и векторам **`v[]`**.

Все вектора на рисунке 3.41 обозначены жирным шрифтом (**`vs`**, **`vd`**, **`b`**, **`m`**), массивы векторов – жирным шрифтом с квадратными скобками (**`v[]`**, **`m[]`**, **`r[]`**, **`b[]`**). На рисунке 3.41 также не показана развертка циклов:

- внутренний цикл может быть развернут на число итераций, равное числу ведущих векторов **`nv`**;
- внешний цикл также может быть развернут на несколько итераций для повышения производительности (шаг цикла делается кратным ширине векторных регистров, отношение шага к ширине регистров больше 1).

После этого в программе необходимо либо сообщить компилятору о необходимости использования выровненного доступа с помощью специальных директив (например, `#pragma vector aligned`), либо использовать

соответствующие машинные инструкции или интринсики (например, `movapd, _mm_load_pd` для SSE).

3.5.3. Метод повышения эффективности использования GPU

Метод повышения эффективности использования GPU представлен на рисунке 3.42.

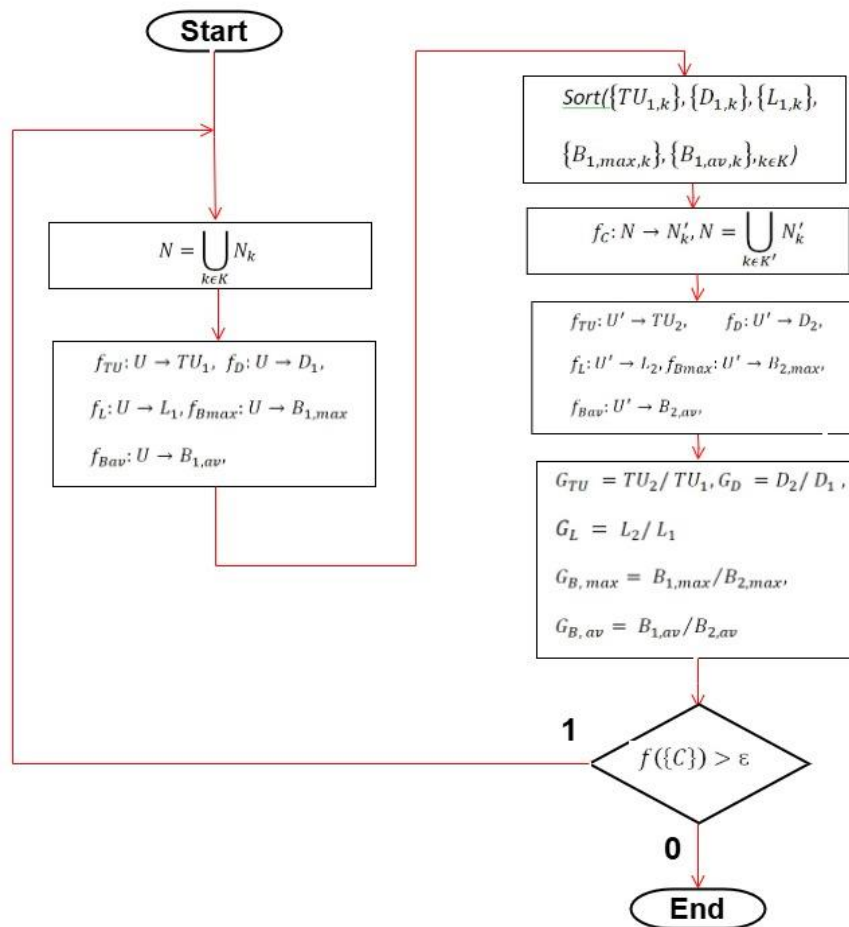


Рисунок 3.42 - Метод повышения эффективности использования GPU

1. Первый этап:

1.1. Разбиение множества операций N на вычислительные ядра (kernels) N_k , задание параметров ядер: число блоков потоков (NB), количество потоков в блоке (NT_b).

1.2. Оценка значений параметров TU_1 , D_1 , L_1 , $B_{1,max}$, $B_{1,av}$ отдельно для каждого вычислительного ядра и на всем множестве операций:

- доля использованных потоков (TU_1), рассчитываемая в соответствии с (3.38); демонстрирует, какая доля из вычислительных потоков, предоставляемых GPU, используется в вычислениях.

- средняя плотность вычислений (D_1), рассчитываемая в соответствии с (3.40); характеризует отношение количества вычислительных операций к количеству запросов в память

- средняя степень локальности данных (L_1), рассчитываемая в соответствии с (3.42); оценивает одно из условий оптимальности доступа к данным, а именно расположение данных в памяти, к которым осуществляется обращение потоками одного варпа;

- максимальный уровень несбалансированности потоков ($B_{1,max}$), рассчитываемый в соответствии с (3.44); оценивает влияние расхождения потоков на итоговую эффективность ПС;

- средний уровень несбалансированности потоков ($B_{1,av}$), рассчитываемый в соответствии с (3.46).

2. Второй этап:

2.1. Анализ полученных результатов, поиск участков, оказывающих негативное влияние на эффективность использования GPU. Необходимо проанализировать полученные значения метрик ПС для каждого вычислительного ядра.

2.2. Коррекция алгоритма, получение нового разбиения $N \rightarrow N_k$.

2.3. Оценка значений параметров TU_2 , D_2 , L_2 , $B_{2,max}$, $B_{2,av}$ отдельно для каждого вычислительного ядра и на всем множестве операций

2.4. Оценка метрик ПС G_{TU} , G_D , G_L , $G_{B,max}$, $G_{B,av}$

После выполнения всех необходимых действий принимается решение о повторном применении описанного метода, основываясь на некоторой

метрике ПС. Приведенный выше метод так же, как и МУСК не позволяет осуществить точную оценку времени выполнения программ на GPU, полученного ускорения и других параметров, но дает возможность исследовать особенности базового алгоритма с точки зрения повышения эффективности использования GPU. Целью его применения является выполнение корректировки исходного алгоритма и улучшения количественных характеристик выбранных метрик, что оказывает влияние на эффективность проектируемой ПС.

3.6 Методология проектирования эффективных ПС

Разработана методология проектирования эффективных ПС для решения инженерных задач в ГВС (рис.3.43), основанная на совокупности моделей и методов, реализуемых в процессе проектирования и модификации ПС в рамках модели улучшения показателей эффективности ПС:

- математической модели оценки эффективности проектируемых ПС;
- имитационной модели исполнения программы в подсистеме памяти;
- комплексе методов повышения эффективности ПС.

Данная методология базируется на взаимодействии проектировщика ПС с системой поддержки жизненного цикла ПС. Процесс проектирования или модификации ПС начинается с поступления соответствующего запроса (сообщения), который поступает от Инжиниринговой команды, выступающей инициатором данного процесса.

В соответствии с моделью улучшения показателей эффективности ПС Проектировщиком осуществляется уточнение требований и выбор или разработка алгоритма решения задачи.

На следующем этапе Проектировщик реализует проектную процедуру модификации алгоритма/оптимизации кода программы. В процессе реализации данной процедуры, в зависимости от ЦВА и требований, предъявляемых к ПС, выполняются следующие действия:

- улучшение стратегии кеширования путем реализации МУСК (рисунок 3.39);
- повышение эффективности векторизации вычислений путем реализации MBV (рисунок 3.41);
- повышение эффективности использования GPU путем реализации предложенного метода (рисунок 3.42).

Далее производится статический анализ исходного кода средствами СП ЖЦ, которая реализует предложенные модели и методы. СП ЖЦ ассистирует Проектировщику, который выступает в роли лица, принимающего проектные решения. Инструментальными средствами СП ЖЦ осуществляется статический анализ исходного кода программы с целью получения количественных значений метрик ПС математической модели оценки эффективности проектируемых ПС в различных ГВС. В ходе статического анализа производится:

- анализ абстрактного синтаксического дерева AST, реализуется метод количественной оценки метрик использования подсистемы памяти;
- имитационное моделирование исполнения ПС на ГВС, реализуется имитационная модель исполнения программы в подсистеме памяти.

Имитационное моделирование осуществляется частично-динамическим методом, в рамках которого осуществляется: подготовка исходного кода, его интерпретация и симуляция выполнения.

В результате определяются количественные значения метрик ПС, которые в дальнейшем анализируются Проектировщиком. На основе проведенного анализа Проектировщик принимает решение о выполнении новой итерации проектных процедур в рамках модели улучшения показателей эффективности ПС.

Полученные параметры выявленных паттернов доступа запоминаются для повторного использования в дальнейшем на этапе получения значений метрик ПС.

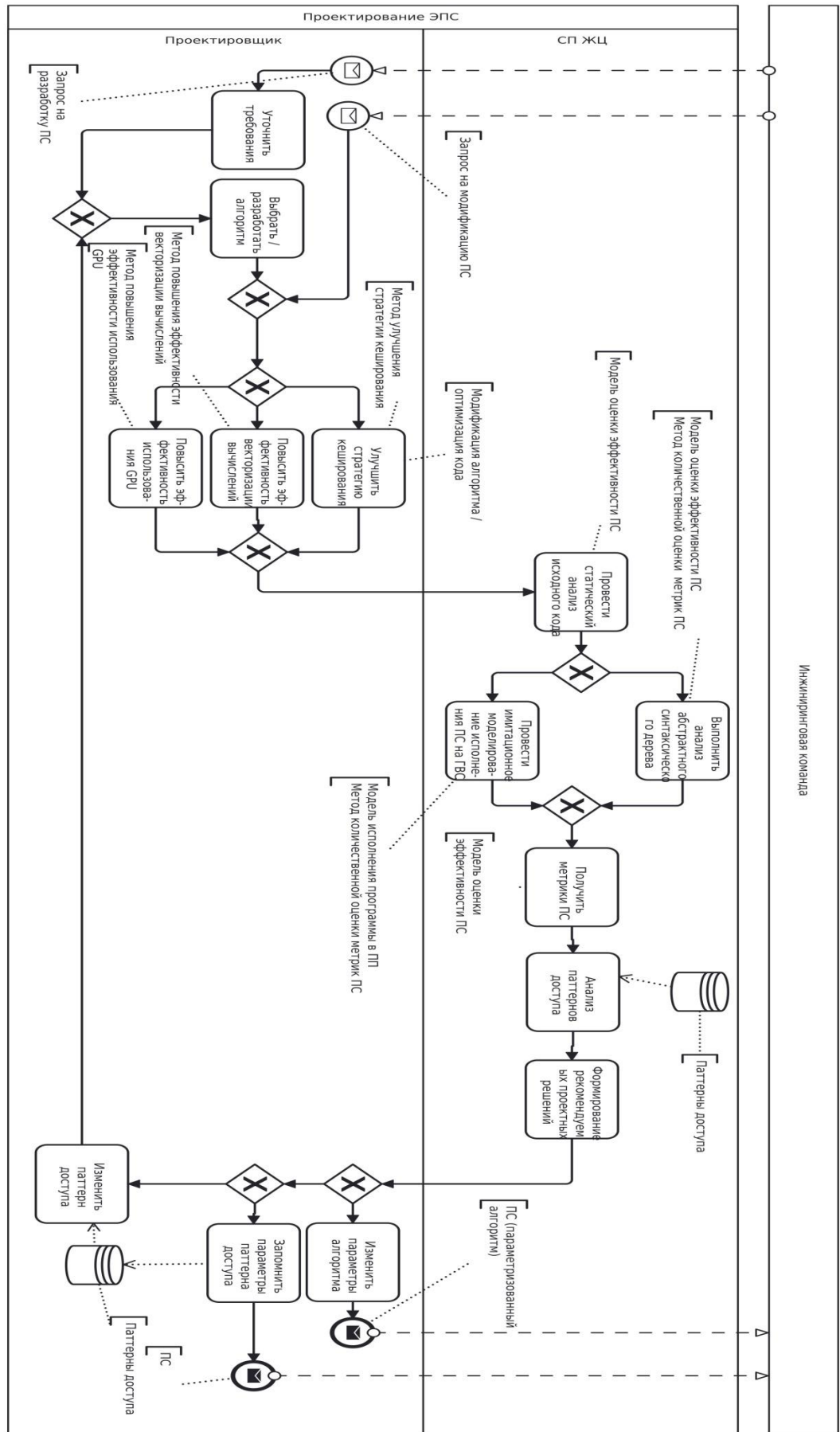


Рисунок 3.43 - Методология проектирования эффективных ПС

3.6.1 Получение метрик ПС методами статического анализа

Анализ абстрактного синтаксического дерева. AST представляет собой древовидное представление структуры исходного кода программы. Каждый узел дерева представляет собой конструкцию языка программирования, такую как объявление переменной, оператор, вызов функции и т.д. AST используется компиляторами и другими инструментами анализа кода для понимания структуры и семантики программы. Существуют ПС, например CLANG [8], которые предоставляют мощные инструменты для построения и обработки AST. AST необходимо для проведения семантического анализа, в рамках которого выполняются проверки типов, разрешения имен и реализуются другие аспекты проверки семантики программы. Каждый узел в AST представляет собой определенный тип конструкции языка программирования. Примерами таких узлов являются:

1. Declaration: Объявление переменной, функции, класса и т.д.
2. Statement: Оператор, такой как присваивание, цикл, условный оператор.
3. Expression: Выражение, такое как арифметическая операция, вызов функции, доступ к члену класса.
4. Type: Тип данных, такой как int, float, char, pointer.

Узлы AST связаны между собой, отражая структуру кода. Например, узел "объявление функции" может иметь дочерние узлы для параметров функции, типа возвращаемого значения и тела функции.

В процессе анализа AST в рамках предлагаемой методологии осуществляется поиск и формализация паттернов доступа в память. Данная задача решается путем поиска соответствующих узлов в AST и их взаимосвязей.

Полученные паттерны доступа могут быть двух типов:

- выявленные ранее, для которых уже были получены необходимые метрики; характеристики данных паттернов используются для анализа эффективности ПС;

- новые паттерны доступа, информация о которых отсутствует в системе; для них производится имитационное моделирование с целью определения метрик ПС, полученные параметры запоминаются для повторного использования.

Кроме того, в отдельных случаях для новых паттернов доступа метрики ПС могут быть определены аналитически.

Формально, поведение программы в подсистеме памяти может быть описано трассой доступа к памяти — упорядоченной последовательностью событий:

$$T = \langle e_1, e_2, \dots, e_n \rangle \quad (3.144)$$

где каждое событие e_i представляет собой как минимум кортеж $(op_i, addr_i)$, состоящий из типа операции $op_i \in \{Read, Write\}$ и виртуального адреса $addr_i$.

Анализ кеш-эффективности сводится к применению функции-симулятора Sim к трассе T для заданной конфигурации кеша C :

$$M = Sim(T, C) \quad (3.145)$$

\Традиционным результатом трасс-ориентированных кеш-симуляторов M является набор агрегированных счетчиков кеш-промахов, как правило, с детализацией по уровню кеша l и типу операции op :

$$M = \{M_{l,op}\} \quad (3.146)$$

Несмотря на свою количественную точность, данная формулировка обладает невысокой диагностической ценностью для разработчика. Ключевое ограничение кроется в определении трассы T , которая содержит лишь низкоуровневую информацию (адреса) и лишена какой-либо связи с высокоуровневыми семантическими конструкциями исходного кода. Разработчик, получив отчет M , не может ответить на главные вопросы:

"Какие именно структуры данных вызывают эти промахи?" и "Какие паттерны доступа к ним необходимо оптимизировать?".

Для преодоления этого «семантического разрыва» и предоставления разработчику практически значимой информации, необходимо расширить определение трассы, обогатив ее семантическим контекстом. Введем понятие семантически аннотированной трассы T_a :

$$T_a = \langle e_{a,1}, e_{a,2}, \dots, e_{a,n} \rangle \quad (3.147)$$

где каждое аннотированное событие $e_{a,i}$ представляет собой кортеж $(op_i, addr_i, S_i)$. Здесь S_i представляет собой семантический дескриптор, который идентифицирует объект исходного кода, связанный с доступом по адресу $addr_i$. Как минимум, S_i должен содержать идентификатор переменной или структуры данных (например, имя массива).

Соответственно, новая функция-симулятор Sim_a должна не просто считать общее число промахов, а вычислять их распределение по семантическим объектам. Результатом ее работы является отображение M_a , которое ставит в соответствие каждому уникальному семантическому дескриптору s его собственный набор счетчиков промахов:

$$M_a(s) = \{M_{s,l,op}\} \quad (3.148)$$

где $M_{s,l,op}$ представляет собой число промахов для объекта s на уровне кеша l при операции типа op . Такая детализация позволяет перейти от общего вывода "в программе много промахов" к конкретному диагностическому заключению.

Таким образом, задача проведения имитационного моделирования с целью определения метрик ПС, связана с автоматической генерацией семантически аннотированной трассы доступа к памяти T_a : и вычислении на ее основе детализированного распределения кеш-промахов $M_a(s)$ для каждого семантически значимого объекта исходного кода s .

Каждое событие в трассе соответствует одному обращению к памяти и представляется кортежем

$$e_a = (addr, op, size, P) \quad (3.149)$$

где

$addr$ — эффективный адрес обращения к памяти;

$op \in \{Read, Write\}$ — тип операции;

$size$ — размер передаваемых данных;

P — набор аннотаций, несущих высокоуровневую семантическую информацию.

Множество параметров P включает, в частности:

- имя и тип переменной или массива, к которому относится обращение;
- при доступе к структурам — имя и тип поля;
- имя функции и (при необходимости) контекст вызовов;
- позицию в исходном файле (файл, строка);
- принадлежность к конкретной итерации цикла или логическому блоку программы на уровне исходного кода.

Для снижения накладных расходов при сохранении достоверности целевых метрик ПС в процессе имитационного моделирования используются два практических допущения.

Допущение 1 (подмножество наблюдаемых переменных).

Ранее введены семантически аннотированная трасса обращений к памяти (3.147) и целевая форма результата анализа как отображение (3.148). В рамках данного допущения вводится множество всех объектов, которые в принципе могут быть источниками обращений к памяти,

$$S = \{s\}, \quad (3.150)$$

и выбирается наблюдаемое подмножество $S_{obs} \subseteq S$, например

$$S_{obs} = S_{arr} \cup S_{struc(arr)} \quad (3.151)$$

т.е. массивы и связанные с ними поля структур. Для объектов $s \in S_{obs}$ метрики $M_a(s)$ вычисляются и агрегируются полностью, тогда как обращение к объектам $S_{ign} = S \setminus S_{obs}$ (в частности, к «чисто скалярным» переменным, не влияющим на адресацию и структуру управления, в т.ч. на

границы циклов, условия ветвлений) могут быть опущены на этапе формирования трассы или исключены из последующей агрегации.

Тем самым предполагается, что вклад игнорируемых объектов в целевые метрики мал по сравнению с вкладом наблюдаемых объектов, игнорируемые объекты дают не более ε доли промахов от общего числа. Формально, для целевых конфигураций анализа ожидается выполнение соотношения вида

$$\frac{\sum_{s \in S_{ign}} M_{s,l,op}}{\sum_{s \in S} M_{s,l,op}} \leq \varepsilon, 0 \leq \varepsilon \ll 1 \quad (3.152)$$

что позволяет существенно сократить стоимость моделирования и объём трассы без заметного искажения распределения кеш-промахов по основным структурам данных/

В рамках этого допущения допускается использовать один из следующих уровней детализации:

- учитывать только обращения к элементам массивов;
- учитывать обращения к элементам массивов и к выбранному подмножеству скалярных переменных;
- учитывать обращения ко всем переменным программы.

Допущение 2 (фильтрация несущественных операций).

В постановке задачи метрики кеш-поведения (3.148) вычисляются на основе трассы обращений к памяти (3.115) и симулятора кеш-подсистемы

$$M_a(s) = Sim_a(T_a, C) \quad (3.153)$$

При этом трасса (3.147) содержит события обращений к памяти и семантические аннотации (3.149).

Обозначим через E^{full} полное исполнение программы (последовательность всех элементарных событий, включая вычислительные операции и обращения к памяти), а через $\pi()$ - операцию извлечения аннотированной трассы обращений к памяти.

$$T_a^{full} = \pi_a(E^{full}), \quad (3.154)$$

Введём множество существенных операций O_{ess} , которые необходимо моделировать точно, поскольку они могут изменить трассу (3.147):

- все операции доступа к памяти и адресная арифметика;
- операции, влияющие на управляющие решения (условия ветвлений), границы и счётчики циклов;
- операции, влияющие на вычисление индексов массивов (указателей).

Остальные вычислительные операции, в частности арифметико-логические преобразования значений элементов массивов и структур, которые не участвуют в вычислении адресов и условий ветвлений/циклов, относятся к множеству несущественных O_{ign} и могут не моделироваться детально/

Пусть E^{red} представляет собой редуцированное исполнение, полученное из E^{full} удалением операций из O_{ign} при сохранении точного моделирования O_{ess} . Содержание допущения состоит в следующем

$$\pi_a(E^{red}) = \pi_a(E^{full}), \quad (3.155)$$

то есть редуцированная симуляция сохраняет ту же аннотированную трассу обращений к памяти, а, следовательно, для фиксированной конфигурации кеша C

$$Sim_a(\pi_a(E^{red}), C) = Sim_a(\pi_a(E^{full}), C), \quad (3.156)$$

и искомые метрики $M_a(s)$ совпадают.

В рамках этого допущения принимаются следующие упрощения:

1. Всегда учитываются все операции доступа к памяти, именно они формируют трассу, подаваемую на вход кеш-симулятору.
2. Не включаются в трассу и могут моделироваться упрощённо арифметические и логические операции без побочных эффектов, которые:
 - не изменяют данные в памяти;
 - не участвуют в вычислении адресов обращений к памяти;
 - не влияют на условия ветвлений, счётчики циклов и другие конструкции, определяющие порядок обращений к памяти.

В частности, это относится к операциям над скалярными переменными, которые не используются в индексах массивов, адресной арифметике и управляющих выражениях.

Таким образом, построение AST производится после семантического преобразования исходного кода, который включает в себя:

- удаление всех вхождений переменных, не наблюдаемых в рамках допущения 1;
- удаление всех операций, игнорируемых в рамках допущения 2.

В дальнейшем полученное AST может быть использовано:

- для определения паттернов доступа к данным;
- проведения имитационного моделирования.

Имитационное моделирование осуществляется в соответствии с диаграммой, представленной на рисунке 3.44.

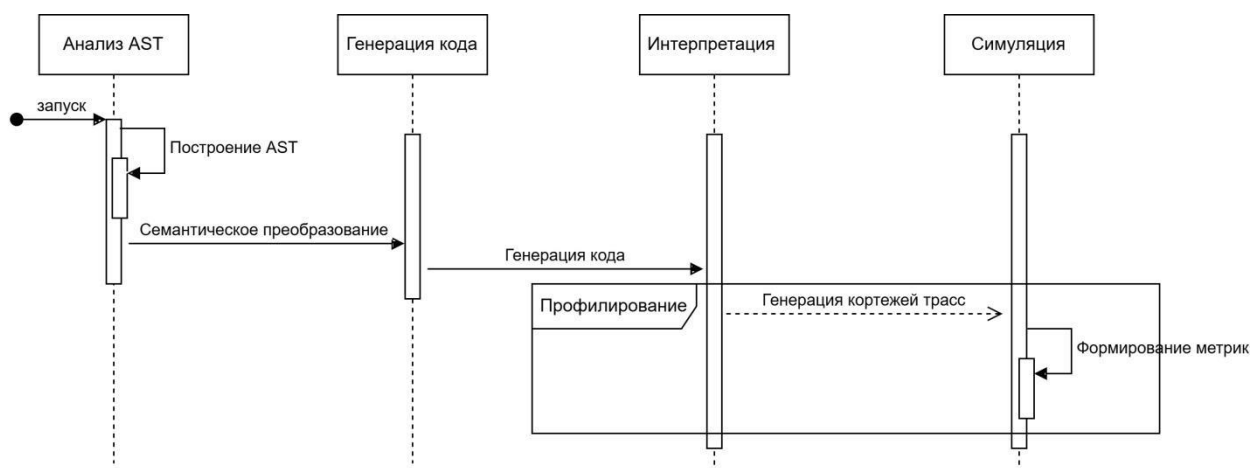


Рисунок 3.44 – Инженерная методика выполнения имитационного моделирования.

На первом этапе выполняется преобразование исходного кода программы в соответствии с допущениями, приведенными выше. Для этого выполняются следующие действия:

- построение AST;
- семантическое преобразование AST;
- генерация кода, полученного на базе полученного AST.

На втором этапе выполняется имитационное моделирование, реализуемое в виде цикла, на каждой итерации которого выполняются следующие действия:

- интерпретатор, выполняя операции, формирует трассы, необходимые для симуляции;
- симулятор, получив очередную трассу, симулирует ее выполнение, при этом изменяя состояние счетчиков, используемых для накопления параметров, необходимых для расчета значений метрик ПС.

При выполнении кеш-симуляции интерпретатор формирует последовательность семантически-аннотированных трасс (3.147), содержащих кроме адресов и типа операции ряд дополнительных параметров.

Дополнительные параметры позволяют после выполнения имитационного моделирования не только получить значения метрик, но и проанализировать то, каким образом они формируются.

Таким образом, после окончания симуляции в результате моделирования будет получены не только значения количества обращений в память, кеш-попаданий и кеш-промахов, но и распределения этих параметров, например, по массивам или функциям. Становится возможным проведение более детального анализа, например, оценки эффективности использования массивов в различных функциях, методах классов т.д.

3.6.2 Оценка эффективности ПС для гетерогенных кластеров

Под гетерогенным кластером понимается вычислительный кластер, состоящий из узлов с различными характеристиками. Часто гетерогенность кластера приобретает в результате включения в кластер различных узлов в разные периоды времени. В результате получается кластер, содержащий узлы с различным оборудованием, имеющим различные параметры производительности [173]. Более того, различные узлы могут содержать

оборудование, для которого используются различные технологии программирования. Например, кроме CPU, разные узлы могут содержать различные типы ускорителей, например, GPU или FPGA, при этом необходимо эффективно использовать все оборудование.

Примером подобного вычислительного кластера является кластер Волгоградского государственного технического университета. Вычислительные узлы кластера являются неоднородными, так как вводились в эксплуатацию в различное время и собирались в единый кластер по мере поступления. Они поделены на различные группы (таблица 3.26). Группы узлов определены сопоставимой процессорной производительностью. Вычислительные эксперименты в рамках диссертационного исследования проводились на узлах данного вычислительного кластера.

Таблица 3.26. Узлы вычислительного кластера Волгоградского государственного технического университета

Наименование узлов	CPU	RAM	Сопроцессоры	Количество узлов
Головной узел	2xAMD Opteron Processor 6272	64 Гб DDR3	-	1
intelv3	2xIntel Xeon CPU E5-2650 v3 @ 2.30GHz	128 Гб DDR4	RTX 3060 (12 Гб) Nvidia P100 (12	11
intelv4	2xIntel Xeon CPU E5-2650 v4 @ 2.20GHz	256 Гб DDR4	-	3
kn1	Intel Xeon Phi CPU 7210 @ 1.30GHz	192 Гб DDR4 + 16 Гб GDDR5	-	8
special	Intel Xeon CPU E5-2660 0 @ 2.20GHz	128 Гб DDR3	3x RTX 3060	2
gold	Intel Xeon Gold 6130 CPU @ 2.10GHz	256 Гб DDR4	2x Nvidia V100	1

Использование предложенной методологии для гетерогенных кластеров предполагает выполнение нескольких этапов:

- реализация предложенной методологии для всех задействованных в вычислениях узлов в отдельности;

- вычисление необходимых метрик ПС для узлов кластера

Полученные результаты можно использовать для балансировки загрузки вычислительных узлов, с целью сокращения максимального времени локальных вычислений (3.157).

$$T = \max_i T_i \quad (3.157)$$

где T – общее время вычислений;

T_i – локальное время вычислений на i -м узле кластера.

В качестве дополнительных метрик ПС в данном случае можно использовать увеличение максимальной степени сбалансированности вычислений (3.158) и увеличение средней степени сбалансированности вычислений (3.159).

Процесс повышения эффективности ПС в данном случае заключается в максимизации данных метрик. Следует отметить, что данный подход не учитывает время передачи данных между вычислительными узлами.

$$B_{I, max} = B_{1, max} / B_{2, max}, \quad (3.158)$$

$$B_{max} = 1 - \min_i T_i / T$$

$$B_{I, av} = B_{1, av} / B_{2, av}, \quad (3.159)$$

$$B_{av} = \sum_i \frac{T_i}{T} / N$$

где N – число узлов.

Данный подход позволяет заменить традиционный подход, который заключается в определении производительности всех сегментов кластера при решении основных типов задач [181, 182] или профилировании ПС на

различных узлах. Полученные результаты в дальнейшем используются в процессе распределения вычислительной нагрузки между узлами кластера.

3.7 Основные результаты и выводы по главе 3

Разработана методология проектирования эффективных ПС для решения инженерных задач в ГВС, основанная на исследовании жизненного цикла ПС, впервые реализующая подход непрерывного улучшения показателей проектируемых ПС, отличающаяся от известных реализацией предложенных моделей и методов и позволяющая принимать проектные решения по эффективному использованию вычислительных ресурсов на начальных этапах жизненного цикла ПС.

Исследован ЖЦ ПС, предложена количественная оценка эффективности на различных этапах ЖЦ. Для количественной оценки степени эффективности ПС предложена мера аппаратурно-временной сложности задачи. Для повышения эффективности ПС на этапах разработки и модификации предложена модель улучшения показателей эффективности ПС, представляющая собой модель итеративного процесса разработки и модификации программных систем. Подробно проанализирован этап разработки, осуществлено сравнение традиционного подхода к проектированию ПС и предлагаемого нового подхода. Выявлены основные недостатки традиционного подхода, предложены решения, позволяющие преодолеть указанные недостатки. Рассмотрен этап модификации ПС, реализующий новую предлагаемую модель улучшения показателей эффективности программных систем.

Предложены математическая модель оценки эффективности проектируемых ПС для различных ГВС и имитационная модель исполнения программы в подсистеме памяти, использующиеся для принятия проектных решений в рамках модели улучшения показателей эффективности программных систем.

Модель оценки эффективности проектируемых ПС представляет собой математическую модель, основанную на совокупности метрик ПС, и позволяет оценить эффективность ПС в ГВС с целевой архитектурой. Предложенные метрики используются при сравнении эффективности различных программных реализаций решения поставленной задачи и позволяют оценить:

- эффективность использования подсистемы памяти;
- эффективность векторизации вычислений;
- эффективность использования GPU.

Метрики, используемые в процессе повышения эффективности использования ПС подсистемы памяти:

- ускорение программы;
- ускорение работы с памятью;
- снижение вероятности кеш – промаха.

Метрики, используемые в процессе повышения эффективности векторизации вычислений:

- увеличение потенциального ускорения;
- снижение доли невыровненных обращений.

Метрики, используемые в процессе повышения эффективности использования GPU:

- увеличение доли использованных потоков;
- увеличение средней плотности вычислений;
- увеличение средней степени локальности данных;
- увеличение максимальной степени сбалансированности потоков.
- увеличение средней степени сбалансированности потоков.

Приведен прикладной протокол определения метрик ПС, в рамках которого произведена оценка:

- вероятности «холодных» кеш-промахов;
- оценка вероятности кеш-промахов при цикловом доступе к данным;

Приведены аналитические оценки метрик ПС для программных реализаций ряда матричных преобразований, приводятся результаты вычислительных экспериментов.

Предложен аналитический метод определения максимального размера блока при регулярном доступе к данным, который позволяет оценить вероятность кеш-промахов при обработке массивов заданного размера и максимальный размер блоков данных, при котором генерируются только «холодные» промахи и не происходит вытеснения строк из кеш-памяти. Метод позволяет производить исследование показателей эффективности использования подсистемы памяти проектируемых ПС. Сформулированы и доказаны 4 теоремы, формально описывающие отображение множества кешируемых данных на множества кеш-памяти. Сформулирован ряд следствий из теорем, позволяющих аналитически оценить степень заполняемости множеств кеш-памяти с различными характеристиками при организации регулярного доступа к данным на основе различных шаблонов доступа. Реализация данного метода в процессе анализа ПС позволяет выбирать параметры алгоритмов, минимизирующие количество кеш-промахов при доступе к данным, тем самым значительно ускоряя работу ПС.

Предложена модель исполнения программы в ГВС с иерархической структурой памяти, основанная на множествах системных и пользовательских операций, где использован полный набор параметров, характеризующих различные уровни подсистемы памяти, что позволяет определять характеристики ПС на конкретной ЦВА, а также характеристики подсистемы памяти для проектируемой ПС. Описаны множества пользовательских и системных операций. Модель представляет собой детерминированную имитационную компьютерную модель для оценки показателей эффективности проектируемых ПС, выступающих в виде технических объектов проектирования, оценка осуществляется на базе предложенной модели оценки эффективности проектируемых ПС на ГВС с

различными характеристиками и метрик ПС. Описаны варианты использования модели и наборы выходных параметров.

Предложен комплекс новых методов повышения эффективности ПС. В методах впервые реализован аналитический подход к оценке проектируемых ПС на базе совокупности метрик ПС, что позволяет принимать проектные решения по повышению эффективности ПС. Данные методы реализуются в рамках моделей улучшения показателей эффективности ПС и оценки эффективности проектируемых ПС на ГВС с различной архитектурой:

- метод повышения эффективности использования подсистемы памяти;
- метод повышения эффективности векторизации вычислений;
- метод повышения эффективности использования GPU.

Разработана методология проектирования эффективных ПС для решения инженерных задач в ГВС, основанная на совокупности моделей и методов, базирующаяся на взаимодействии проектировщика и СП ЖЦ. Описаны метод оценки эффективности ПС для гетерогенных кластеров и методы статического анализа для получения метрик ПС.

4. ВЕРИФИКАЦИЯ МЕТОДОЛОГИИ ПРОЕКТИРОВАНИЯ ЭФФЕКТИВНЫХ ПРОГРАММНЫХ СИСТЕМ

Глава содержит описание верификации предложенной методологии проектирования ЭПС на примере практически значимых вычислительно-сложных задач.

В разделе 4.1 приводится реализация методологического подхода для задачи BLAS 3 - приведения матрицы общего вида к Хессенберговой форме методом Хаусхолдера и двухдиагональной формам. Получены строчно-ориентированная и однопроходная схема выполнения двустороннего преобразования Хаусхолдера. Значительно снижено время выполнения преобразования, проанализирована энергоэффективность полученных вычислительных схем.

В разделе 4.2 приводится реализация методологического подхода для задачи BLAS 2 – матрично-векторного умножения, получено значительное ускорение работы ПС, реализующей данное преобразование.

В разделе 4.3 приводится реализация методологического подхода для задач обработки числовых последовательностей, в частности преобразований одномерной и двумерной свертки.

В 4.4 приводятся основные результаты и выводы по главе 4.

4.1 Реализация методологического подхода для задачи приведения матрицы общего вида к ХФ методом Хаусхолдера

Типовые задачи линейной алгебры, такие, как решение СЛАУ, поиск собственных значений и собственных векторов, сингулярное разложение и другие, широко применяются как при решении прикладных задач, так и при выполнении фундаментальных исследований, как этапы задач более высокого уровня. Используемые в них типовые матричные преобразования, такие как треугольные разложения, приведение к ХФ, обращения матриц,

часто составляют основную вычислительную сложность в процессе решения, особенно когда речь идет о матрицах большой размерности, что типично для современных высокоточных и масштабных математических моделей. Отдельно выделим некоторые двусторонние преобразования, широко используемые в линейной алгебре [152]. Преобразование (4.1) называют преобразованием подобия.

$$B = F^{-1} A F \quad (4.1)$$

У подобных матриц совпадает ряд характеристик, в частности ранг, определитель, собственные значения и некоторые другие. Унитарным двусторонним преобразованием называют преобразование (4.2).

$$B = U A T \quad (4.2)$$

Здесь U и T - унитарные матрицы. Данное преобразование также сохраняет ряд характеристик исходной матрицы, в частности значения ее сингулярных чисел. В силу того, что преобразования (4.1) и (4.2) сохраняют значения собственных и сингулярных чисел соответственно, они часто используются в алгоритмах их поиска. Сами задачи хорошо известны [152]. Задача поиска собственного разложения заключается в поиске векторов и чисел, удовлетворяющих условию (4.3).

$$A v = \lambda v \quad (4.3)$$

Вектор v в данном случае называется собственным вектором линейного оператора A , число λ – собственным значением. Задача поиска сингулярного разложения заключается в поиске векторов и чисел, удовлетворяющих условию (4.4).

$$A^* u = \lambda v \quad (4.4)$$

Такие векторы u и v называются левым и правым сингулярным вектором, соответствующим сингулярному числу λ , A^* - матрица, эрмитово – сопряженная к A . В вычислительной математике получили распространение итерационные алгоритмы, в которых собственные значения и векторы вычисляются как результаты последовательности преобразований, сходящихся к искомым величинам.

Сингулярным разложением матрицы является разложение вида (4.5).

$$A = U \Sigma V^* \quad (4.5)$$

Здесь A – исходная матрица, U и V – унитарные матрицы, Σ – матрица, у которой элементы, лежащие на главной диагонали, являются сингулярными числами. В вещественном случае также используются ортогональные и транспонированные матрицы.

Современные ВС, как правило, имеют параллельную архитектуру, при этом параллелизм может реализовываться на разных уровнях этих архитектур, могут использоваться разные виды параллелизма, зачастую несколько в рамках одной архитектуры. Соответственно, проводятся работы по адаптации известных алгоритмов реализации численных методов для эффективной реализации на современных параллельных архитектурах, либо по созданию полностью альтернативных алгоритмов, оценивается эффективность реализации алгоритмов на различных параллельных архитектурах (например, [94, 122]).

Для повышения эффективности подобных преобразований были применены предложенные модели и методы, в частности методы МУСК и MBV в рамках моделей улучшения показателей эффективности ПС и оценки эффективности проектируемых ПС на ГВС с различной архитектурой. Выбор именно данных преобразований обусловлен их широким распространением при решении научных и инженерных задач, при этом указанные разложения (и особенно – лежащие в их основе матричные преобразования) часто являются базовыми для других методов более высокого уровня, выполняются многократно, используются для матриц большой размерности, что приводит к большим временным затратам.

В процессе решения задачи о собственных числах часто исходная матрица общего вида приводится к «почти треугольной» форме, отличающейся от треугольной наличием ненулевой диагонали под или над главной диагональю[193]. Матрицы такого вида называют соответственно верхней или нижней Хессенберговой матрицей. Существует конечная

последовательность преобразований, приводящая матрицу произвольного вида к ХФ с сохранением ее собственных значений. Подобные матрицы, наряду с трехдиагональными, широко применяются в алгоритмах поиска собственных значений, усиливающих главную диагональ, например, в QR-алгоритме или методе Якоби, сокращение времени приведения матриц к таким формам сокращает и время решения задачи о собственных значениях в целом.

В случае сингулярного разложения матриц общего вида одним из наиболее эффективных методов является приведение матрицы к двухдиагональному виду с помощью преобразований (4.5), где в данном случае Σ является двухдиагональной матрицей, с дальнейшим применением эффективных алгоритмов вычисления сингулярного разложения двухдиагональной матрицы. Соответственно, повышение эффективности методов приведения матриц к двухдиагональному виду приводит к уменьшению времени выполнения сингулярного разложения в целом.

И в (4.1), и в (4.2) в качестве операторов используются унитарные, а в вещественном случае – ортогональные матрицы. В качестве базового метода приведения матрицы общего вида к ХФ будет использован метод отражения Хаусхолдера.

$$A = QRQ^T \quad (4.6)$$

Здесь A – исходная матрица, R – матрица Хессенберга или двухдиагональная матрицы, Q – ортогональная матрица, в данном случае представляющая собой произведение матриц отражения Хаусхолдера, задаваемых каждая своим вектором отражения. Матрица Хаусхолдера является симметричной и совпадает со своей обратной матрицей.

В работе [32] рассматривается разложение матриц с использованием преобразований Хаусхолдера на параллельных ВС с общей памятью. При умножении на матрицу отражения справа имеем QR-разложение, при умножении слева получаем LQ-разложение, в работе [32] рассмотрены оба варианта. При QR-разложении вектор отражения определяется исходя из

обнуляемых столбцов исходной матрицы, а при LQ – соответственно исходя из обнуляемых строк. LQ-разложение оказывается в данном случае более предпочтительным из-за меньшей вероятности кеш-промахов. Так как данные в кеш-память подгружаются строками, то более эффективными с точки зрения использования кеш-памяти оказываются алгоритмы, в которых обработка ведется вдоль строк матрицы, как это было показано в главе 3. Таким алгоритмом является вариант LQ – разложения, в отличие от варианта QR – разложения, в котором обработка данных ведется вдоль столбцов матрицы.

В случае выполнения двустороннего преобразования (4.1) или (4.2) ситуация не столь однозначна, т.к. необходимо выполнять умножение с двух сторон для сохранения собственных или сингулярных значений.

Для приведения матрицы размерности $n \times n$ к верхней ХФ необходимо выполнить $(n-2)$ шага. В рамках каждого шага необходимо:

- вычислить элементы вектора отражения по элементам очередного столбца исходной матрицы;
- умножить исходную матрицу на матрицу отражения справа;
- умножить исходную матрицу на матрицу отражения слева.

Алгоритм выполнения k -го шага преобразования можно записать следующим образом (4.7).

$$\begin{aligned}
 s_k &= -\operatorname{sign}(a_{k+1,k}) \left(\sum_{i=k+1}^n a_{ik}^2 \right)^{\frac{1}{2}}, \\
 \varphi_k &= (s_k^2 - a_{k+1,k})^{-1} \\
 \mathbf{u}_k^T &= (0, \dots, 0, a_{k+1,k} - s_k, a_{k+2,k}, \dots, a_{nk}) \\
 \lambda_j &= \varphi_k \mathbf{u}_k^T \mathbf{a}_j \Big|_{j=\overline{k,n}} \\
 \mathbf{a}_j &= \mathbf{a}_j - \lambda_j \mathbf{u}_k \Big|_{j=\overline{k,n}} \\
 \lambda'_j &= \varphi_k \mathbf{b}_j \mathbf{u}_k \Big|_{j=\overline{k,n}} \\
 \mathbf{b}_j &= \mathbf{b}_j - \lambda'_j \mathbf{u}_k^T \Big|_{j=\overline{k,n}}
 \end{aligned} \tag{4.7}$$

Здесь $\mathbf{a}$ - столбцы исходной матрицы, $\mathbf{b}$ - ее строки, выражение $j = \overline{k, n}$ указывает на диапазон изменения индексов при обработке строк и столбцов матрицы на текущем шаге преобразования. Процесс выполнения преобразования проиллюстрирован на рисунке 4.1.

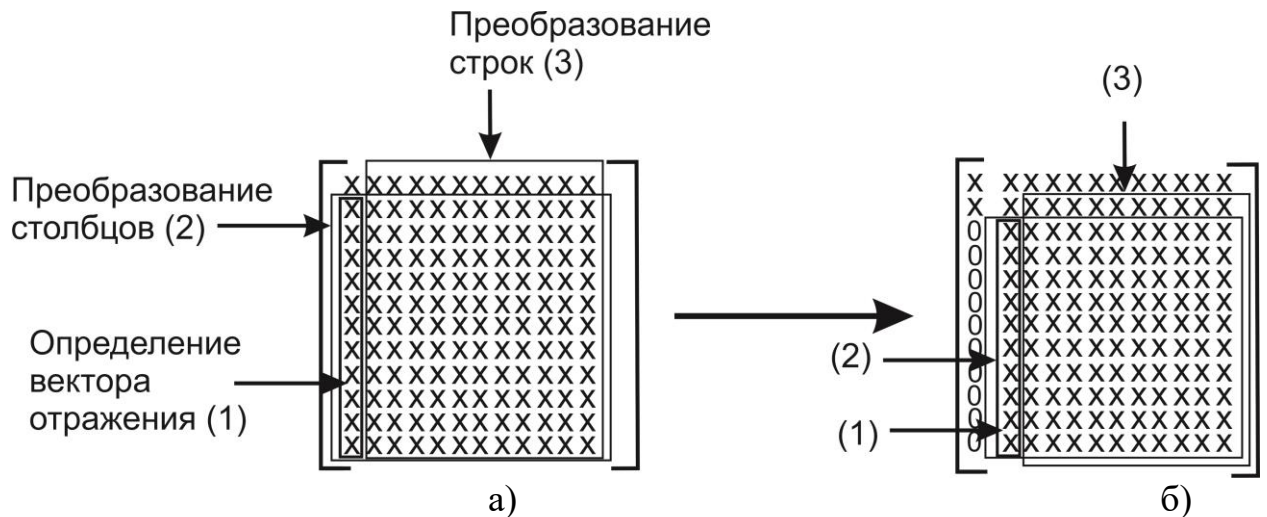


Рисунок 4.1 – Процесс приведения матрицы к верхней ХФ.

На рисунке 4.1 представлены два шага преобразования. На первом шаге, имеющем в (4.7) номер $k=0$, обнуляется первый столбец матрицы. Для этого:

- определяется вектор отражения из значений элементов первого столбца матрицы, находящихся под главной диагональю;
- осуществляется умножение на матрицу умножения справа, в результате которого обнуляются элементы первого столбца (рисунок 4.1 б)), изменяются $(n-1)$ элементов всех остальных столбцов, начиная с первого;
- осуществляется умножение на матрицу умножения слева, в результате чего изменяются $(n-1)$ элементов всех строк матрицы.

После выполнения $(n-2)$ шагов матрица будет приведена к верхней ХФ.

В соответствии с законом Амдала ускорение программы с параллельной реализацией любого алгоритма ограничено частью программы, не подлежащей распараллеливанию. Для рассматриваемого алгоритма, очевидно, что, во-первых, три этапа преобразования на каждом его шаге (рисунок 4.1) выполняются строго последовательно. Кроме того, первый этап

(определение вектора отражения) выполняется последовательно, т.к. его распараллеливание не даст ощутимого ускорения из-за невысокой вычислительной сложности. Во-вторых, из рисунка 4.1 видно, что первые два этапа выполняются как операции со столбцами матрицы, в то время как матрицы хранятся по строкам, из-за этого генерируется большое число кеш-промахов [168]. Эти два обстоятельства существенно снижают эффективность программ, основанных на распараллеливании алгоритма (4.7).

Для приведения матрицы к нижней ХФ на каждом шаге преобразования необходимо выполнить следующие действия.

$$\begin{aligned}
 s_k &= -\operatorname{sign}(a_{k,k+1}) \left(\sum_{i=k+1}^n a_{ki}^2 \right)^{\frac{1}{2}}, \\
 \varphi_k &= (s_k^2 - s_k a_{k,k+1})^{-1} \\
 \mathbf{u}_k^T &= (0, \dots, 0, a_{k,k+1} - s_k, a_{k,k+2}, \dots, a_{kn}) \\
 \lambda_j &= \varphi_k \mathbf{b}_j \mathbf{u}_k \Big|_{j=\overline{k,n}} \\
 \mathbf{b}_j &= \mathbf{b}_j - \lambda_j \mathbf{u}_k^T \\
 \lambda'_j &= \varphi_k \mathbf{u}_k^T \mathbf{a}_j \Big|_{j=\overline{k,n}} \\
 \mathbf{a}_j &= \mathbf{a}_j - \lambda'_j \mathbf{u}_k
 \end{aligned} \tag{4.8}$$

На рисунке 4.2 представлен процесс приведения матрицы к нижней ХФ.

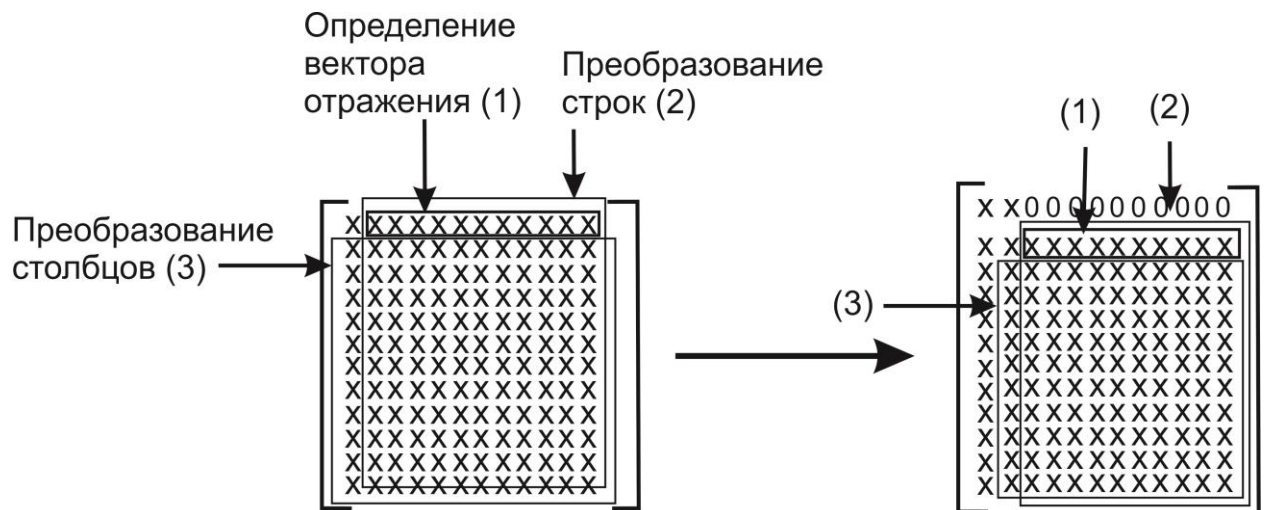


Рисунок 4.2 – Процесс приведения матрицы к нижней ХФ

В целом выполняются те же действия, с учетом следующих замечаний:

- вектора отражения формируются на базе элементов строк матриц;

- сначала преобразуются строки, затем столбцы матриц.

Для приведения матрицы к двухдиагональной форме необходимо выполнить большее число операций. Это связано с тем, что элементы вектора отражения вычисляются на обоих этапах преобразования. Алгоритм приведения матрицы в верхней двухдиагональной форме можно записать следующим образом (4.9).

Последовательность действий в целом похожа на последовательность действий при приведении к верхней ХФ, за исключением:

- при умножении справа для вычисления столбцов $\mathbf{a}_j$ при формировании вектора отражения используются элементы столбца под главной диагональю вместе с элементом главной диагонали;
- при умножении слева формируется новый вектор отражения на основе элементов строки, лежащих справа от главной диагонали;
- общее число шагов в данном случае составляет (n-1);
- при умножении слева выполняется меньшее число шагов (n-2), на что указывает обозначение $k < n-1$, указывающее на номера шагов, в рамках которых необходимо выполнять данные преобразования.

$$\begin{aligned}
 s_k &= -\operatorname{sign}(a_{k,k}) \left(\sum_{i=k}^n a_{ik}^2 \right)^{\frac{1}{2}}, \\
 \varphi_k &= (s_k^2 - s_k a_{k,k})^{-1} \\
 \mathbf{u}_k^T &= (0, \dots, 0, a_{k,k} - s_k, a_{k+1,k}, \dots, a_{nk}) \\
 \lambda_j &= \varphi_k \mathbf{u}_k^T \mathbf{a}_j \Big|_{j = \overline{k, n}} \\
 \mathbf{a}_j &= \mathbf{a}_j - \lambda_j \mathbf{u}_k \Big|_{j = \overline{k, n}} \\
 s'_k &= -\operatorname{sign}(a_{k,k+1}) \left(\sum_{i=k+1}^n a_{ki}^2 \right)^{\frac{1}{2}}, \\
 \varphi'_k &= (s_k'^2 - s'_k a_{k,k+1})^{-1} \\
 \mathbf{u}'_k^T &= (0, \dots, 0, a_{k,k+1} - s'_k, a_{k,k+2}, \dots, a_{kn}) \\
 \lambda'_j &= \varphi'_k \mathbf{b}_j \mathbf{u}'_k \Big|_{j = \overline{k, n}} \\
 \mathbf{b}_j &= \mathbf{b}_j - \lambda'_j \mathbf{u}'_k \Big|_{j = \overline{k, n}}
 \end{aligned} \Bigg\| \quad k < n-1 \tag{4.9}$$

На рисунке 4.3 изображены блок–схемы алгоритмов приведения матриц к ХФ и двухдиагональной форме.

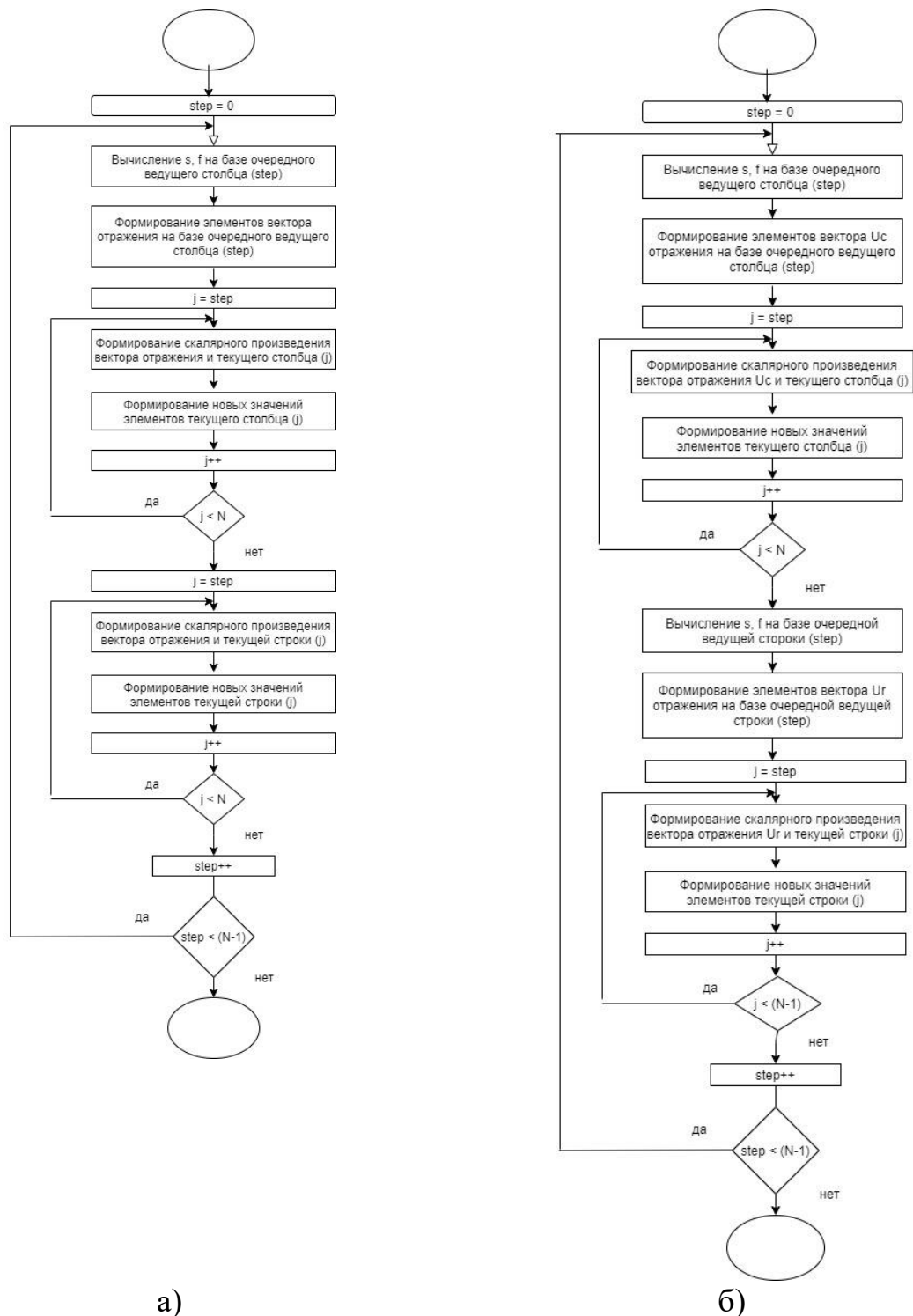


Рисунок 4.3 - Блок – схемы алгоритмов приведения матриц к Хессенберговой (а) и двухдиагональной (б) формам

Так как преобразования в целом похожи, сосредоточимся на приведении матрицы к ХФ, в дальнейшем распространив полученные выводы на алгоритм приведения матрицы к двухдиагональной форме.

В обоих алгоритмах, представленных на рисунках 4.1 и 4.2, осуществляются одни и те же преобразования над столбцами и строками матрицы, меняется только порядок выполнения действий. Поэтому данные этапы на ГВС будут выполняться за одно и то же время. Существенным отличием этих алгоритмов является схема формирования вектора отражения. С этой точки зрения более предпочтительным оказывается алгоритм приведения матрицы к нижней ХФ, представленный на рисунке 4.2, т.к. формирование вектора отражения осуществляется на основе элементов строк матрицы, в отличие от алгоритма, представленного на рисунке 4.1, где вектор отражения формируется на основе элементов столбцов. Однако, в силу того, что доля определения элементов вектора отражения в общем объеме вычислений достаточно мала, оба алгоритма будут иметь схожую вычислительную эффективность. Рассмотрим алгоритмы, представленные на рисунках 4.1 и 4.2 с точки зрения увеличения их эффективности на ГВС с иерархической структурой памяти. Можно выделить несколько узких мест:

- в данных алгоритмах присутствуют три последовательных этапа на каждом шаге преобразования; одним из этапов является вычисление элементов вектора отражения; однако существуют реализации, в которых осуществляется предварительное вычисление элементов вектора отражения для следующего шага преобразования, описанные, например, в [35].

- при умножении на матрицу отражения справа осуществляется обработка столбцов матрицы; это существенно снижает эффективность по сравнению с вариантом вычислений, основанным на обработке строк матрицы.

4.1.1 Реализация МУСК для базового преобразования Хаусхолдера

Рассмотрим модернизацию данных алгоритмов, направленную на повышение эффективности, связанную с ускорением обработки столбцов исходной матрицы.

В соответствии с (4.7) в процессе приведения исходной матрицы к верхней ХФ для выполнения очередного шага преобразования необходимо вычислить элементы вектора отражения, затем выполнить умножение на матрицу Хаусхолдера справа и слева.

При умножении на матрицу отражения справа, в процессе которого обнуляется очередной столбец исходной матрицы, выполняются следующие действия:

- определяются элементы вектора отражения на основе значений соответствующего столбца исходной матрицы;
- вычисляются новые значения столбцов матрицы; столбцы могут обрабатываться как последовательно, так и параллельно, т.к. в данном случае отсутствуют информационные зависимости.

При этом в процессе пересчета значений столбцов выполняются следующие вычисления:

- определяются значения скалярных произведений столбцов и вектора отражения;
- на основе полученных значений скалярных произведений вычисляются новые значения элементов столбцов.

В силу того, что данные обрабатываются вдоль столбцов, генерируется достаточно большое значение кеш-промахов, что сильно замедляет процесс вычислений.

Применим МУСК к данному преобразованию. Вначале разобьем множество операций обращения к памяти на подмножества в соответствии со способом обращения к памяти, который оказывает влияние на вероятность появления кеш-промаха и оценим количество операций в каждом подмножестве.

Как было показано в главе 3, вероятность кеш-промаха оценивается значением вероятности того, что при следующем считывании данные в кеш-памяти отсутствуют. Эту вероятность можно оценивать от 1 в случае, когда гарантированно наступит кеш-промах, до $1/m$, где m – число элементов в

кеш–линии. Выделим подмножества операций обращения к памяти с равным показателем вероятности кеш-промаха и для каждого подмножества оценим число операций (таблица 4.1).

Таблица 4.1. Подмножества операций обращений к памяти, классическая схема.

Подмножество	Вероятность кеш-промаха	Число операций	Примечание
H1	$\alpha_V=1$	$(n - k - 1)$	элементы исходной матрицы в процессе вычисления модуля ведущего столбца, формирования вектора отражения
H2	$\alpha_{VR} = 0$	$(n - k - 1)$	элементы вектора отражения в процессе его формирования
H3	$\alpha_C = 1$	$2 * (n - k - 1)$ $* (n - k - 1)$	умножение на матрицу Хаусхолдера справа, доступ вдоль столбцов
H4	$\alpha_R=1/m$	$2 * (n - k - 1)$ $* (n - k - 1)$	умножение на матрицу Хаусхолдера слева, доступ вдоль строк
H5	$\alpha_{VEC.} = 0$	$2 * (n - k - 1)$ $* (n - k - 1).$	обращения к элементам вектора отражения при умножении на матрицы Хаусхолдера

В рамках одного шага преобразования можно выделить четыре цикла:

- вычисление нормы ведущего столбца (1);
- вычисление элементов вектора отражения (2);
- обработка столбцов (3);

- обработка строк (4);

Циклы (1) и (2) имеют меньшую вычислительную сложность по сравнению с (3) и (4), при этом циклы (1), (2) и (3) выполняются вдоль столбцов матрицы, цикл (4) – вдоль строк. Т.к. матрицы хранятся в памяти по строкам, циклы обработки строк имеет больший потенциал в контексте оптимизации работы с кеш–памятью и векторизации. В случае векторизации это связано с тем, что данные, используемые в цикле (4) расположены в памяти непрерывно, поэтому могут загружаться в SIMD–регистры операциями векторной загрузки. В циклах (1), (2) и (3) данные не расположены непрерывно, поэтому при векторной обработке будут загружаться в SIMD-регистры поэлементно, что отрицательно влияет на производительность.

Первый цикл, связанный с вычислением нормы ведущего столбца, генерирует кеш–промах при каждом обращении к памяти, т.к. данные выбираются вдоль столбцов матрицы. Вероятность кеш–промаха в данном случае фактически равна единице. При выполнении второго цикла, связанного с вычислением элементов вектора отражения, данные обрабатываются также вдоль столбцов, однако, при разумном размере исходной матрицы вероятность кеш–промаха в данном случае фактически равна нулю, т.к. все необходимые для вычислений элементы матрицы были подгружены в кеш–память в процессе выполнения предыдущего цикла. Если же размер матрицы станет настолько большим, что один ее столбец целиком не помещается в кеш–память, вероятность кеш–промаха при выполнении второго цикла станет расти и фактически станет равной единице. Третий цикл также реализуется вдоль столбцов, вероятность кеш-промаха примет равной единице, четвертый цикл выполняется вдоль строк, вероятность кеш-промаха примет равной $1/m$.

Вероятность кеш–промахов будет зависеть от соотношения размеров кеш–памяти и обрабатываемой матрицы и в самом простом случае может быть оценена следующим образом:

- если размер матрицы меньше размера кеш-памяти, то кеш-промахи возникнут при первых обращениях к элементам матрицы, при последующих проходах вдоль элементов матрицы они уже будут размещены в кеш-памяти;
- если размер матрицы больше размера кеш-памяти, кеш-промахи будут генерироваться при каждом проходе.

При оценке вероятностей в таблице 4.1 принимаются следующие допущения:

- размер матрицы достаточен для того, чтобы целиком не поместиться в кеш-память;
- элементы вектора отражения после формирования загружаются только из кеш-памяти;
- значение параметра α_C принимаем равным 1, хотя оно может быть несколько меньшим, особенно при выполнении шагов с большим номером и с учетом параметра α_V ;
- значение параметра α_R также может быть несколько меньшим, чем $1/m$;
- значение параметра α_{VEC} принимаем равным 0, хотя оно может увеличиться вплоть до $1/m$ при большой размерности задачи.

Арифметико-логические операции учитывать не будем, т.к. используется только перестановочная оптимизация, не влияющая на общее число арифметико-логических операций. Данный параметр является инвариантом в существующем контексте, и оценивать его не будем.

Оценим время работы с памятью. Необходимо получить выражение, оценивающее время выполнения программы без учета времени выполнения арифметико-логических операций, то есть время, которое будет затрачено на работу с памятью в процессе работы программы. В рамках одного шага преобразования выражение (3.29) примет вид:

$$T_M = \sum_{i=1}^5 (\alpha_i * K_i * t_{mem} + (1 - \alpha_i) * K_i * t_{cache}), \quad (4.10)$$

где α_i - соответствующая вероятность кеш-промаха;

K_i - соответствующее число операций.

Подставим в выражение (4.10) оцененные выше значения вероятностей кеш-промахов и числа операций в каждой группе.

$$T_{MEM,CL} = (\alpha_V + \alpha_{VR} + 2 * L * (\alpha_C + \alpha_R + \alpha_{VEC})) * L * t_{mem} + ((1 - \alpha_V) + (1 - \alpha_{VR}) + 2 * L * ((1 - \alpha_C) + (1 - \alpha_R) + (1 - \alpha_{VEC}))) * L * t_{cache} =$$

$$(1 + 2 * L * (1 + 1/m)) * L * t_{mem} + (1 + 2 * L * ((1 - 1/m) + 1)) * L * t_{cache}$$
(4.11)

где $L = (n - k - 1)$.

Анализ полученного выражения позволяет сделать следующие выводы:

- параметры α_V , α_{VR} , α_{VEC} не оказывают значимого влияния на совокупное время работы с памятью, т.к. в исследованных группах операций невысокая вероятность кеш-промаха, а также из-за того, что число операций в первых двух группах пропорционально размерности матрицы;
- наибольшее влияние на совокупное время работы с памятью оказывают параметры α_C и α_R , т.к. число операций в этих группах пропорционально квадрату размерности матрицы;
- группа операций α_R хорошо оптимизирована, т.к. вероятность кеш-промаха составляет $1/m$, следовательно, для повышения эффективности реализации алгоритма необходимо оптимизировать группу операций α_C , предлагаемый вариант оптимизации приведен на рисунке 4.4.

На рисунке 4.4 показана последовательность преобразования строк и столбцов исходной матрицы в рамках одного шага преобразования приведения к верхней ХФ. Слева приведена схема традиционного алгоритма на базе преобразования Хаусхолдера. Она крайне неэффективна из-за необходимости обработки столбцов матрицы, в рамках которой происходит большое число кеш-промахов для матриц, хранящихся вдоль строк.

Изменить ситуацию можно, модифицировав порядок обработки элементов матрицы. Справа приведена схема RОН, в рамках которой обработка столбцов также ведется вдоль строк.

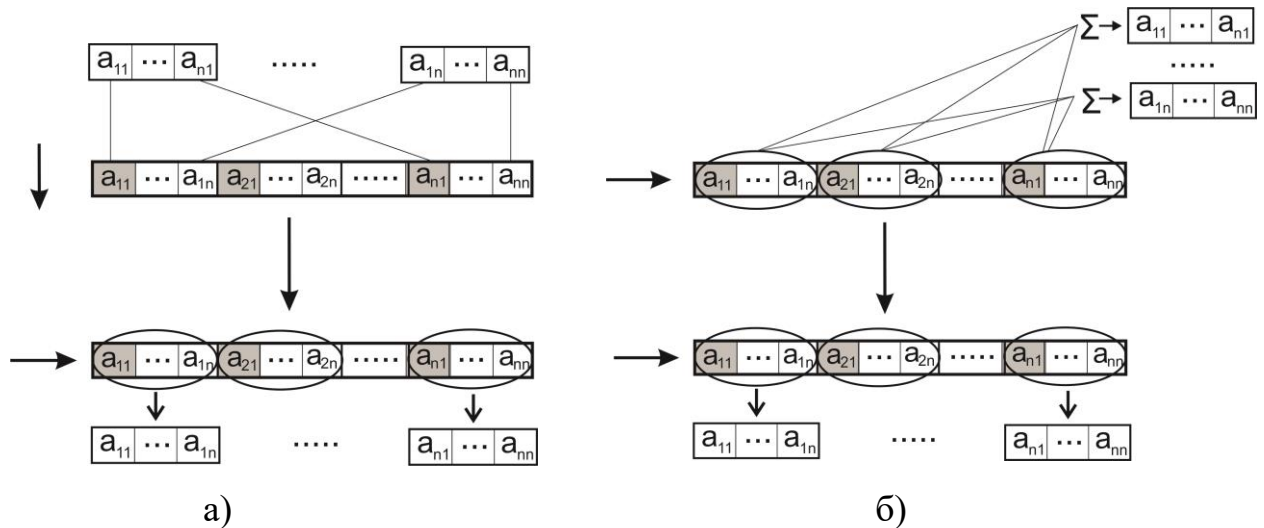


Рисунок 4.4. Традиционная (а) и строчно-ориентированная (Row-Oriented Householder – ROH) (б) схемы выполнения одного шага двустороннего преобразования Хаусхолдера

Предварительно вычисляются скалярные произведения столбцов исходной матрицы A и вектора отражения u_k^T . В результате формируется вектор скалярных произведений sc , вычисления осуществляются с учетом масштабного коэффициента φ_k . Дальнейшая последовательность действий соответствует традиционной схеме выполнения преобразования. Вычисляются значения элементов столбцов матрицы, но в данном случае вычисление выполняется вдоль строк, что является гораздо более эффективной стратегией кеширования данных. Затем производится умножение на матрицу Хаусхолдера с другой стороны, в результате чего вычисляются элементы строк полученной матрицы. Эту схему будем называть строчно-ориентированной схемой выполнения двустороннего преобразования Хаусхолдера.

В преобразовании значение параметра α_c изменяется и становится равным $1/m$. В результате оценка времени работы с памятью примет следующий вид.

$$T_{MEM,ROH} = \left(1 + 2 * L * \frac{2}{m}\right) * L * t_{mem} + \left(1 + 2 * L * \left(\left(2 - \frac{2}{m}\right) + 1\right)\right) * L * t_{cache}. \quad (4.12)$$

С учетом того, что $m > 1$, также $t_{mem} > t_{cache}$ получаем $T_{MEM,ROH} < T_{MEM,CL}$, т.е. полученный алгоритм более эффективно работает с подсистемой памяти. Данный процесс можно записать следующим образом (4.13). Здесь и в дальнейшем алгоритм формирования вектора отражения аналогичен (4.7) при приведении к верхней Хессенберговой форме и аналогично (4.8) при приведении к нижней Хессенберговой форме. Поэтому, за исключением случаев, где это действительно необходимо, приводиться не будет.

$$\begin{aligned} sc &= \varphi_k \mathbf{u}_k^T \mathbf{A} \\ \mathbf{D} &= \text{diag}(sc_0, sc_1, \dots, sc_n) \\ \mathbf{U}[:, j] &= [\mathbf{u}_k] \\ \mathbf{SCU} &= \mathbf{UD} \\ \mathbf{b}_j &= \mathbf{b}_j - \mathbf{scu}_j | j = \overline{k, n} \\ \lambda'_j &= \varphi_k \mathbf{b}_j \mathbf{u}_k^T | j = \overline{k, n} \\ \mathbf{b}_j &= \mathbf{b}_j - \lambda'_j \mathbf{u}_k^T | j = \overline{k, n} \end{aligned} \quad (4.13)$$

Перед обработкой столбцов вычисляется вектор скалярных произведений sc как произведение вектора отражения $\mathbf{u}_k^T$ на исходную матрицу $\mathbf{A}$ с учетом масштабного коэффициента φ_k . Далее осуществляется вычисление новых значений столбцов матрицы, причем вычисление осуществляется построчно, что гораздо эффективнее с точки зрения использования кеш-памяти. Необходимо отметить, что в (4.13) используются следующие обозначения: $\mathbf{D}$ – диагональная матрица, содержащая на своей главной диагонали значения скалярных произведений sc , $\mathbf{U}$ – матрица, столбцы которой составлены из вектора отражения, $\mathbf{scu}$ – строки матрицы $\mathbf{SCU}$, $\mathbf{b}$ – строки матрицы $\mathbf{A}$. Как видно из (4.13), при умножении матрицы, как справа, так и слева, обработка ведется построчно, что значительно

увеличивает эффективность программной реализации метода, снижая время выполнения программы.

Алгоритм приведения матрицы к нижней Хессенберговой форме с использованием строчно-ориентированной схемы обработки столбцов матрицы приведен в (4.14).

$$\begin{aligned}
 \lambda'_j &= \varphi_k \mathbf{b}_j \mathbf{u}_k \Big|_{j = \overline{k, n}} \\
 \mathbf{b}_j &= \mathbf{b}_j - \lambda'_j \mathbf{u}_k^T \\
 \mathbf{sc} &= \varphi_k \mathbf{u}_k^T \mathbf{A} \\
 \mathbf{D} &= \text{diag}(\mathbf{sc}_0, \mathbf{sc}_1, \dots, \mathbf{sc}_n) \\
 \mathbf{U}[:, j] &= [\mathbf{u}_k] \\
 \mathbf{SCU} &= \mathbf{UD} \\
 \mathbf{b}_j &= \mathbf{b}_j - \mathbf{scu}_j \Big|_{j = \overline{k, n}}
 \end{aligned} \tag{4.14}$$

В (4.13) и (4.14) не показано формирования вектора отражения, которое осуществляется аналогично (4.7) и (4.8) соответственно, т.е. на основании столбцов и строк матрицы соответственно.

Подобная строчно-ориентированная схема позволяет существенно повысить эффективность программной реализации алгоритма приведения матрицы к ХФ. Однако потенциал повышения производительности здесь не исчерпан.

Повторное применение метода показывает неэффективность необходимости двух проходов по матрице, в результате, при определенной размерности ко второму проходу элементы матрицы начинают вытесняться из кеш-памяти (данный паттерн доступа к данным подробно рассмотрен в главе 3). По-прежнему вначале обрабатываются столбцы, затем – строки. К моменту окончания обработки столбцов для больших матриц первые строки будут вытеснены из кеш-памяти, что приводит к дополнительным кеш-промахам.

Соответственно, алгоритм (4.13) имеет хорошие характеристики с точки зрения кеширования данных и векторизации вычислений. Традиционно программные реализации двустороннего преобразования

Хаусхолдера реализуются в два прохода: обработка столбцов и обработка строк. Причем в данном алгоритме третий и четвертый циклы, связанные с обработкой столбцов и строк исходной матрицы, выполняются вдоль строк, поэтому вероятность кеш–промаха примем равной $1/m$.

Снизить вероятность кеш–промахов можно, используя блочный метод реализации обоих строковых циклов, причем размер блоков необходимо выбирать таким образом, чтобы они целиком помещались в кеш–память.

Анализируя выражение (4.13), можно сделать следующий вывод. Так как вычисление новых значений столбцов исходной матрицы при умножении на матрицу Хаусхолдера справа ведется вдоль строк, после выполнения очередной итерации цикла становятся известными соответствующие значения всех столбцов матрицы, которые находятся в одной строке. Так как в ходе дальнейших преобразований, значения данных элементов матрицы не меняются, можно сказать, что очередная строка готова к выполнению умножения на матрицу Хаусхолдера слева. Это позволяет изменить порядок выполнения операций. Вместо того, чтобы сначала полностью закончить умножение на матрицу Хаусхолдера справа, затем приступить к умножению на нее слева, поступим следующим образом. На каждой итерации сначала вычислим элементы очередной строки матрицы в рамках умножения справа, затем вычислим ее элементы в рамках умножения слева. В ходе дальнейших вычислений элементы данной строки меняться не будут. Преобразование можно записать следующим образом.

$$\begin{aligned}
 \mathbf{v} &= \varphi_k \mathbf{u}_k^T \mathbf{A} \\
 \mathbf{D} &= \text{diag}(v_0, v_1, \dots, v_n) \\
 \mathbf{U}[:, j] &= [\mathbf{u}_k] \\
 \mathbf{W} &= \mathbf{UD} \\
 \left. \begin{aligned} \mathbf{b}_j &= \mathbf{b}_j - \mathbf{w}_j \\ \lambda'_j &= \varphi_k \mathbf{b}_j \mathbf{u}_k \\ \mathbf{b}_j &= \mathbf{b}_j - \lambda'_j \mathbf{u}_k^T \end{aligned} \right| j = \overline{k, n}
 \end{aligned} \tag{4.15}$$

В (4.15) не показано формирования вектора отражения, которое осуществляется аналогично (4.7). В данном случае фактически организован один цикл, в рамках которого каждая строка обрабатывается дважды в рамках одной итерации. Это приводит к тому, что при первой обработке строки по-прежнему будут генерироваться кеш-промахи со средней вероятностью $1/m$, однако при повторной обработке строки все ее элементы уже будут находиться в кеш-памяти. Средняя вероятность кеш-промахов в данном случае снижается до $1/2m$. Так как два последних цикла имеют сложность порядка $O(n^2)$, а первые два – порядка $O(n)$, можно сказать, что в целом для всего преобразования значительно снижена вероятность кеш-промахов.

В (4.15) присутствует еще один цикл, обрабатывающийся вдоль столбцов. Это цикл формирования вектора скалярных произведений (первая строка в (4.15)). Однако и этот цикл можно достаточно легко изменить и вести обработку вдоль строк матрицы (4.16).

$$\begin{aligned} \mathbf{v}_K' &= \sum_{i=0}^n u_i \mathbf{b}_i \\ \mathbf{v} &= \varphi_k \mathbf{v}_K' \end{aligned} \quad (4.16)$$

Здесь $\mathbf{v}$ – вектор скалярных произведений, u_i – элемент вектора отражения, $\mathbf{b}_i$ – строка матрицы. Модификации, описанные в (4.15), также с учетом (4.16), позволяют значительно повысить эффективность программной реализации двустороннего преобразования Хаусхолдера за счет улучшения алгоритма кеширования. Данную схему выполнения преобразования в дальнейшем будем называть однопроходной схемой выполнения двустороннего преобразования Хаусхолдера (рисунок 4.5). Теперь данные обрабатываются в последовательности первый столбец – первая строка – второй столбец – вторая строка и т.д. Такой подход существенно снижает вероятность кеш-промахов.

Видно, что полученная схема SPH использует гораздо более эффективный алгоритм кеширования по сравнению с традиционной схемой выполнения преобразования, так как:

- обработка элементов матрицы осуществляется только вдоль строк, в отличие от традиционной схемы отсутствует обработка элементов матрицы вдоль столбцов;

- в процессе обработки осуществляется однократный проход вдоль элементов матрицы (в традиционной схеме - 2 прохода).

Ниже представлен псевдокод реализации данного алгоритма. Процедура Hessenberg используется для приведения матрицы общего вида к верхней ХФ, далее вводятся следующие обозначения: A – входная матрица, N – размерность квадратной матрицы, *Householder* – процедура вычисления вектора отражения, u_s - вектор отражения, g – нормировочный коэффициент. *SSH* – процедура выполнения одного шага одностороннего преобразования Хаусхолдера (Single-Sided Householder). Жирным шрифтом выделены матрицы и вектора, в скобках указаны диапазоны изменения индексов.

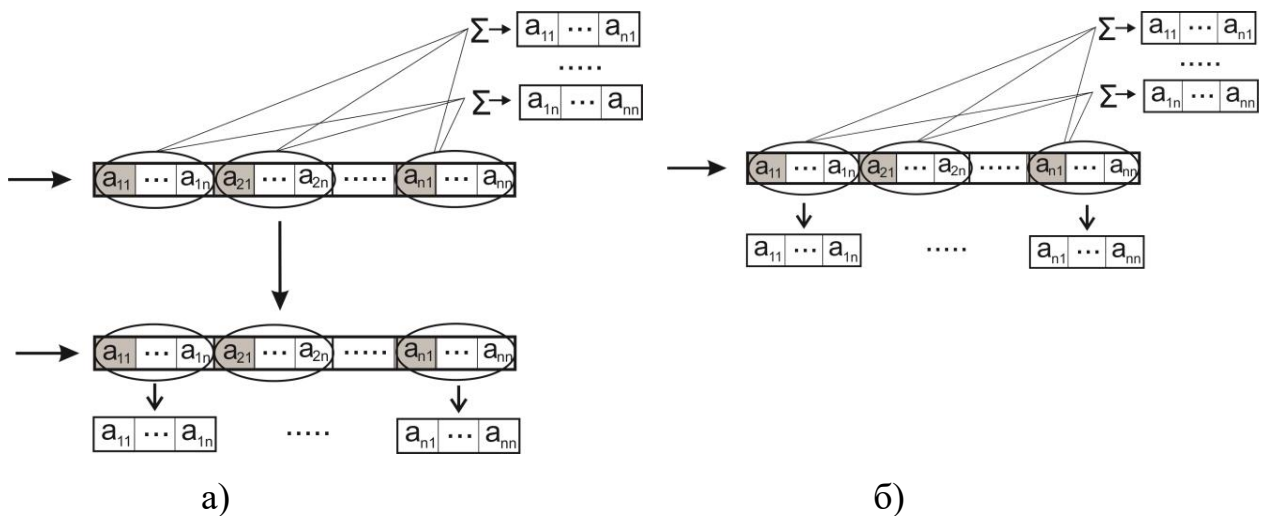


Рисунок 4.5 - SOH (a) и Single-Pass Householder (SPH) (б) схемы выполнения одного шага двустороннего ПХ

```

1:Hessenberg (A, N)
2:   for s = 1 to N-2 do
3:       (g, us) = Householder (As, s);
4:       SSH(A, N, s, g, us)
5:   end for
6: end Hessenberg

```

```

1: SSH(A, N, s, g,  $\mathbf{u}_s$ )
2:    $\mathbf{v}_s^T[s:N] = g \mathbf{u}_s^T[s+1:N] \mathbf{A}[s+1:N, s:N]$ 
3:   for r = s to N do
4:      $\mathbf{A}[r, s : N] \mathrel{:=} \mathbf{u}_s[r] \mathbf{v}_s^T[s : N]$ 
5:      $\mathbf{vr}_s = g \mathbf{A}[r, s+1:N] \mathbf{u}_s[s+1:N]$ 
6:      $\mathbf{A}[r, s+1:N] \mathrel{:=} \mathbf{vr}_s \mathbf{u}_s^T[s+1:N]$ 
7:   end for
8: end SSH

```

Для оценки полученных результатов получены усредненные значения вероятностей кеш-промахов (таблица 4.2), являющиеся одной из метрик ПС в рамках предложенной модели оценки эффективности ПС на ГВС с различной архитектурой.

Таблица 4.2 - Расчет усредненных значений кеш-промахов.

Classic	N_{MEM}	$\sum_{s=1}^{N_s} \left(1 + 2 * L_s * \left(1 + \frac{1}{m}\right)\right) * L_s$
	N_{CACHE}	$\sum_{s=1}^{N_s} \left(1 + 2 * L_s * \left(\left(1 - \frac{1}{m}\right) + 1\right)\right) * L_s$
ROH	N_{MEM}	$\sum_{s=1}^{N_s} \left(1 + 2 * L_s * \frac{2}{m}\right) * L_s$
	N_{CACHE}	$\sum_{s=1}^{N_s} \left(1 + 2 * L_s * \left(\left(2 - \frac{2}{m}\right) + 1\right)\right) * L_s$
SPH	N_{MEM}	$\sum_{s=1}^{N_s} \left(1 + 2 * L_s * \frac{1}{m}\right) * L_s$
	N_{CACHE}	$\sum_{s=1}^{N_s} \left(1 + 2 * L_s * \left(\left(2 - \frac{1}{m}\right) + 1\right)\right) * L_s$

Усредненные значения вероятности кеш-промаха для трех схем преобразования представлены на рисунке 4.6. Данные значения слабо зависят от размерности решаемой задачи, а для алгоритма SPH значение данного параметра имеет наименьшее значение.

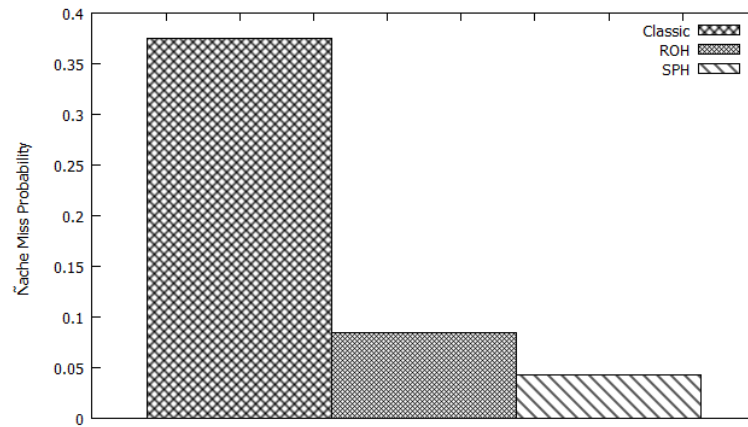


Рисунок 4.6 - Усредненное значение вероятности кеш-промаха (метрика).

Были проведены вычислительные эксперименты на узле кластера на базе микропроцессора Xeon E5-2650 v3(x2) 2.3 GHz. В результате были получены зависимости времени выполнения программных реализаций различных вычислительных схем (рисунок 4.7) и ускорения полученных вычислительных схем относительно алгоритма на базе классического преобразования Хаусхолдера (рисунок 4.8). Обе полученных характеристики являются метриками ПС в рамках предложенной модели оценки эффективности проектируемых ПС на ГВС с различной архитектурой.

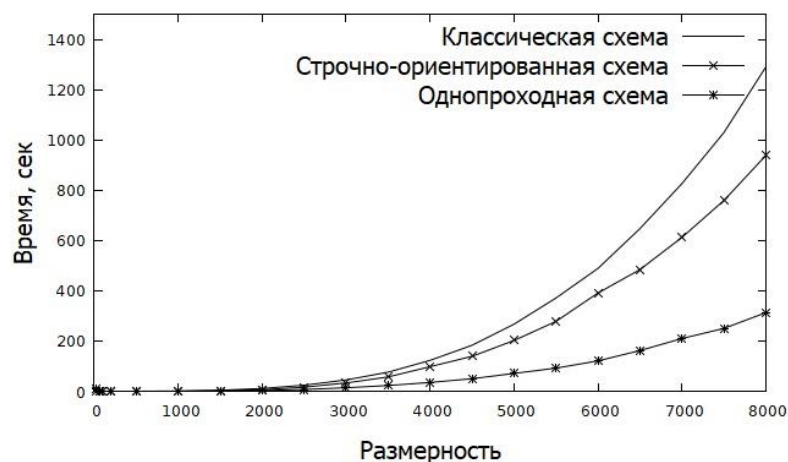


Рисунок 4.7 - Время выполнения программных реализаций различных вычислительных схем реализации двустороннего преобразования Хаусхолдера (метрика)

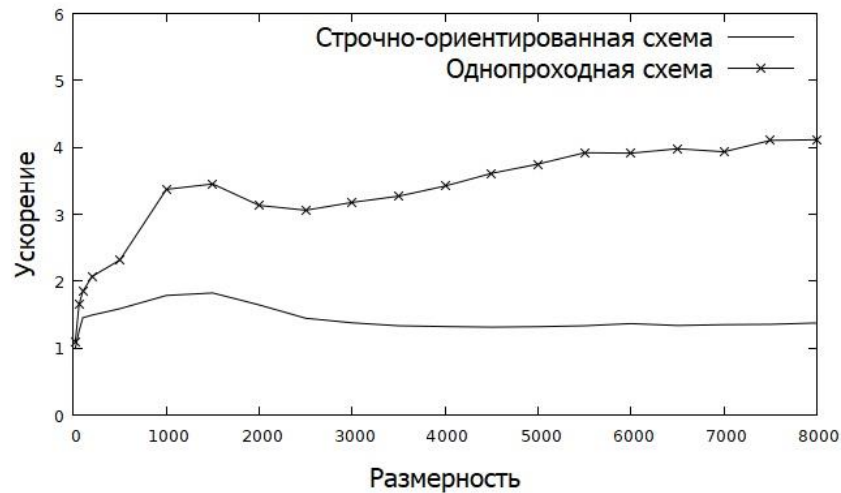


Рисунок 4.8 - Ускорения полученных программных реализаций различных вычислительных схем двустороннего преобразования Хаусхолдера относительно классической реализации (метрика)

Как и ожидалось в соответствии с рассчитанными метриками, самой быстрой оказалась однопроходная схема реализации преобразования, самой медленной – реализация классической схемы двустороннего преобразования Хаусхолдера.

4.1.2 Энергоэффективность полученных вычислительных схем

Для оценки энергоэффективности ПС используются различные методы. Так, в работе [11] описаны модели, позволяющие прогнозировать энергоэффективность ПС, которые используют как статические показатели исходного кода, так и динамические измерения времени выполнения, мощности и энергопотребления. Авторы исследования заявляют, что после обучения моделей можно использовать только статические показатели для прогнозирования. В работе [74] описана методика оценки энергоэффективности ПС на основе профилирования. Стоит отметить, что существуют и другие методики оценки и прогнозирования энергоэффективности ПС.

Для повышения энергоэффективности исследователи предлагают различные оптимизации ПС, связанные с использованием авторских

оптимизаций [78], оптимизирующих возможностей компиляторов [46], другие оптимизации.

Большое количество работ посвящено проблемам снижения энергопотребления вычислительных кластеров, обработке больших данных [89], использованию облачных технологий [72]. При этом разрабатываются методы для динамического управления узлами с целью снижения энергопотребления, разрабатываются стратегии контроля состояния узлов. Так, в работе [69] рассмотрены методы снижения энергопотребления единого компьютерного кластера за счет гибких стратегий управления состояниями узлов (их пробуждением или выключением) и порядком выполнения ожидающих задач.

Часто возникает необходимость оценить энергопотребление ВС в процессе выполнения проектируемой ПС, возможно, различных участков проектируемой программы. Существующие инструментальные средства могут быть использованы лишь в некоторой степени, т.к. либо не имеют API для взаимодействия с другим программным обеспечением, либо не позволяют оценить энергопотребление отдельных ядер многоядерного МП и других элементов ВС.

Решением здесь может стать Running Average Power Limit (RAPL). RAPL представляет собой программный интерфейс, разработанный компанией Intel, который предоставляет разработчикам ПС счетчики энергопотребления МП и некоторых подсистем ВС. RAPL не является аналоговым инструментом, в нем используется оригинальная программная модель, которая для оценки энергопотребления использует встроенные счетчики производительности. Точность этих измерений зависит от типа МП, однако, разработчики утверждают, что в современных процессорах измерения осуществляются достаточно точно. Стоит также отметить, что RAPL доступен только в системах с процессорами, поддерживающими данную технологию. В первую очередь она поддерживается процессорами

Intel, AMD также предоставляет совместимый интерфейс для некоторых своих систем.

RAPL позволяет осуществлять измерение энергопотребления в разных доменах (domains): собственно, сам МП, DRAM и встроенный GPU. Данная возможность появилась, начиная с архитектуры Sandy Bridge, далее развивалась во всех последующих поколениях МП Intel. На рисунке 4.9 представлена структура доменов, поддерживаемых RAPL.

RAPL поддерживает следующие домены для измерения и ограничения:

- Package (PKG) – энергопотребление всего сокета;
- PowerPlane0 (PP0) – энергопотребление ядер сокета;
- PowerPlane1 (PP1) – энергопотребление GPU (встроенная графика);

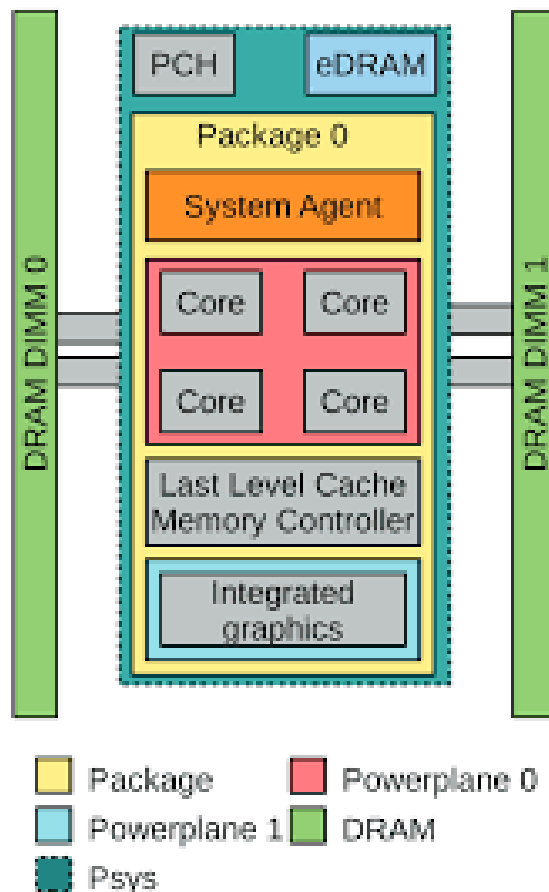


Рисунок 4.9 – Структура доменов, поддерживаемых RAPL [71]

- DRAM – энергопотребление DRAM, подключенной к встроенному контроллеру памяти;

- Psys – новый домен, представленный в Intel Skylake, относящийся к энергопотреблению однокристальной системы (SOC – System-on-a-Chip); согласно рисунку 4.9 включает в себя, в том числе потребление PCH (Platform Controller Hub – устройство, выполняющее функции южного и часть функций северного моста), eDRAM (embedded DRAM – встраиваемая DRAM).

Необходимо отметить, что не все МП поддерживают весь набор доменов.

Для получения данных об энергопотреблении с использованием RAPL используются регистры MSR (Model-Specific Register). Это специальные регистры, назначение которых на разных моделях процессоров может быть различным (Model-Specific - модельно-зависимые). Для чтения и записи этих регистров используются привилегированные команды RDMSR и WRMSR. В состав этих регистров входят регистры для подсчета времени, мониторинга производительности, контроля отладки и других специфических функций. Доступ к привилегированным командам из прикладных программ невозможен, поэтому операционные системы предоставляют механизмы доступа к данным регистрам. Например, входящий в состав Linux MSR Kernel Module создает псевдо-файл `"/dev/cpu/x/msr"` (с уникальным x для каждого процессора).

Были проведены вычислительные эксперименты, получены зависимости потребляемой энергии от размерности задачи для всех вычислительных схем реализации двустороннего преобразования Хаусхолдера. Графики потребляемой энергии оказались похожи на графики времени выполнения программ, наиболее энергоэффективной оказалась самая быстрая схема, наименее – самая медленная. Так, например, при обработке матриц размерностью 8000 реализация однократной схемы привело к потреблению 28,7 КДж, в то время как реализация классической схемы преобразования на той же размерности привело к потреблению 109 КДж, что в 3,8 раза больше.

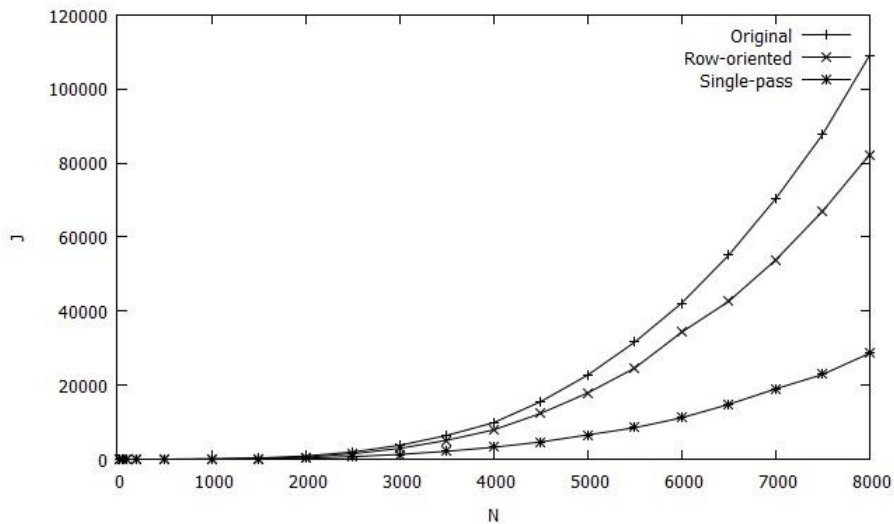


Рисунок 4.10 – Зависимость энергии, потребляемой в процессе выполнения программ от размерности решаемой задачи.

Ответ на вопрос, какова количественная зависимость между сокращением времени выполнения программы и сокращением энергопотребления, не так очевиден. На рисунках 4.11 и 4.12 представлены графики мощности, потребляемой в процессе выполнения программ.

Оба графика имеют одинаковую структуру, на них представлены данные, соответствующие различным доменам RAPL:

PKG – мощность, потребляемая сокетом в целом (Package);

DRAM – мощность, потребляемая DRAM;

PKG Idle – мощность, потребляемая сокетом микропроцессора в режиме «простоя»;

DRAM Idle - мощность, потребляемая DRAM в режиме «простоя».

Данные графики подтверждают предположения о том, что оптимизированные версии ПС потребляют больше энергии. Таким образом, при реализации классической схемы потребляется 49 Вт против 52 Вт в однопроходной схеме (домен PKG) и 37 Вт против 41 Вт (домен DRAM). Однако благодаря значительному сокращению времени выполнения программ снижается энергопотребление.

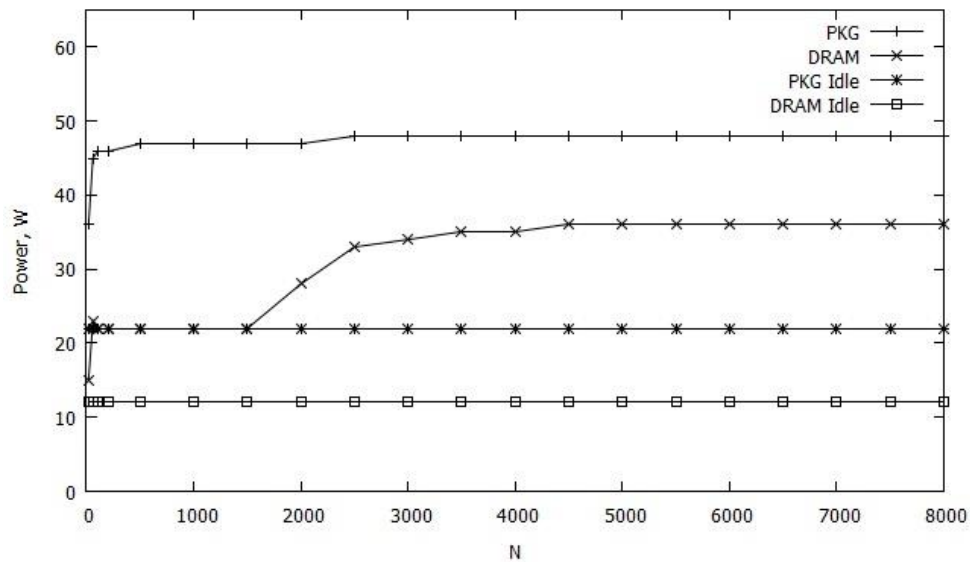


Рисунок 4.11 – Зависимость мощности, потребляемой в процессе выполнения программ, реализующей классическую схему выполнения преобразования, от размерности решаемой задачи.

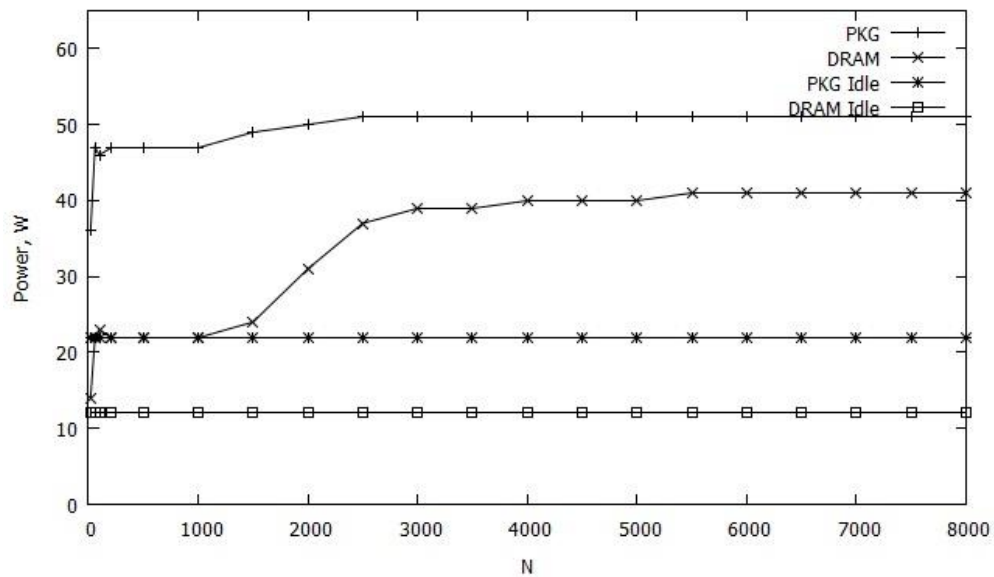


Рисунок 4.12 – Зависимость мощности, потребляемой в процессе выполнения программ, реализующей однократную схему выполнения преобразования, от размерности решаемой задачи.

Под режимом «простоя» здесь понимается режим работы МП без запуска пользовательских программ. В данном режиме на МП выполняется ряд системных процессов, фактически не потребляющих ресурсы (в соответствии с данными утилиты «top»). Анализируя рисунки 4.11 и 4.12, очевидно, что в режиме «простоя» CPU потребляет мощность 22 Вт и 12,5 Вт

для доменов PKG и DRAM соответственно. Следует также отметить, что графики на рисунке 4.10 получены путем суммирования значений PKG и DRAM.

Анализируя полученное ускорение, приведенное на рисунке 4.8, можно отметить, что обе реализации демонстрируют хороший рост производительности при размерности матрицы 1000-1500, который затем несколько снижается. Эти результаты связаны с характеристиками МП, в частности, с объемом кеш-памяти, который для используемого МП составляет 25 Мб (интеллектуальный кеш L3). Наибольшее ускорение достигается при размерах примерно 1750 при таком объеме кеш-памяти и двойном типе (длина 8 байт), который использовался для моделирования. Это связано с тем, что матрицы большей размерности больше не помещаются в кеш, при меньших размерах оптимизированные версии увеличивают ускорение за счет лучшего использования кешей L1/L2. При размерах, превышающих 1750, все три версии начинают генерировать промахи в L3, что несколько снижает значение ускорения. Далее ускорение продолжает снижаться и для строчно-ориентированной схемы, стабилизируется на уровне около 1,35. Для однопроходной схемы ускорение начинает снова увеличиваться и достигает значения 4,1 при значении размерности 8000. Прирост производительности в этом случае обеспечивается эффективным алгоритмом кеширования данных для L2, в то время как базовая схема генерирует все большее число промахов в кеше.

Далее было рассчитано соотношение энергопотребления базовой схемы и схемы однопроходного преобразования Хаусхолдера (рисунок 4.13). Для расчета были использованы данные рисунка 4.10 (соотношение общей энергии для доменов PKG и DRAM, рассчитанное для базовой схемы и схемы однопроходного преобразования). В целом, график похож на график ускорения для однопроходной схемы, при этом несколько отличается конкретными значениями.

Зависимость, показанная на рисунке 4.14, иллюстрирует это различие. Здесь показано отношение значения ускорения к значению уменьшения энергопотребления для однократной схемы. Можно видеть, что это соотношение немного превышает единицу и варьируется в пределах 1,05–1,1 для разных размеров матрицы.

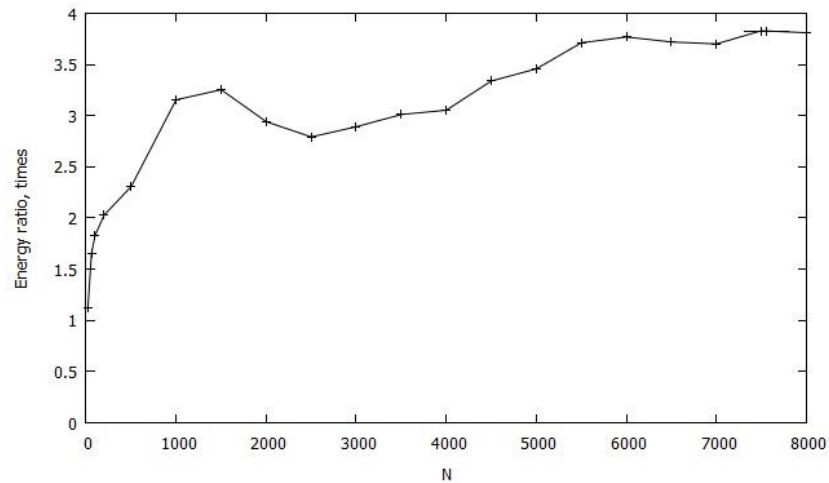


Рисунок 4.13 - Соотношение энергопотребления базовой и однократной схем преобразования

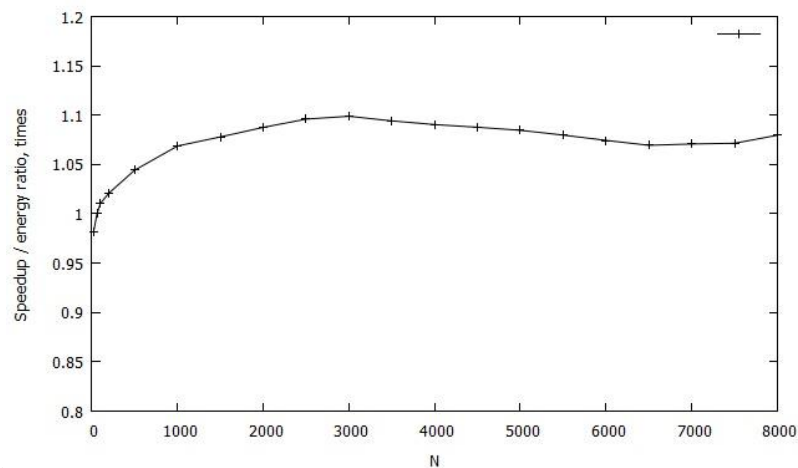


Рисунок 4.14 - Соотношение ускорения и снижения энергопотребления при однократной схеме

То есть абсолютное значение ускорения для конкретного значения размерности немного превышает значение снижения энергопотребления, хотя эти значения близки. В ходе экспериментов была также проведена оценка снижения «дополнительного энергопотребления», которое мы можем

определить как разницу между текущим потреблением энергии и потреблением энергии в режиме «холостого хода» (4.17).

$$E_{diff} = (PKG + DRAM) - (PKG\ Idle + DRAM\ Idle) \quad (4.17)$$

Анализ показал, что динамика этого параметра полностью совпадает с динамикой энергопотребления в целом. Кроме того, графики соотношения «дополнительного энергопотребления» и общего энергопотребления практически совпадают для двух исследованных вычислительных схем. Пусть w_1 и w_2 — «дополнительная» мощность, потребляемая двумя ПС, а $K = (PKG\ Idle + DRAM\ Idle)$ — мощность, потребляемая CPU в режиме «ожидания». Тогда

$$\frac{w_1}{w_2} - \frac{(K + w_1)}{(K + w_2)} = \frac{\left(\frac{w_1}{w_2} - 1 \right)}{\left(1 + \frac{w_2}{K} \right)} \quad (4.18)$$

Ранее (рисунки 4.11 и 4.12) было показано, что разница между энергопотреблением для базовой и однопроходной схем не так велика (не превышает 10%). Соответственно, значение выражения (4.18), показывающее разницу между соотношением «дополнительной» и общей потребляемой мощности для целевой ВС, невелико.

4.1.3 Реализация MBV для базового преобразования Хаусхолдера

Еще одним направлением ускорения программ помимо собственно распараллеливания вычислений является векторизация. Рассмотрим векторизацию цикла, возникающего при умножении на матрицу Хаусхолдера слева (4.19).

$$\begin{aligned} \lambda'_j &= \varphi_k \mathbf{b}_j \mathbf{u}_k \\ \mathbf{b}_j &= \mathbf{b}_j - \lambda'_j \mathbf{u}_k^T \end{aligned} \Big|_{j = \overline{k, n}} \quad (4.19)$$

Рассмотрим выполнение 0 шага преобразования (4.19) ($k = 0$ в (4.7)). На нулевом шаге обработка начинается с первого элемента вектора отражения, также с первых элементов строк матрицы. Проанализируем обработку строк:

- первая строка; элементы строки и вектора отражения выровнены одинаково, обработка начинается с первого элемента, необходим пролог цикла для обработки первых трех элементов, далее векторизованная часть, далее – эпилог цикла для обработки последних элементов;

- вторая строка; элементы строки и вектора отражения выровнены по-разному; в зависимости от способа обработки начала цикла, может быть сформирован пролог цикла, далее следует векторизованная часть, далее – эпилог цикла.

При выполнении любого другого, например, 5 шага преобразования наблюдается так же картина, т.е. необходимо формирование пролога и эпилога цикла, присутствует различное выравнивание элементов вектора отражения и строк матрицы.

Современные оптимизирующие компиляторы осуществляют автоматическую векторизацию кода программ. При этом схема векторизации циклов в общем случае выглядит примерно следующим образом:

- формируется не векторизованный пролог цикла, после выполнения которого текущие элементы обрабатываемых массивов находятся по выровненным адресам;

- векторизуется средняя часть цикла;

- формируется эпилог, обрабатывающий последние элементы, число которых меньше ширины SIMD – регистра;

- если в цикле используются несколько массивов, выровнены они по-разному, компилятор выбирает один из них, обработка данного массива будет осуществляться с использованием выровненных считываний из памяти, обработка остальных – с использованием невыровненных обращений; пролог и эпилог цикла формируется на основе выбранного массива.

Компиляция исходной программы, реализующей приведение матрицы к верхней ХФ, осуществлялась компилятором Intel C++ Compiler. В результате автоматической векторизации был получен значительный прирост производительности, о чем будет сказано ниже.

Однако компилятор не может преодолеть описанные выше ограничения. Повысить эффективность векторизации можно, решив проблемы с формированием неэффективных пролога и эпилога цикла, а также выполнив взаимное выравнивание элементов вектора отражения и элементов строк матрицы.

Рассмотрим обработку нулевой строки. Элементы вектора отражения и элементы нулевой строки выровнены одинаково, что позволяет использовать эффективные операции выровненного считывания из памяти.

Несколько повысить эффективность реализации данного цикла можно, избавившись от пролога. Если учесть, что все элементы вектора отражения с индексами, меньшими $(N + 1)$ для шага с номером N в соответствии с алгоритмом (4.7) равны нулю и данные нулевые значения записывать в необходимые элементы V при его формировании, обработку можно начинать с ближайшего меньшего выровненного индекса. В случае нулевого шага обработку можно начинать с нулевого элемента, на пятом шаге – с четвертого,

Ранее был представлен метод повышения эффективности векторизации вычислений (MBV) в рамках модели оценки эффективности проектируемых ПС для ГВС с различной архитектурой. Метод был применен для повышения эффективности реализации однопроходной схемы преобразования Хаусхолдера в рамках задачи приведения матрицы общего вида к верхней ХФ. Основное внимание было уделено векторизации циклов, реализуемых вдоль строк.

Рассмотрим векторизацию цикла (4.19). Для этого необходимо проанализировать три случая:

- размерность матрицы кратна ширине SIMD-регистра; например, в случае использования типа данных `double` и технологии AVX – кратна 4-ем, при использовании AVX и `float` – кратна 8-ми и т.д.;
- размерность матриц четная, но не кратна ширине SIMD-регистра, например, 10 для AVX и `double`;
- размерность матриц нечетная.

Для каждого из этих случаев будет использоваться свой алгоритм векторизации. Проанализируем каждый из этих случаев по отдельности.

На рисунке 4.15 приведено выполнение шага преобразования для кратной размерности матрицы. Для примера используется матрица 12×12 .

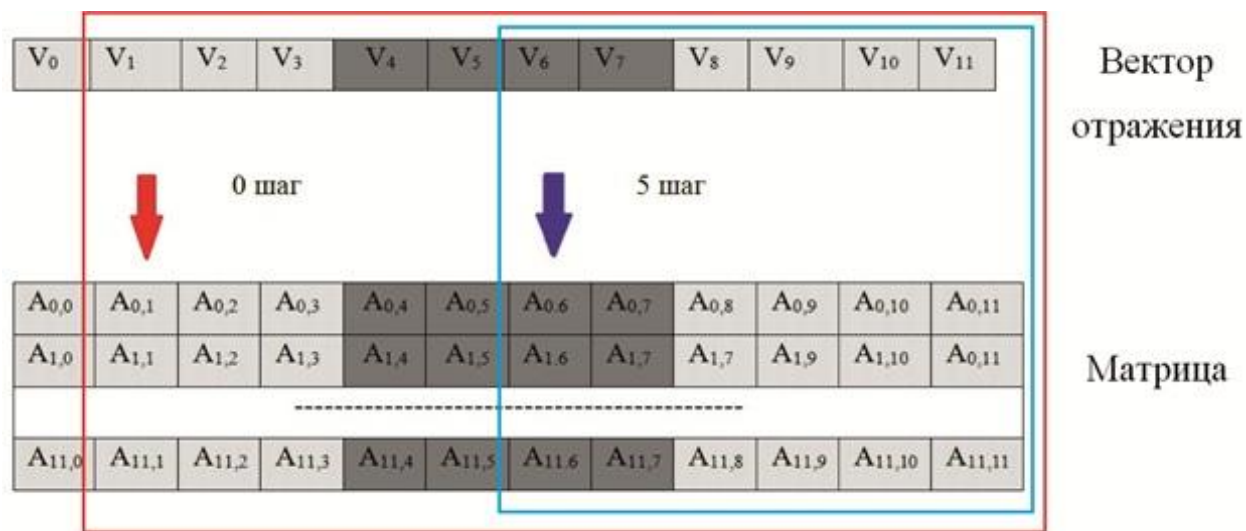


Рисунок 4.15 - Выполнение шага преобразования для матрицы с размерностью, кратной ширине SIMD-регистра

Анализируя рисунок 4.15, можно сделать следующие выводы:

- элементы вектора отражения и строк исходной матрицы выровнены одинаково;
- эпилог цикла отсутствует в силу того, что длина строки кратна ширине SIMD-регистра (в данном случае 4).

Рассмотрим исходный не векторизованный код обработки строк.

```
for(int row = 0; row < N; row++)
{
    double scalar = 0;
```

```

for(int i = (step + shift); i < N; i++)
    scalar += matr[row * N + i] * vect[i];
scalar *= gamma;
for(int i = (step + shift); i < N; i++)
    matr[row * N + i] -= scalar * vect[i];
}

```

Для каждой строки матрицы сначала вычисляется скалярное произведение с вектором отражения, затем вычисляются новые значения элементов строки. В строке обрабатываются элементы, начиная с $(\text{step} + \text{shift})$, где step – номер шага преобразования, $\text{shift} = 1$ для приведения матрицы к ХФ. Очевидно, что обработка в общем случае начинается с невыровненного адреса.

Применим MBV к данному преобразованию.

Первый этап.

1. Определение характеристик векторизуемого цикла.

Ширина SIMD-регистров зависит от выбранной технологии векторизации и используемых типов данных. В рассматриваемом случае ширина равна 4 элементам, т.к. используется технология AVX (256 бит) и тип данных `double` (64 бита).

Под размером обрабатываемых матриц в направлении обработки `lda` понимаем количество элементов матрицы, хранящихся между элементами соседних строк с одинаковыми индексами, например между A_{00} и A_{10} . При этом для хранения матриц (многомерных массивов) используем способ, принятый в BLAS – плоское хранение, т.е. хранение в виде одномерного массива. Таким образом, т.к. в языке C++ матрицы хранятся по строкам, данный параметр в рассматриваемом случае равен длине строки, т.е. N (12 на рисунке 4.15).

Под смещением здесь понимаем расположение элемента данных в SIMD-регистре при последовательной загрузке элементов матрицы. Предположим, что первый элемент ($A_{0,0}$) матрицы расположен по

выровненному адресу, он имеет смещение 0, тогда элемент $A_{0,1}$ будет иметь смещение 1, элемент $A_{0,2}$ будет иметь смещение 2. Элемент $A_{0,4}$ будет опять иметь смещение 0 и т.д. Смещение начальных элементов зависит от lda и использованной технологии векторизации, влияние здесь оказывает ширина SIMD-регистра.

Смещения первых элементов обрабатываемых строк в данном случае будут совпадать (N кратно ширине SIMD-регистра) и зависеть от номера шага преобразования k (первый шаг имеет номер 0) (4.20)

$$S = ((k + 1) + rel_{0,0}) \% SIMD_{width} \quad (4.20)$$

где $SIMD_{width}$ – ширина SIMD-регистра в элементах базового типа;

$rel_{0,0}$ – смещение первого элемента матрицы;

% - операция получения остатка от деления.

2. Формирование необходимого количества ведущих векторов.

Число ведущих векторов зависит от количества несовпадающих смещений начальных элементов строк подматрицы и определяется lda и используемой технологией векторизации. Пусть

$$a = lda \% SIMD_{width} \quad (4.21)$$

Выразим a следующим образом.

$$a = k * (SIMD_{width} * 2^{-n}) \quad (4.22)$$

Тогда число ведущих векторов будет определяться следующим выражением.

$$nv = \begin{cases} 1, & \text{если } a = 0 \\ SIMD_{width}, & \text{если } (a \% 2) = 1 \\ 2^n, & \text{если } (a \% 2) = 0 \text{ и } a \neq 0 \end{cases} \quad (4.23)$$

В рассматриваемом случае необходим один вектор.

3. Определение длин ведущих векторов и их смещений относительно «естественного положения».

Для эффективной векторизации прологов и эпилогов циклов с использованием выровненного доступа к данным необходимо сформировать ведущие вектора таким образом, чтобы умножение соответствующих строк

подматрицы и ведущих векторов давало результат, аналогичный результату обычного умножения подматрицы на вектор.

Для этого ведущие вектора дополняются нулями слева и справа таким образом, чтобы его длина стала кратной ширине SIMD-регистра. Число нулей слева зависит от смещения первого элемента подстроки, число нулей справа зависит от смещения первого элемента подстроки и длины подстроки.

Число нулей слева для ведущего вектора в рассматриваемом случае равно смещению первого обрабатываемого элемента строке.

$$Z^L = S \quad (4.24)$$

Второй этап в основном связан с написанием программного кода: изменяются циклы доступа к данным в соответствии с полученными параметрами. Программную реализацию рассматриваемого варианта преобразования и использованием директив компилятора с применением MBV можно представить следующим образом.

```
alignVect = stepShift % 4;
for(int row = 0; row < N; row++)
{
    // Скалярное произведение
    double scalar = 0;
    #pragma vector aligned
        for(int i = stepShift - alignVect; i < N; i++)
            scalar += matr[row * N + i] * vect[i];
    scalar *= gamma;
    #pragma vector aligned
        for(int i = stepShift - alignVect; i < N; i++)
            matr[row * N + i] -= scalar * vect[i];
}
```

Это позволяет несколько повысить эффективность реализации данного цикла, т.к. пропала необходимость формировать пролог цикла. Т.к. все элементы вектора отражения с индексами, меньшими (step + shift), в

соответствии с алгоритмом (4.7) равны нулю и данные нулевые значения можно записывать в необходимые элементы V при его формировании, то обработку можно начинать с ближайшего меньшего выровненного индекса. В случае нулевого шага обработку можно начинать с нулевого элемента, на пятом шаге – с четвертого, Здесь alignVect и составляет смещение, рассчитанное с использованием (4.20).

Рассмотрим матрицы с четной размерностью, величина которой не кратна ширине SIMD-регистров. Для примера рассмотрим матрицу 10×10 (рисунок 4.16).

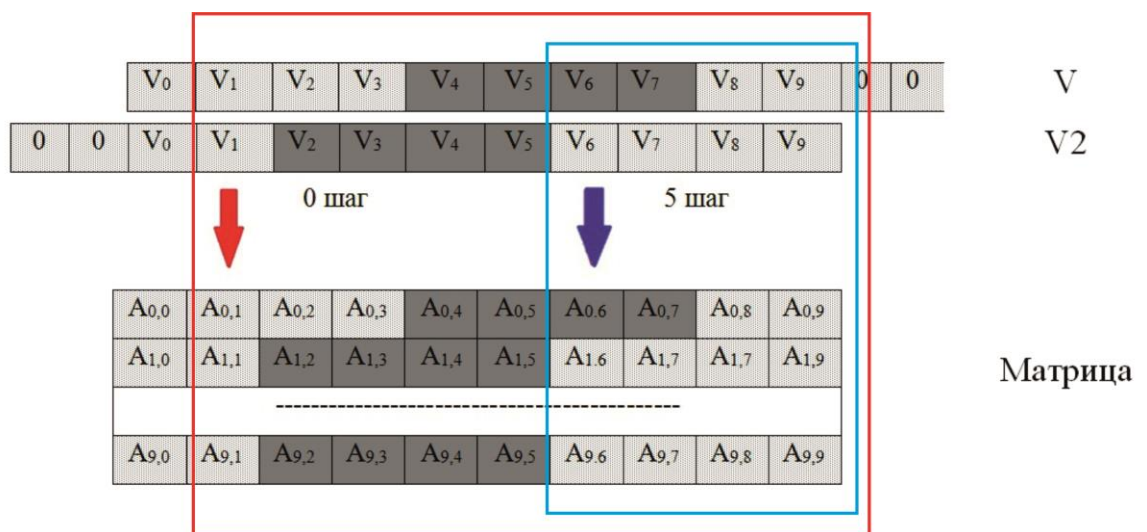


Рисунок 4.16 - Выполнение шага преобразования для матрицы с четной размерностью, не кратной ширине SIMD – регистра

Анализируя рисунок 4.16, можно сделать следующие выводы:

- элементы вектора отражения и строк исходной матрицы в общем случае выровнены по-разному;
- эпилог цикла необходим (например, элементы $A_{0,8}$ и $A_{0,9}$ первой строки).

Применим MBV к данному преобразованию.

Первый этап.

1. Определение характеристик векторизуемого цикла.

Ширина SIMD-регистров и размер обрабатываемых матриц в направлении обработки lda определяются аналогично.

Смещения первых элементов обрабатываемых строк в данном случае совпадать не будут, и зависят от номера шага преобразования k и номера строки матрицы. В общем случае данный параметр можно определить в соответствии с выражением (4.25).

$$S_i = ((i * lda + (k + 1)) \% SIMD_{width} + rel_{0,0}) \% SIMD_{width} \quad (4.25)$$

Пусть $rel_{0,0} = 0$, $lda = 10$ (рисунок 4.16). Тогда смещения вторых элементов строк ($j = 1$) будут равны:

$$- i = 0, S = ((0 * 10 + 1) \% 4 + 0) \% 4 = 1;$$

$$- i = 1, S = ((1 * 10 + 1) \% 4 + 0) \% 4 = 3;$$

$$- i = 2, S = ((2 * 10 + 1) \% 4 + 0) \% 4 = 1;$$

$$- i = 3, S = ((3 * 10 + 1) \% 4 + 0) \% 4 = 3;$$

Как видно, смещения могут принимать одно из двух значений, порядок смещений повторяется. В общем случае количество и значения смещений зависит от значения

$$L_{disp} = lda \% SIMD_{width} \quad (4.26)$$

В данном случае количество смещений равно двум (значения смещений 1 и 3).

2. Формирование необходимого количества ведущих векторов.

Число ведущих векторов в соответствии с (4.23) равно 2.

3. Определение длин ведущих векторов и их смещений относительно «естественного положения».

Число нулей слева для ведущих векторов в рассматриваемом случае равно смещению первого обрабатываемого элемента строке.

$$Z_i^L = S_i \quad (4.27)$$

Число нулей справа для i -го ведущего вектора определяется следующим образом:

$$Z_i^R = SIMD_{width} - (Z_i^L + (N - (k + 1))) \% SIMD_{width}. \quad (4.28)$$

Общая длина i -го вектора равна

$$L_i = Z_i^L + (N - (k + 1)) + Z_i^R \quad (4.29)$$

Программная реализация рассматриваемого варианта преобразования будет содержать отдельную обработку четных и нечетных строк и использованием директив компилятора.

Необходимость в прологе отпала так же, как и в предыдущем случае, т.е. путем смещения первого обрабатываемого элемента в сторону первого меньшего выровненного индекса. Однако такой подход сработает только для строк с четными номерами (0, 2 и т.д.), т.к. они выровнены так же, как и вектор отражения. Для строк нечетными номерами необходимо сформировать еще один вектор отражения **V2**, как показано на рисунке 4.16. В этом случае, например обработка элемента $A_{1,1}$ будет осуществляться совместно с $A_{0,8}$, $A_{0,9}$ и $A_{1,0}$. Но за счет наличия ведущих нулей в векторе **V2**, результат будет корректным.

В строках с четными номерами необходимо формировать эпилог, однако, за счет добавления нулей в конец вектора **V** результат так же будет корректен.

Для матриц с нечетной размерностью возможны два алгоритма. Они представлены на рисунках 4.17 и 4.18.

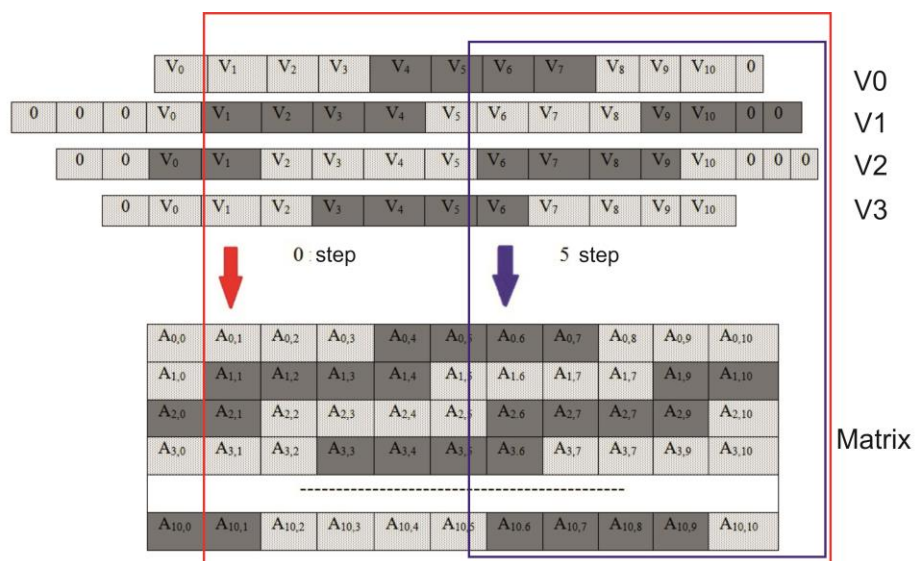


Рисунок 4.17 - Выполнение шага преобразования для матрицы с нечетной размерностью

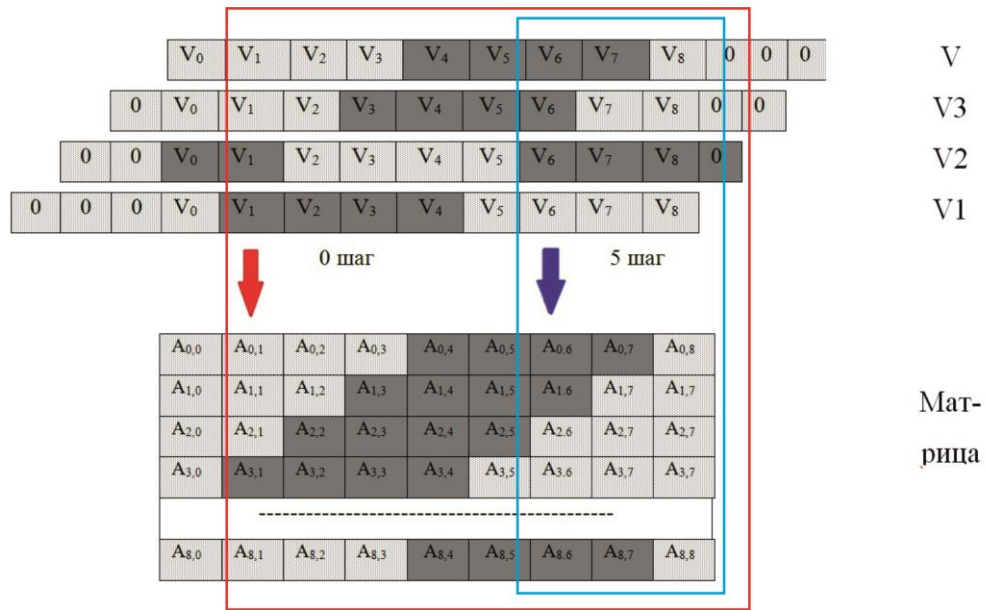


Рисунок 4.18 - Выполнение шага преобразования для матрицы с 9нечетной размерностью (вариант 2)

Применим MBV к данному преобразованию.

Первый этап.

1. Определение характеристик векторизуемого цикла.

Ширина SIMD-регистров и размер обрабатываемых матриц в направлении обработки lda определяются аналогично.

Смещения первых элементов обрабатываемых строк определяются в соответствии с (4.25).

Пусть $rel_{0,0} = 0$, $lda = 11$ (рисунок 4.17). Тогда смещения вторых элементов строк ($j = 1$) будут равны:

- $i = 0, S = ((0 * 11 + 1) \% 4 + 0) \% 4 = 1;$
- $i = 1, S = ((1 * 11 + 1) \% 4 + 0) \% 4 = 0;$
- $i = 2, S = ((2 * 11 + 1) \% 4 + 0) \% 4 = 3;$
- $i = 3, S = ((3 * 11 + 1) \% 4 + 0) \% 4 = 2;$
- $i = 4, S = ((4 * 11 + 1) \% 4 + 0) \% 4 = 1;$
- $i = 5, S = ((5 * 11 + 1) \% 4 + 0) \% 4 = 0;$
- $i = 6, S = ((6 * 11 + 1) \% 4 + 0) \% 4 = 3;$
- $i = 7, S = ((7 * 11 + 1) \% 4 + 0) \% 4 = 2;$

Как видно, смещения могут принимать одно из четырех значений, порядок смещений повторяется.

2. Формирование необходимого количества ведущих векторов.

Число ведущих векторов в соответствии с (4.23) равно 4.

3. Определение длин ведущих векторов и их смещений относительно «естественного положения».

Число нулей слева для ведущих векторов в рассматриваемом случае равно смещению первого обрабатываемого элемента строке (4.27).

Число нулей справа для ведущих векторов в рассматриваемом случае определяется в соответствии с (4.28).

Цикл (4.19) в результате применения указанных преобразований будет обрабатываться в соответствии с (4.30). В самом деле, сравним выражения, в соответствии с которыми осуществляется обработка. На рисунке 4.20 приведено графическое представление цикла (4.30) для примера с нечетной размерностью матрицы, приведенного на рисунке 4.17..

$$\left. \begin{aligned} \lambda'_j &= \varphi_k \mathbf{b}_j^m \mathbf{m}_k \\ \mathbf{b}_j^m &= \mathbf{b}_j^m - \lambda'_j \mathbf{m}_k^T \end{aligned} \right| j = \overline{k, n} \quad (4.30)$$

Отличие данного выражения от выражения (4.19) заключается в другом диапазоне итераций цикла при обработке отдельной строки. Строка $\mathbf{b}_j^m$ формируется с учетом смещения начальной и конечной итерации цикла, вектор $\mathbf{m}_k$ дополняется нулями. Начальная итерация смещается на величину align смещения первого обрабатываемого элемента в «текущем SIMD регистре» в сторону младших индексов. Конечная итерация смещается на величину offset, зависящую от размерности матрицы и номера обрабатываемой строки. Для нулевой строки матрицы (рисунок 4.19) offset=1.

Параметр “align” вычисляется следующим образом. Для шагов с номерами 0 – 2 обработка начинается с элемента $A_{0,0}$, поэтому для нулевого

шага $\text{align}=-1$, для первого $\text{align}=-2$, для второго $\text{align}=-3$. Для шагов с номерами 3 – 6 обработка начинается с $A_{0,4}$ и т.д.

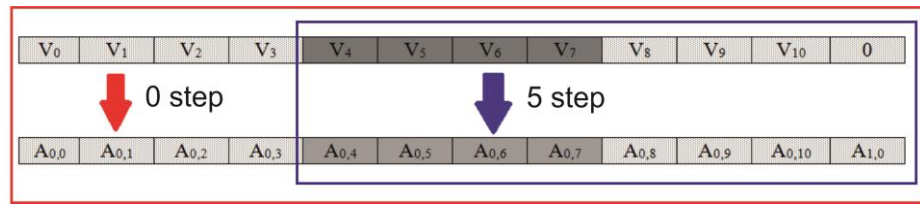


Рисунок 4.19 - Обработка нулевой строки матрицы в соответствии с (4.30)

Если учесть, что элементы вектора отражения с номерами, меньшими номера текущего шага, равны 0, очевидно, что обработка дополнительных элементов строки матрицы не повлияет на результат. При этом обработка всех элементов строки будет вестись в векторном режиме, отпадает необходимость в обработке пролога и эпилога цикла.

На рисунке 4.20 представлена обработка второй строки матрицы. Используется вектор отражения с другим выравниванием, при этом $\text{offset} = 3$.

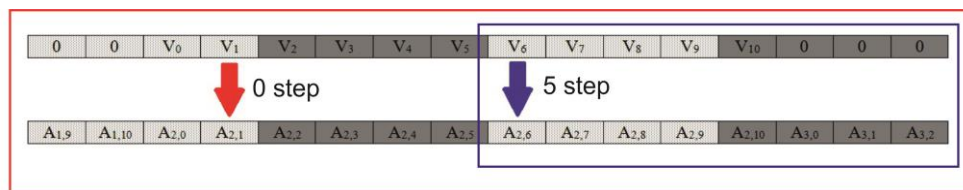


Рисунок 4.20 - Обработка второй строки матрицы в соответствии с (4.30).

Формально векторизацию цикла можно записать следующим образом (4.31).

Преобразование $V_1()$ определяет дополнение обрабатываемой строки слева и справа дополнительными элементами для устранения пролога и эпилога циклов. Аналогично преобразование $V_2()$ определяет дополнение строки матрицы W , а преобразование $V_3()$ – вектора отражения u_k . Для строки обрабатываемой матрицы данное преобразование $V_1()$ будет выглядеть следующим образом (4.32)

$$\begin{aligned}
\mathbf{v} &= \varphi_k \mathbf{u}_k^T \mathbf{A} \\
\mathbf{D} &= \text{diag}(v_0, v_1, \dots, v_n) \\
\mathbf{U}[:, j] &= [\mathbf{u}_k] \\
\mathbf{W} &= \mathbf{U} \mathbf{D} \\
\left. \begin{aligned}
\mathbf{b}_j^m &= V_1(\mathbf{b}_j) \\
\mathbf{w}_j^m &= V_2(\mathbf{w}_j) \\
\mathbf{m}_k &= V_3(\mathbf{u}_k) \\
\mathbf{b}_j^m &= \mathbf{b}_j^m - \mathbf{w}_j^m \\
\lambda'_j &= \varphi_k \mathbf{b}_j^m \mathbf{m}_k \\
\mathbf{b}_j^m &= \mathbf{b}_j^m - \lambda'_j \mathbf{m}_k^T \\
\mathbf{b}_j &= V_1^{-1}(\mathbf{b}_j^m)
\end{aligned} \right| j = \overline{k, n}
\end{aligned} \tag{4.31}$$

$$\begin{aligned}
\mathbf{b}_j^m[0: (n - 1 + \text{offset} - (k - \text{align}))] &= \\
&= \mathbf{b}_j[(k - \text{align}): (n - 1 + \text{offset})]
\end{aligned} \tag{4.32}$$

Преобразование $V^{-1}()$ определяет обратное преобразование, предназначенное для помещения результата вычислений в строку матрицы.

Преобразование $V_3()$ определяется выражением (4.33).

$$\begin{aligned}
\mathbf{Z}^T &= (0, 0, \dots, 0) \\
\mathbf{U}_{\text{row}}[i][0: (i - 1)] &= \mathbf{Z}^T[i - 1] \\
\mathbf{U}_{\text{row}}[i][i: (i - 1 + n)] &= \mathbf{u}_k^T \\
\mathbf{U}_{\text{row}}[i][(i + n): (n + L - 1)] &= \mathbf{Z}^T[L - i - 1] \\
N_{\text{offset}} &= \text{func}(j, n, n\text{Row}) \\
\mathbf{m}_k &= \mathbf{U}_{\text{row}}[\text{offset}]
\end{aligned} \tag{4.33}$$

Здесь $\mathbf{Z}^T$ – вектор – строка, содержащая нули, $\mathbf{U}_{\text{row}}$ – матрица размерностью $L*(n+L-1)$, содержащая вектора отражения с различными значениями смещений, L – размерность SIMD-регистров в элементах базового типа данных (i – значение смещения), N_{offset} – номер используемого вектора отражения, определяется по номеру строки, j – номер шага, n размерность матрицы, $n\text{Row}$ – номер строки.

Вычислительные эксперименты проводились в параллельной ВС с общей памятью, представляющей собой двухпроцессорный сервер на базе многоядерных МП Xeon E5-2650v3(x2) 2.3 GHz.

На рисунке 4.21 приведено ускорение для программ, реализующих однопроходную вычислительную схему, полученных с помощью автоматической векторизации и векторизованных с использованием предложенных алгоритмов.

На рисунке 4.21 используются следующие обозначения:

- Auto – значение ускорения для программы, реализующей однопроходную вычислительную схему, полученной с помощью автоматической векторизации;
- Vector – значение ускорения для программы, реализующей однопроходную вычислительную схему, векторизованной с помощью MBV.

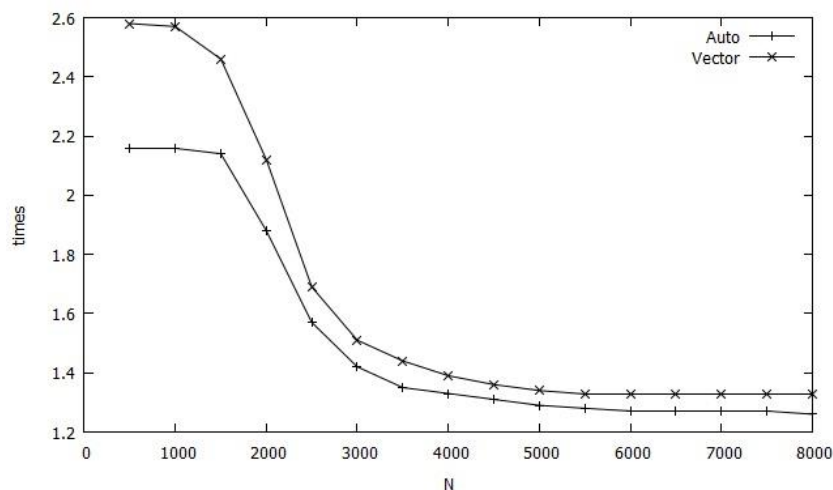


Рисунок 4.21 - Ускорение для программ, реализующих однопроходную вычислительную схему, полученных с помощью автоматической векторизации и векторизованных с использованием предложенных алгоритмов

Использовалась технология векторизации AVX, вещественные числа с двойной точностью (double). Ускорение определялось следующим образом:

- замерялось время выполнения программ без векторизации (компиляция с ключом `-no-vec`) для различных размерностей;

- компилировались программы с использованием автоматической векторизации и «ручной векторизации» в соответствии с предложенным алгоритмом;

- замерялось время выполнения полученных программ для различных размерностей;

- определялось значение ускорения.

Анализируя результаты, представленные на рисунке 4.21, можно сделать следующие выводы:

- автоматическая векторизация дает неплохие результаты для исследуемого алгоритма (ускорение до 2,15 на небольших значениях размерности);

- «ручная векторизация» в соответствии с MBV дает значительно лучший результат (ускорение до 2,65);

- с увеличением размерности ускорение уменьшается из-за увеличивающегося числа кеш–промахов и стабилизируется на значении 1,25 для автоматической векторизации и 1,33 для предложенного алгоритма.

На рисунке 4.22 приведены значения ускорения для программ, реализующих различные вычислительные схемы и векторизованные с использованием MBV по отношению к программной реализации скалярного классического преобразования Хаусхолдера.

Ускорение в данном случае определялось по отношению к одному и тому же алгоритму – классическому преобразованию Хаусхолдера без векторизации.

Анализируя результаты, представленные на рисунке 4.22, можно сделать следующие выводы:

- наименее эффективной оказалась реализация классического преобразования Хаусхолдера; из-за большого числа кеш–промахов с ростом размерности значение ускорения стремится к значению 1,03;

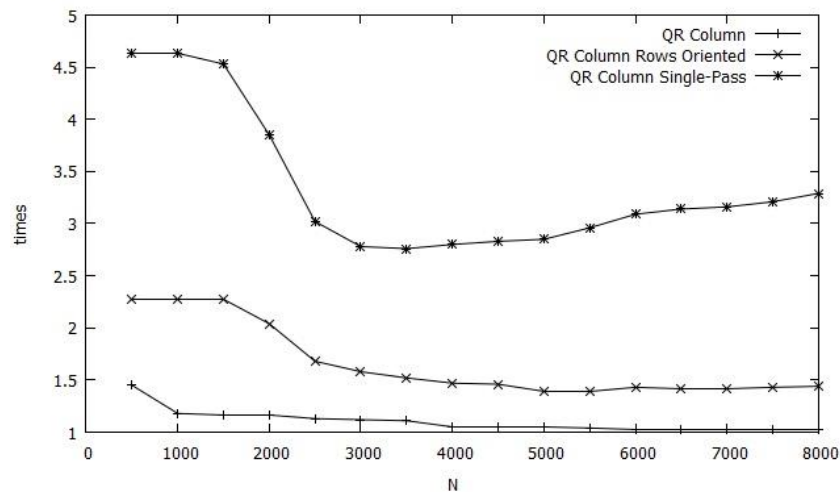


Рисунок 4.22 - Ускорение для программ, реализующих различные вычислительные схемы и векторизованные с использованием предложенных алгоритмов по отношению к программной реализации скалярного классического преобразования Хаусхолдера

- строчно-ориентированные схемы оказываются гораздо более эффективными, предложенная однопроходная схема – наиболее эффективной;

- видно, что значение ускорения для однопроходной схемы растет с ростом размерности; это означает, что скорость уменьшения эффективности данной вычислительной схемы меньше скорости уменьшения эффективности классического преобразования.

Также был проведен вычислительный эксперимент, целью которого было сравнение полученных решений с существующими аналогичными решениями [164]. В качестве объекта сравнения было решено использовать библиотеку Intel MKL.

Было осуществлено сравнение времени выполнения программ, осуществляющих приведение матриц общего вида к ХФ, написанных с использованием предложенных алгоритмов и с использованием функции LAPACKE_dgehrd() библиотеки Intel MKL. Результаты сравнения приведены на рисунке 4.23.

Видно, что на размерностях до 200 программная реализация приведения матрицы к ХФ, основанная на предложенных методах

превосходит реализацию, основанную на использовании функции LAPACKE_dgehrd() библиотеки Intel MKL, причем на небольших размерностях (до 120) превосходит значительно (от 2 до 4.5 раз). Однако в дальнейшем начинает проигрывать, и на больших размерностях проигрывать значительно. На размерностях до 60 использование МУСК не дает результата, т.к. матрица полностью помещается в кеш-память L1 (объем 32Кб, используемый тип данных – double с протяженностью 8 байт).

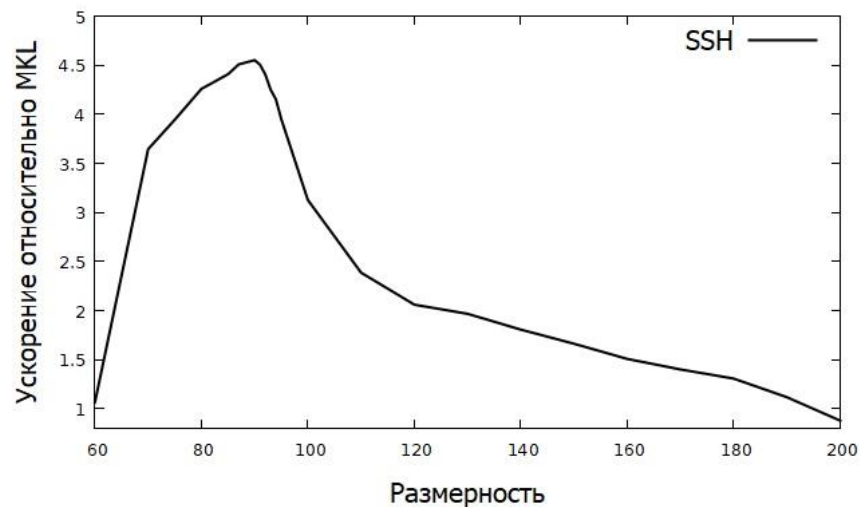


Рисунок 4.23 - Ускорение программы, реализованной с использованием предложенной однопроходной схемы по отношению к реализации с использованием LAPACKE_dgehrd() Intel MKL

Алгоритм, использованный в LAPACKE_dgehrd(), основан на блочном преобразовании Хаусхолдера, основанном на WY – представлении, которое также называют блочным преобразованием Хаусхолдера [152].

Пусть Q (4.34) – произведение матриц Хаусхолдера размерностью $n \times n$.

$$Q = Q_1 \dots Q_r \quad (4.34)$$

Матрица Q может быть представлена в виде (4.35), где W и Y – матрицы размерностью $n \times r$.

$$Q = I + WY^T \quad (4.35)$$

Алгоритм блочного представления матрицы Q может быть представлен следующим образом (4.36).

$$\begin{aligned}
Y &= v^{(1)} \\
W &= -2 v^{(1)} / [v^{(1)}]^T v^{(1)} \\
\text{for } j &= 2 \text{ to } r \\
z &= -2(I + WY^T) v^{(j)} / [v^{(j)}]^T v^{(j)} \\
W &= [W \ z] \\
Y &= [Y \ v^{(j)}] \\
\text{end}
\end{aligned} \tag{4.36}$$

Здесь $v^{(j)}$ – j -ый вектор отражения, Y – матрица векторов отражения.

Блочное представление преобразования Хаусхолдера основано на накоплении векторов отражения и в основном использует операции BLAS уровня 3 (операциях матрица – матрица, в т.ч. матричного умножения `gemm`). В то же самое время традиционное преобразование Хаусхолдера основано на операциях уровня BLAS 2 (операции матрица - вектор). При этом операции BLAS 2 реализуются на современных параллельных вычислительных архитектурах менее эффективно, чем операции BLAS 3. Это связано с тем, что операции BLAS 3 в т.ч. потенциально лучше оптимизируются с точки зрения использования кеш-памяти МП. Использование накопления векторов отражения и эффективная реализация матричного умножения вместо реализации операций BLAS 2 в традиционной реализации двустороннего преобразования Хаусхолдера дают значительный результат на больших размерностях.

Данное блочное представление преобразования Хаусхолдера является основой большого числа эффективных алгоритмов преобразования матриц, в т.ч. приведения матриц общего вида к ХФ.

В частности, в работах [70] рассматриваются вопросы эффективной реализации преобразования Хаусхолдера с использованием WY-представления, в т.ч. числе в применении к решению задач поиска собственных чисел и системы собственных векторов. Анализируются вопросы эффективного использования памяти при выполнении данных преобразований [112].

4.1.4 Результат реализации методологического подхода для задачи приведения матрицы общего вида к ХФ методом Хаусхолдера

Процесс проектирования содержал три итерации проектных процедур, применены МУСК и MBV (таблица 4.3).

Таблица 4.3 – Реализация методологического подхода для задачи приведения матрицы общего вида к ХФ методом Хаусхолдера

№ итерации	Реализуемый метод	Улучшаемые метрики	Полученная вычислительная схема
1	МУСК	R_p, S_p	Строчно-ориентированная
2	МУСК	R_p, S_p	Однопроходная
3	MBV	S_p, R_U	Однопроходная векторизованная

На первых двух итерациях осуществлялось разбиение множества операций доступа к памяти на непересекающиеся подмножества, оценивались количество элементов и вероятности кеш-промахов в каждом подмножестве, выявлялись подмножества, оказывающие существенное негативное влияние на эффективность реализации исследуемых схем преобразования. Выполнялись перестановочные оптимизации, в результате чего были получены две новых схемы выполнения преобразования, которые позволили значительно улучшить значения метрик ПС. Эксперименты показали, что МУСК позволяет до 4 раз ускорить вычисления, до 3-4 раз снизить энергопотребление ПС. На третьей итерации был реализован MBV, что позволило избежать потерь производительности при доступе к данным. На ряде размерностей было получено значительное ускорение по отношению к программной реализации, полученной с использованием Intel MKL.

4.2 Реализация методологического подхода для задачи матрично-векторного умножения

В BLAS выделяются три уровня: первый уровень содержит векторные операции, второй уровень – операции типа матрица – вектор и третий

уровень операции типа матрица-матрица. Рассмотрим реализацию MBV для операции BLAS уровня 2, а именно для операции умножения вектора на матрицу (в нотации функции BLAS `?gemv()`) (рисунок 4.24).

На рисунке 4.24 проиллюстрировано используемое преобразование, при этом используются следующие обозначения:

X - исходный вектор;

Y - результирующий вектор;

$M*N$ - размер подматрицы;

lda - размер исходной матрицы в направлении обработки.

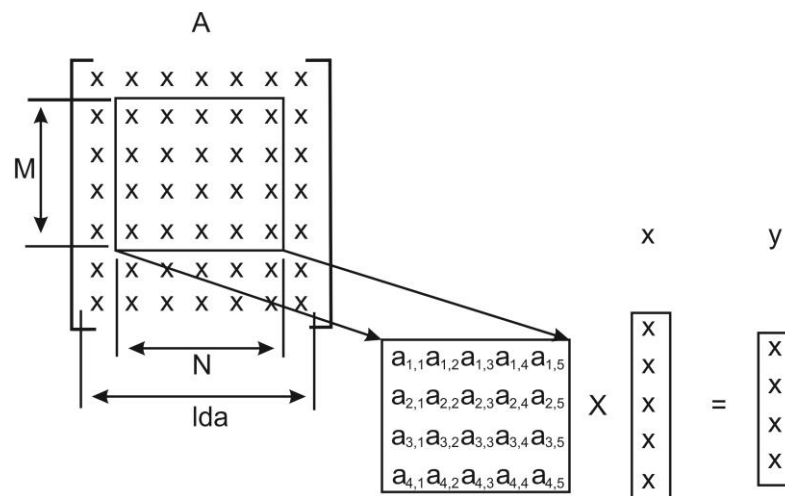


Рисунок 4.24 - Операция умножения вектора на подматрицу $y = A'x$

Оно заключается в умножении подматрицы, привязанной к некоторому начальному элементу (A_{11} на рисунке 4.24) на вектор x , при этом lda зависит от способа хранения матрицы. По-прежнему матрица хранится по строкам, поэтому ведущая размерность – длина строки.

Указанное преобразование может быть реализовано следующим образом.

```

1: for (int i = 0; i < M; i++)
2: {   Y[i] = 0;
3:     for (int j = 0; j < N; j++)
4:         Y[i] += ptr[j] * X[j];
5:     ptr = ptr + lda; }

```

(4.37)

Для компиляции программы использовался компилятор Intel C++ Compiler с использованием следующей командной строки (в процессе компиляции использовались OpenMP и AVX, был создан отчет компилятора).

```
icpc *.cpp -o vect -qopenmp -xAVX -qopt-report=5 (4.38)
```

Отчет содержит следующую информацию:

- созданы несколько версий кода: векторизованная и не векторизованная;
- создан пролог векторизованной версии (peeled loop); т.к. использовался тип данных float, максимальная длина пролога составляет 7 элементов, т.к. в регистр AVX помещается 8 элементов float;
- в основной части использован невыровненный доступ к элементам массива ptr;
- сформирован эпилог, в рамках которого используется невыровненный доступ к данным;
- эпилог не был векторизован, т.к. компилятор считал это неэффективным;
- приведена информация о не векторизованной версии цикла.

4.2.1 Реализация MBV для задачи матрично-векторного умножения

Применим MBV к данному преобразованию.

Первый этап.

1. Определение характеристик векторизуемого цикла

Ширина SIMD-регистров зависит от выбранной технологии векторизации и используемых типов данных. В рассматриваемом случае ширина равна 8 элементам, т.к. используется технология AVX (256 бит) и тип данных float (32 бита).

Т.к. матрица по строкам, ее размер в направлении обработки равен lda , размер обрабатываемого вектора в направлении обработки равен N .

Смещения первых элементов обрабатываемых строк в общем случае будут отличаться, но при этом будут формировать повторяющийся паттерн, число элементов которого зависит от lda и ширины SIMD-регистра. Смещение j -го элемента i -ой обрабатываемой строки можно вычислить с использованием следующего выражения.

$$S_{i,j} = ((i * lda + j) \% SIMD_{width} + rel_{0,0}) \% SIMD_{width} \quad (4.39)$$

Пусть $rel_{0,0} = 0$, $lda = 1000$. Тогда смещения вторых элементов строк ($j =$

1) будут равны:

- $i = 0, S = ((0 * 1000 + 1) \% 8 + 0) \% 8 = 1;$
- $i = 1, S = ((1 * 1000 + 1) \% 8 + 0) \% 8 = 1;$
- $i = 2, S = ((2 * 1000 + 1) \% 8 + 0) \% 8 = 1;$
- ...

Как видно, смещения элементов в разных строках совпадают.

Пусть $rel_{0,0} = 0$, $lda = 1002$. Тогда смещения вторых элементов строк ($j =$

1) будут равны:

- $i = 0, S = ((0 * 1002 + 1) \% 8 + 0) \% 8 = 1;$
- $i = 1, S = ((1 * 1002 + 1) \% 8 + 0) \% 8 = 3;$
- $i = 2, S = ((2 * 1002 + 1) \% 8 + 0) \% 8 = 5;$
- $i = 3, S = ((3 * 1002 + 1) \% 8 + 0) \% 8 = 7;$
- $i = 4, S = ((4 * 1002 + 1) \% 8 + 0) \% 8 = 1;$
- $i = 5, S = ((5 * 1002 + 1) \% 8 + 0) \% 8 = 3;$
- ...

Как видно, смещения могут принимать одно из четырех значений, порядок смещений повторяется. В общем случае количество и значения смещений зависит от значения (4.26).

Если lda кратна ширине SIMD-регистра (восьми в случае использования AVX и float), то смещения первых элементов всех строк

подматрицы будут одинаковыми и не обязательно равными нулю. В самом деле,

$$\begin{aligned}
 S_{i,j} &= ((i * lda + j) \% SIMD_{width} + rel_{0,0}) \% SIMD_{width} \\
 &= (((i * lda) \% SIMD_{width} + j \% SIMD_{width}) \% SIMD_{width} \\
 &\quad + rel_{0,0}) \% SIMD_{width} \\
 &= (j \% SIMD_{width}) \% SIMD_{width} + rel_{0,0} \\
 &= (j \% SIMD_{width} + rel_{0,0}) \% SIMD_{width} \\
 &= j \% SIMD_{width} + rel_{0,0}
 \end{aligned} \tag{4.40}$$

Из (4.40) видно, что смещение в данном случае зависит только от номера столбца, но не зависит от номера строки, т.е. одинаково для всех строк.

2. Формирование необходимого количества ведущих векторов.

Число ведущих векторов зависит от количества несовпадающих смещений начальных элементов строк подматрицы и определяется lda и используемой технологией векторизации в соответствии с (4.23).

Так, например, при использовании AVX и $lda=1004$, число ведущих векторов будет равно 2 (рисунок 4.25). В самом деле, в соответствии с (4.21) $a=4$, тогда из (4.22) следует, что $n=1$, тогда $nv=2$ (последнее условие в 4.23).

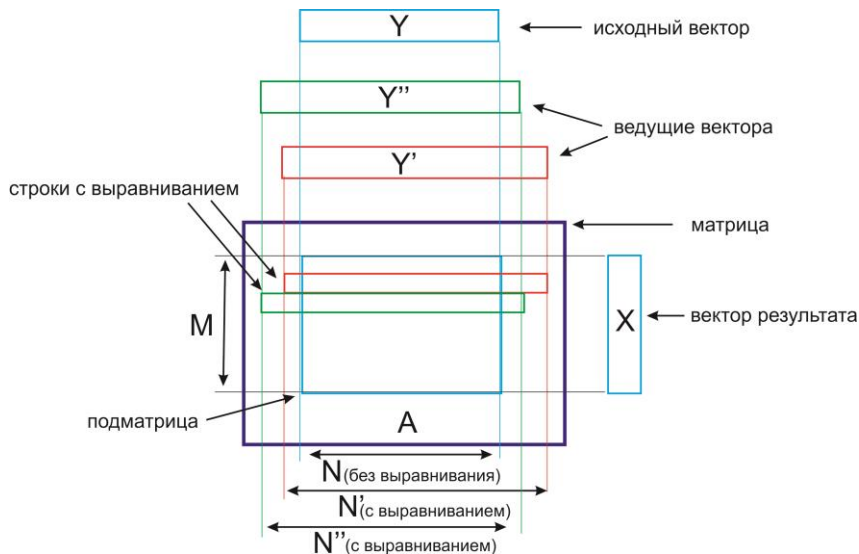


Рисунок 4.25 – Формирование двух ведущих векторов

3. Определение длин ведущих векторов и их смещений относительно «естественного положения».

Число нулей слева для k -го ведущего вектора равно смещению первого обрабатываемого элемента в k -й строке

$$Z_k^L = S_{k,j} \quad (4.41)$$

Число нулей справа для k -го ведущего вектора определяется следующим образом.

$$Z_k^R = (SIMD_{width} - (Z_k^L + l) \% SIMD_{width}) \% SIMD_{width} \quad (4.42)$$

Здесь l – длина обрабатываемой подстроки. Общая длина k - вектора равна

$$L_k = Z_k^L + l + Z_k^R \quad (4.43)$$

и кратна ширине SIMD-регистра.

Второй этап в основном связан с написанием программного кода, программную реализацию исследуемой функции с применением MBV можно представить следующим образом.

```

1: getVectors(&nv, m, b, A, stride, Y);
2: for (int i = 0; i < M; i += nv)
3:   for (int j = 0; j < nv; j++)
4:     for (int k = 0; k < N; k += L * SIMD_WIDTHH)
5:       Y[i + j] = _kernel(b[i + j], m[j]);

```

(4.44)

В (4.44) используются те же обозначения, что и на рисунке 3.41. В первой строке осуществляется получение всех необходимых параметров и системы ведущих векторов. Функция `_kernel()` реализует умножение очередной строки подматрицы на вектор-столбец. В процессе разработки программы на начальном этапе была предпринята попытка использования только автоматической векторизации полученного кода с использованием соответствующих директив компилятора Intel C++ Compiler. Однако такой подход не дал ожидаемого результата, в связи с чем были использованы интринсики для векторизации вычислений, т.е. более низкоуровневый способ. Компилятор самостоятельно не справился с задачей, т.к. в процессе

формирования кода он не делает предположений о длине используемых векторов. В связи с этим он вставлял эпилоги циклов, что привело к тому, что положительный эффект от применения MBV совместно с автоматической векторизацией практически полностью нивелировался необходимостью обработки эпилогов циклов.

Эксперименты проводились на вычислительной системе, построенной на базе процессоров Xeon E5-2650v3(x2). Были исследованы реализации, полученные с использованием автоматической векторизации и реализации, полученные с использованием MBV. Использовался язык программирования C++, компилятор Intel C++ Compiler, технология векторизации AVX, тип данных float. В процессе моделирования были получены следующие результаты:

- для обеих реализаций время выполнения для равных значений N (длины вектора X) зависит от размера матрицы в направлении обработки (lda);
- для реализации, полученной в результате автовекторизации, время выполнения существенным образом зависит от N : минимально для значений, кратных ширине SIMD-регистра (8, 16, ...), максимально для размерностей с максимальным «скалярным остатком» (15, 23, ...);
- для реализации, полученной с использованием MBV, время выполнения нарастает монотонно с ростом N .

Таким образом, для фиксированного значения lda ускорение, полученное с учетом MBV, не возрастает монотонно с ростом N , минимально для значений $(k * 8)$, максимально для значений $(k * 8 + 7)$. Стоит отметить, что во всех опытах начальный элемент матрицы располагался по выровненному адресу. Другие варианты не рассматривались, так как в противном случае будут получены результаты, которые будут дублировать друг друга для различных сочетаний lda и начального выравнивания. Варианты различного выравнивания умножаемого вектора также не исследовались, поскольку, если умножаемый вектор не будет выровнен,

данный аспект будет несложно учесть на этапе формирования ведущих векторов.

На рисунке 4.26 представлена зависимость получаемого ускорения для $lda=1000$ и $lda=1001$.

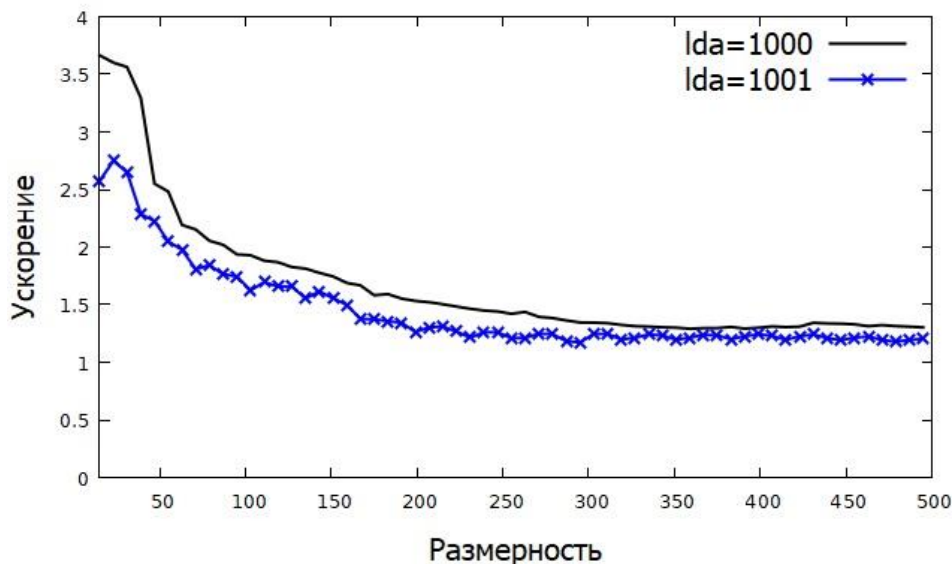


Рисунок 4.26 - Зависимость ускорения выполнения программы относительно автоматической векторизации

На графике представлены две зависимости, причем ускорение для $lda=1000$ выше для всех значений размерности. Это происходит из-за того, что для $lda=1000$ необходимо сформировать один ведущий вектор, для $lda=1001$ – восемь векторов. Замеры времени осуществлялись для умножения ленточной подматрицы $1000 * N$ на вектор длины N , где N отложено вдоль оси абсцисс.

Также необходимо отметить, что в данном эксперименте анализировались только проблемы векторизации, поэтому размер исходного вектор-столбца выбирался таким образом, чтобы целиком помещаться в кеш L1 микропроцессора. При больших значениях N , когда вектор прекратит помещаться в L1 или в кеш-память верхних уровней, будут генерироваться кеш-промахи, поэтому необходимо будет в рамках методологического подхода реализовывать МУСК для исследуемого преобразования [165, 166, 169].

Эксперименты показали, что применение MBV значительно ускоряет работу ПС. На небольших размерностях выигрыш составляет до 3,7 раз, с ростом размерности значение ускорения уменьшается и стабилизируется в районе 1,2. Такое поведение объясняется тем, что кроме эффекта прологов и эпилогов, который сказывается в основном на небольших размерностях, MBV позволяет избежать невыровненного доступа к данным, что оказывает положительный эффект на обработку массивов любых размеров, формируя асимптотическое ускорение.

Были также проведены исследования векторизации с технологией SSE, которые показали ощутимый прирост производительности. В целом необходимо отметить, что данный подход может быть использован для любой ширины SIMD-регистров (например, AVX-512), причем ожидается, что эффект будет усиливаться с ростом ширины регистров.

MBV также может применяться для обработки массивов, в которых все строки матрицы и векторы сразу располагаются по выровненным адресам, при этом отсутствуют заключительные элементы, формирующие эпилоги циклов. Исходя из отчета компилятора, в процессе автовекторизации в данном случае все равно формируются прологи и эпилоги, но для таких «идеально выровненных» данных они будут нулевой длины. Но процессорное время на их обработку тратится. Кроме того, при доступе к элементам одного из массивов организуется невыровненный доступ, что приводит к временным «штрафам». Соответственно, асимптотическое ускорение (в наших опытах 1,2) будет наблюдаться и в этом случае.

При этом ситуация с «идеальным выравниванием» возможна далеко не во всех случаях. Во многих алгоритмах используется умножение на подматрицу, причем размерность подматрицы меняется. В качестве примера можно привести алгоритм приведения матрицы общего вида к ХФ форме с использованием блочного преобразования Хаусхолдера [152]. Там активно используется перемножение подматриц, размерность которых меняется от шага к шагу. Выравнивание в этом случае также будет меняться.

Полученное ускорение было получено благодаря значительному снижению количества невыровненных обращений к памяти. На рисунке 4.27 приведена зависимость снижения доли невыровненных обращений (метрика, рассчитываемая в соответствии с 3.35).

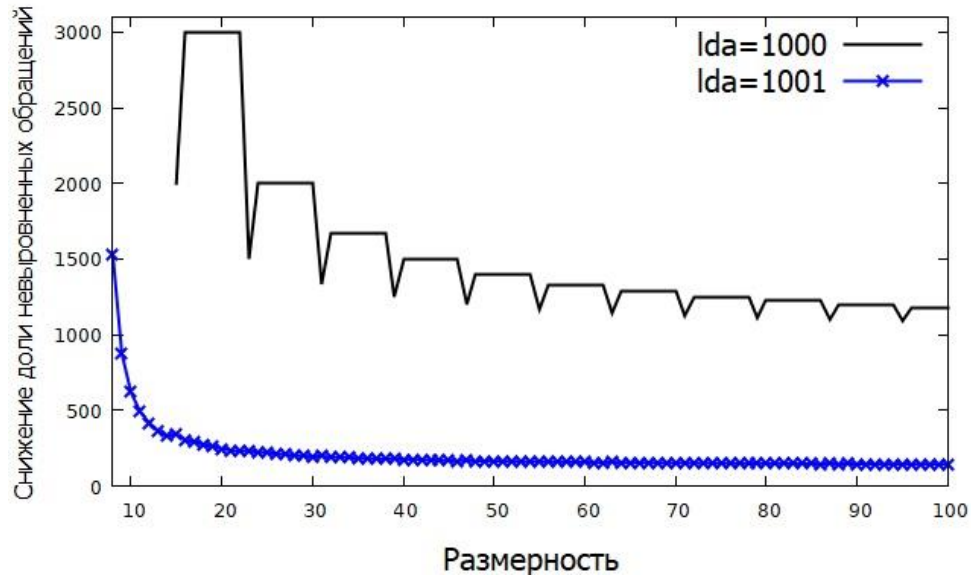


Рисунок 4.27 - Снижение доли невыровненных обращений

По тем же причинам, что и в случае полученного ускорения (рисунок 4.26), значение данной метрики ПС для $lda=1000$ оказывается значительно лучшим, чем для $lda=1001$.

4.2.2 Результат реализации методологического подхода для задачи матрично-векторного умножения

Процесс проектирования содержал одну итерации проектных процедур, применен MBV (таблица 4.4).

На первом этапе MBV были определены характеристики векторизуемого цикла и формирование необходимого числа ведущих векторов, определены их длины и смещения. Второй этап заключался в написании программного кода: были изменены циклы доступа к данным в соответствии с полученными параметрами.

Таблица 4.4 – Реализация методологического подхода к задачам матрично-векторного умножения и обработки числовых последовательностей

№ итерации	Реализуемый метод	Улучшаемые метрики	Полученное ускорение относительно результатов, полученных с помощью автоматической векторизации
1	MBV	S_p, R_U	до 3,7 раз

4.3 Реализация методологического подхода для задачи обработки числовых последовательностей

Под числовыми последовательностями понимается последовательности чисел одного типа, числовые последовательности являются одним из основных объектов рассмотрения в математическом анализе. Элементы последовательности пронумерованы, нумерация осуществляется целыми, иногда натуральными числами. Последовательности определяются на некотором множестве, например, на множестве вещественных или целых чисел. На множестве всех последовательностей элементов базового множества определяются операции, простейшими из которых являются арифметические операции, которые выполняются поэлементно. Очевидно, что обработка последовательностей имеет высокий потенциал к повышению эффективности их обработки за счет векторизации.

Помимо арифметических операций, для последовательностей вводят и другие, более сложные операции, одной из которых является свертка. Свертка в общем случае представляет собой математическую операцию над двумя функциями, в результате получается третья функция. Она вычисляется как интеграл над произведением исходных функций, одна из которых отражена относительно оси ординат и сдвинута. Данная операция широко используется в цифровой обработке сигналов, т.к. описывает взаимодействие сигналов, описываемых исходными функциями, между собой. Так, например,

если один из сигналов представляет собой импульсную характеристику фильтра, то свертка входного сигнала и импульсной характеристики дает реакцию цепи на входное воздействие [184].

Необходимо отметить, что на практике в численных расчетах используется дискретная свертка, которая представляет собой свертку двух последовательностей, получающихся при переходе от непрерывных сигналов к дискретным (4.45)

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) = \sum_{k=-\infty}^{\infty} h(k)x(n-k), \quad (4.45)$$

где $x(k)$ – входное воздействие;

$y(k)$ – выходной сигнал;

$h(k)$ – импульсная характеристика.

На практике используются дискретные последовательности конечной длины. При этом из (4.45) легко получить, что если одна последовательность имеет N ненулевых элементов, вторая – M , то выходная последовательность имеет $N + M - 1$ ненулевых элементов.

Часто выполнение операции свертки трактуется как свертка исходной последовательности и ядра свертки, которое представляет собой последовательность меньшей длины. Выполнение свертки осуществляется путем вычисления скалярных произведений ядра свертки с соответствующими подпоследовательностями исходной последовательности. Пусть x – исходная последовательность, дополненная нулями, h – ядро свертки, тогда получение i -го элемента результирующей последовательности описывается скалярным произведением векторов в соответствии с (4.46)

$$y[i] = x_i h, \quad (4.46)$$

где x – исходная последовательность длиной N ;

h – ядро свертки длиной M ;

x_i – подпоследовательность, полученная из исходной последовательности длиной M , начиная с i -го элемента.

Общая длина результирующей последовательности y равна $N - M + 1$.

4.3.1 Реализация MBV для задачи одномерной свертки

Применим MBV к данному преобразованию.

Первый этап.

1. Определение характеристик векторизуемого цикла

Ширина SIMD-регистров зависит от выбранной технологии векторизации и используемых типов данных. Рассматривается вариант использования технологии AVX и типа данных float, поэтому ширина SIMD-регистров равна 8.

Размер обрабатываемого вектора в направлении обработки равен M , т.е. длине ядра свертки.

Т.к. осуществляется последовательное перемещение окна свертки вдоль исходной последовательности, смещения первых элементов обрабатываемых векторов будут последовательно изменяться на 1.

2. Формирование необходимого количества ведущих векторов.

Т.к. считаем размер исходной последовательности достаточно большим, понадобится 8 ведущих векторов.

3. Определение длин ведущих векторов и их смещений относительно «естественного положения».

Число нулей слева для k -го ведущего вектора будет равно его номеру (от 0 до 7.).

Число нулей справа для k -го ведущего вектора определяется в соответствии с (4.42).

Были исследованы реализации, полученные с использованием автоматической векторизации и реализации, полученные с использованием MBV. Использовался язык программирования C++, компилятор Intel C++ Compiler, технология векторизации AVX, тип данных float.

Были получены две реализации преобразования одномерной свертки:

- в соответствии с классическим алгоритмом (4.45), (4.46);
- реализация одномерной свертки с использованием MBV.

Для классической реализации компиляция осуществлялась двумя способами:

- с ключом `-xAVX`, указывающим на необходимость использования технологии AVX для векторизации вычислений;
- без дополнительных ключей, указывающих способ оптимизации кода.

Были получены версии программы, которые отличаются друг от друга временем выполнения. Результат представлен на рисунке 4.28.

Свертка выполнялась над последовательностью длиной 32000 элементов. Видно, что на маленьких длинах ядра свертки более предпочтительной является вариант «без ключей оптимизации». С ростом размера ядра свертки все более предпочтительной становится вариант с ключом компилятора «`-xAVX`», представляющий собой нелинейную зависимость, где максимумы и минимумы чередуются с интервалом 16.

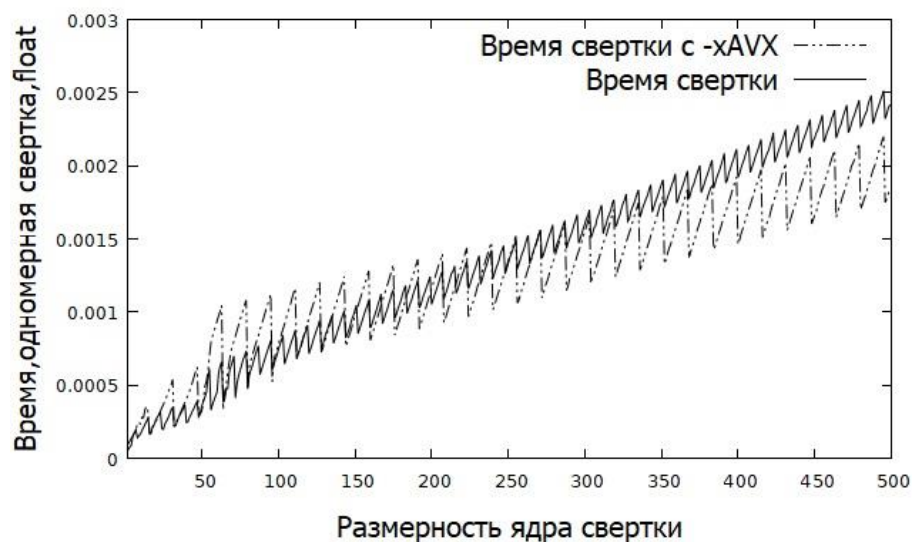


Рисунок 4.28 – Зависимость времени выполнения одномерной свертки от размера ядра свертки с автоматической векторизацией

Это связано с тем, что согласно отчету компилятора цикл разворачивается на 2 итерации. Каждая итерация связана с обработкой 8 элементов (тип данных `float`, технология векторизации AVX), две итерации представляют собой 16 элементов. В процессе автоматической векторизации

компилятор формирует прологи и эпилоги циклов, при этом время вычислений зависит от длины прологов и эпилогов циклов, которые меняются при скольжении окна свертки вдоль исходной последовательности, при росте этих значений время увеличивается, при уменьшении снижается. Для варианта «по умолчанию» максимумы и минимумы чередуются с другим шагом, т.к. компилятор выбрал другой вариант векторизации.

Для реализации с использованием MBV компиляция осуществлялась тремя способами:

- без дополнительных директив компилятора;
- с директивой `#pragma vector aligned`;
- с директивой `#pragma omp simd`.

При этом также возможно использование различных ключей компилятора. Результат представлен на рисунке 4.29.

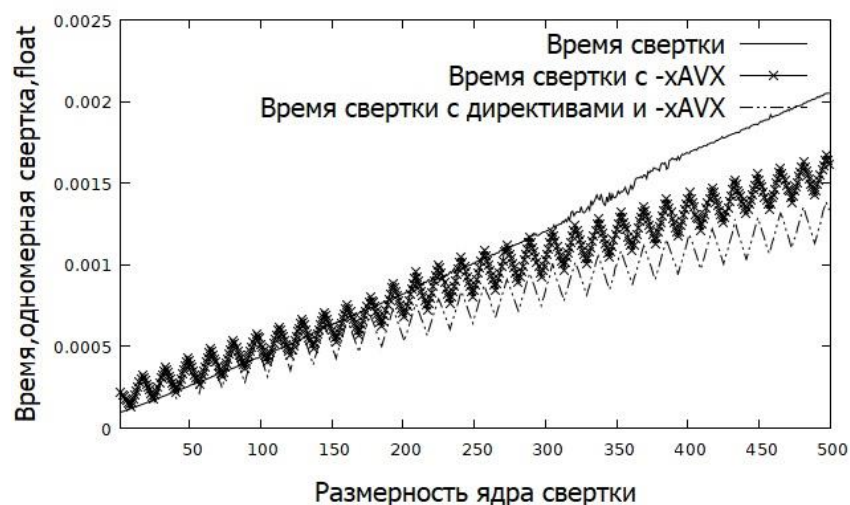


Рисунок 4.29 – Зависимость времени выполнения одномерной свертки от размера ядра свертки с использованием MBV

Видно, что на малых размерностях более предпочтительной является версия вариант «без ключей оптимизации». С ростом размера ядра свертки все более предпочтительной становится вариант с ключом компилятора «-xAVX» и директивами компилятора (оба варианта директив компилятора дают похожий результат), Варианты с ключом «-xAVX» также дают нелинейную зависимость по тем же причинам, чередование максимумов и минимумов также происходит с интервалом 16.

Данный вид зависимости объясняется нелинейным ростом времени выполнения для обеих реализаций:

- для автоматической векторизации;
- для MBV время вычислений зависит от текущей длины обрабатываемой подпоследовательности, которая в свою очередь зависит от текущего смещения окна свертки в исходной последовательности (4.43).

Были определены ускорения, получаемые при использовании различных вариантов свертки, результат представлен на рисунках 4.30, 4.31.

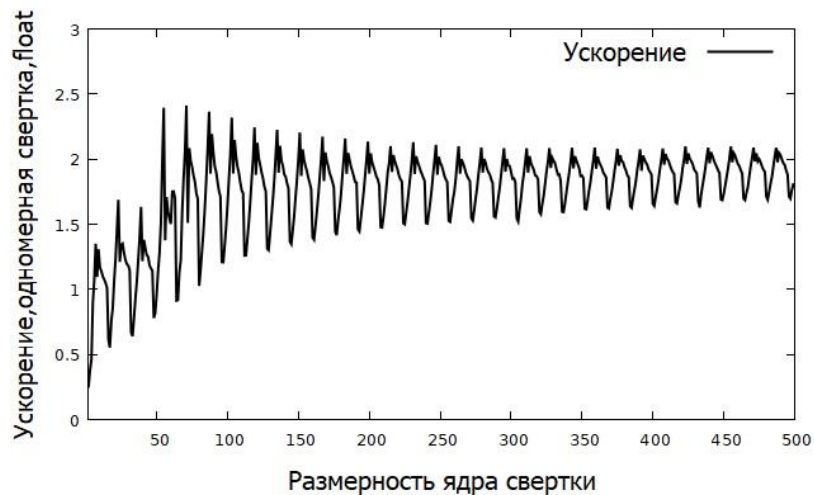


Рисунок 4.30 — Ускорение лучшей программной реализации одномерной свертки с MBV относительно автоматической векторизации

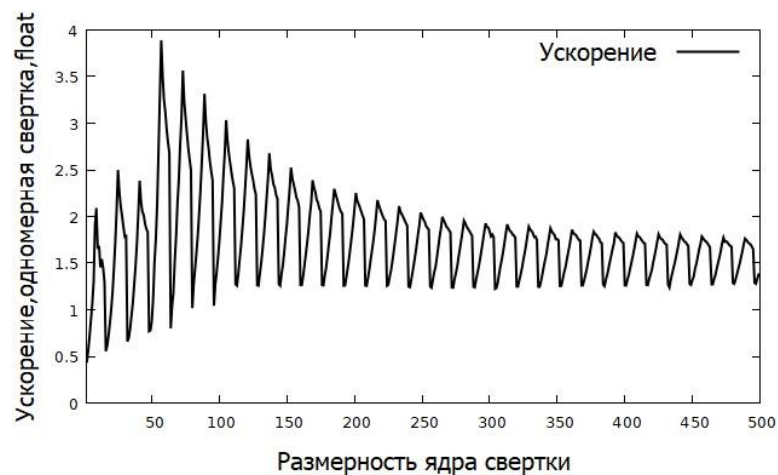


Рисунок 4.31 — Ускорение лучшей программной реализации одномерной свертки с MBV относительно автоматической векторизации с ключом «-xAVX»

Таким образом, ускорение периодически изменяется, в каждом периоде достигая максимального и минимального значения. Зависимости максимальных и минимальных значений ускорения выполнения программы приведены рисунках 4.32 и 4.33 (в качестве базовой использовалась зависимость на рисунке 4.30).

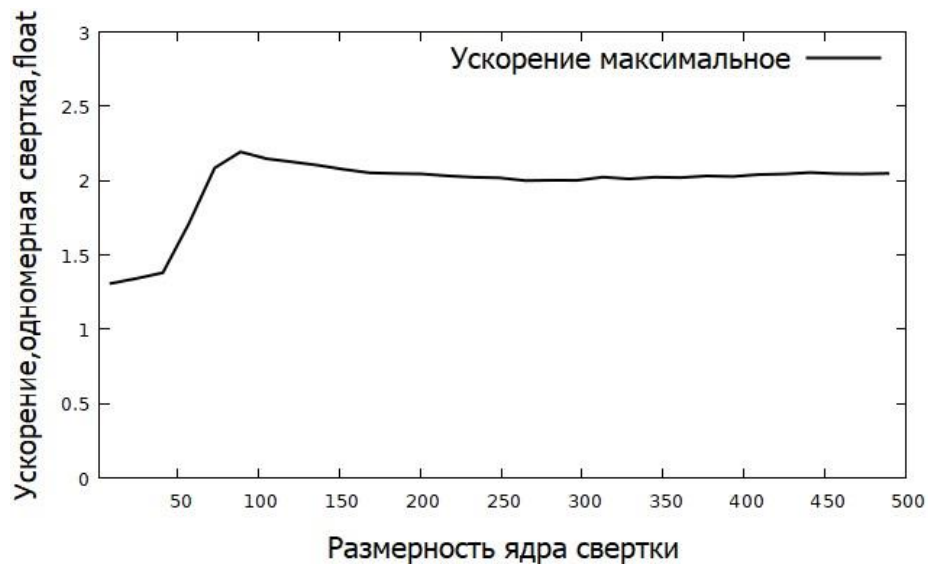


Рисунок 4.32 - Зависимость максимальных значений ускорения выполнения программы относительно автоматической векторизации

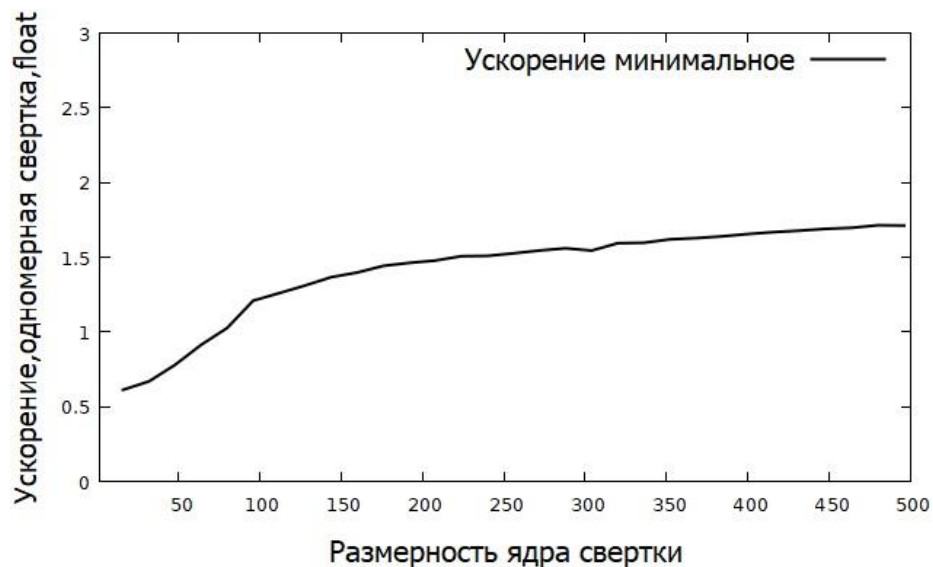


Рисунок 4.33 - Зависимость минимальных значений ускорения выполнения программы относительно автоматической векторизации

Анализируя зависимости, приведенные на рисунках 4.32 и 4.33, можно сделать следующие выводы:

- максимальное значение ускорения, достигнув максимума в 2,25 раза, снижается с ростом размерности и стабилизируется на значении 2, возрастая с ростом размерности ядра свертки;
- минимальное значение ускорения увеличивается с ростом размерности ядра свертки, уменьшая разницу между максимальным и минимальным достигаемым ускорением.

Были также проведены вычислительные эксперименты по выполнению одномерной свертки с использованием технологии AVX и типа данных `__int16`, результаты приведены на рисунке 4.34.

Видно, что зависимость ускорения также носит нелинейный характер, при этом с ростом размерности среднее значение ускорения стабилизируется в районе 3. Т.е. увеличение ширины SIMD-регистра в 2 раза (8 для float и 16 для `__int16`) привело к увеличению среднего значения ускорения в 1,5 раза.

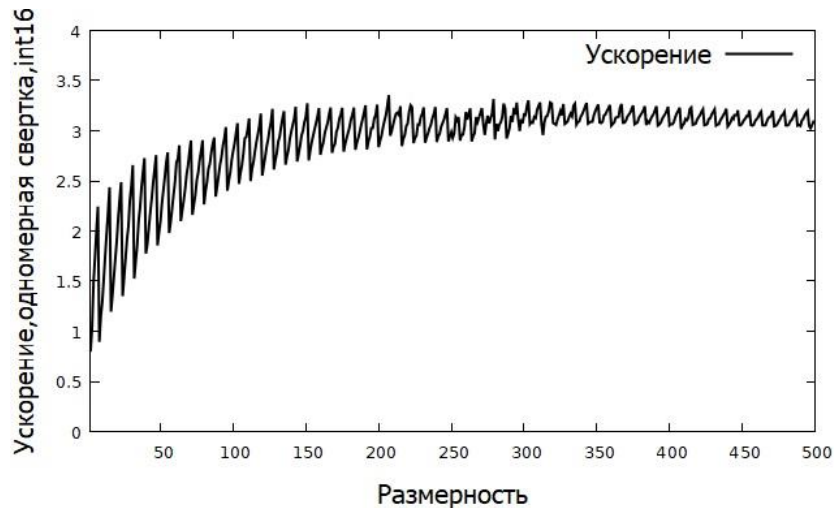


Рисунок 4.34 – Ускорение лучшей программной реализации одномерной свертки с MBV относительно автоматической векторизации для типа данных `__int16`.

Проведенный анализ представляет собой настройку параметризованного алгоритма, т.к. из анализа зависимостей, приведенных на рисунках 4.28 – 4.31 видно, что на различных диапазонах размерностей более предпочтительными являются различные способы применения

оптимизирующего компилятора (использование различных директив, ключей компиляции). ПС, использованные для анализа разрабатывались только с использованием высокоуровневого языка программирования C++. В процессе написания программ не использовались низкоуровневые средства, такие как интринсики или прямое использование инструкций МП (язык ассемблера), таким образом эффективность ПС можно дополнительно значительно повысить, при этом снизить зависимость от эвристики оптимизирующего компилятора.

4.3.2 Реализация MBV для задачи двумерной свертки

Обобщением операции дискретной свертки является операция многомерной дискретной свертки, где исходные последовательности определены на n -решетке, создаваемая последовательность также имеет размерность n .

$$\begin{aligned}
 y(n_1, n_2, \dots, n_M) & \quad (4.47) \\
 &= \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} \dots \sum_{k_M=-\infty}^{\infty} x(k_1, k_2, \dots, k_M) h(n_1 - k_1, n_2 - k_2, \dots, n_M - k_M)
 \end{aligned}$$

В случае одномерной свертки одномерное окно свертки перемещается вдоль одномерной исходной последовательности, в случае двумерной свертки двумерное окно, прямоугольник (часто квадрат), перемещается вдоль двух координат исходной последовательности, определенной на двумерной решетке, в трехмерном случае окно свертки представляет собой прямоугольный параллелепипед и т.д.

Операция двумерной свертки широко используется при построении сверточных сетей, предназначенных для классификации изображений. Слой свёртки, как можно догадаться по названию типа нейронной сети, является самым главным слоем сети. Его основное назначение – выделить признаки на

входном изображении и сформировать карту признаков. Карта признаков представляет собой тензор (массив матриц), в котором каждый канал отвечает за определенный выделенный признак. Свёрточные нейронные сети обрабатывают изображения, представленные в виде тензоров. Тензоры представляют собой многомерные массивы чисел. Изображения в компьютере представляются в виде пикселей, каждый пиксель представляет собой значение интенсивности соответствующего цветового канала, описывается целым числом, чаще всего от 0 до 255, хотя существуют и другие форматы файлов. Поэтому для описания изображения в сверточных сетях используются тензоры, при этом отдельные матрицы тензора отвечают за отдельные цветовые каналы.

Применим MBV к операции двумерной дискретной свертки. Будем считать, что окно свертки представляет собой квадрат и шаг свертки равен единице.

Формирование ведущих векторов в данном случае отличается от случая одномерной свертки тем, что для каждой строки ядра необходимо формировать требуемое число векторов, зависящее от технологии векторизации и используемого типа данных (в рассматриваемом случае для AVX типа float необходимо 8 векторов).

Однако такой подход не даст ожидаемого прироста производительности. Проблема связана с тем, что уменьшается эффективность использования подсистемы памяти. Сравним количество обращений к памяти при выполнении классического преобразования двумерной свертки и алгоритма, полученного в результате применения MBV.

В обоих случаях количество обращений к памяти при обработке исходной двумерной последовательности размера $M \times N$ с ядром свертки размера $U \times V$ составляет порядка $O(M \times N \times U \times V)$, однако, в случае использования MBV количество операций обращения к памяти превышает данное количество в случае реализации классической схемы преобразования. Это связано с тем, что размерность ведущих векторов может превышать

размерность обрабатываемых векторов. В случае двумерной свертки это приводит к значительному увеличению времени выполнения преобразования.

Изменим порядок выполнения операций следующим образом. Перед выполнением операции умножения ядра свертки на его проекцию в исходной последовательности:

- скопируем проекцию ядра свертки на исходную последовательность в одномерный вектор;
- скорректируем полученный вектор в соответствии с MBV таким образом, чтобы его длина была пропорциональна длине векторного регистра;
- ядро свертки, хранящееся в плоском массиве, который может быть интерпретирован в виде одномерного массива, скорректируем в соответствии с MBV таким образом, чтобы его длина была пропорциональна длине векторного регистра.

Далее для выполнения операции умножения ядра свертки на его проекцию в исходной последовательности выполним операции скалярного умножения полученных векторов, данную операцию повторим для всех проекций ядра свертки на исходную последовательность.

В соответствии с полученным алгоритмом двумерной свертки была разработана ПС на языке C++, в которой использовались только высокоуровневые средства оптимизации, включающие в себя директивы и ключи компилятора, были проведены вычислительные эксперименты. На рисунке 4.35 представлено полученное ускорение программной реализации двумерной свертки с MBV относительно автоматической векторизации для типа данных float.

Использовались ядра свертки с нечетной размерностью. Зависимость ускорения от размерности ядра свертки имеет нелинейный характер, при этом достигается двукратное ускорение относительно результата, полученного с помощью автоматической векторизации.

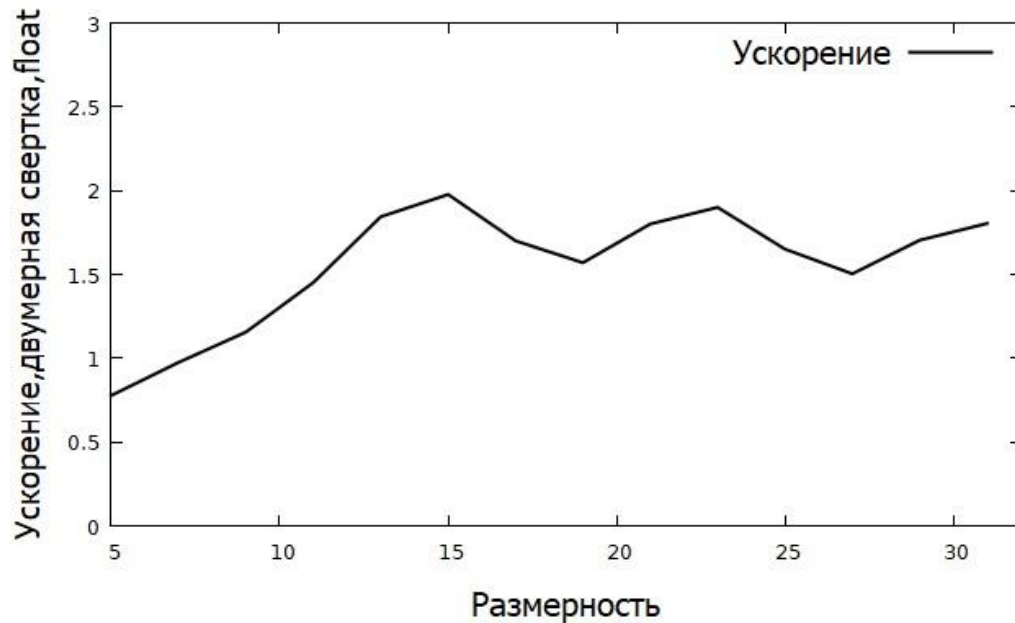


Рисунок 4.35 - Ускорение программной реализации двумерной свертки с MBV и МУСК относительно автоматической векторизации для типа данных float.

Необходимо отметить, что для выполнения многомерных, в т.ч. двумерных свертки, применяется ряд методов, которые не являются прямыми.

4.3.3 Результат реализации методологического подхода для задачи обработки числовых последовательностей

Процесс проектирования содержал две итерации проектных процедур, применены МУСК и MBV (таблица 4.5).

Таблица 4.5 – Реализация методологического подхода для задачи обработки числовых последовательностей

№ итерации	Реализуемый метод	Улучшаемые метрики	Полученное ускорение
1	МУСК	R_p, S_p	Одномерная свертка – до 2,5 - 3
3	MBV	S_p, R_U	Двумерная свертка – до 2

На первой итерации проектных процедур выполнена перестановочная оптимизация в рамках МУСК, снизившая число кеш-промахов. На второй

итерации реализован MBV, сформированы ведущие вектора, изменены циклы доступа к данным в соответствии с полученными параметрами.

4.4 Основные результаты и выводы по главе 4

Выполнена верификация предложенной методологии проектирования ЭПС на примере практически значимых вычислительно-сложных задач (операции BLAS 2 и 3 уровня, задачи обработки числовых последовательностей).

Методологический подход был реализован для задачи приведения матрицы общего вида к Хессенберговой форме и двухдиагональной формам методом Хаусхолдера. Реализация МУСК на первой и второй итерации проектных процедур позволила значительно улучшить значения ряда метрик ПС, в частности снизить усредненное значение кеш-промаха и время работы программы. В результате были получены две вычислительные схемы данного преобразования: строчно-ориентированная и однокроходная схемы выполнения двустороннего преобразования Хаусхолдера, показавшие значительное, до 4 раз, ускорение по сравнению с реализацией классического двустороннего преобразования. Отдельно была рассмотрена энергоэффективность полученных алгоритмов, для чего был использован программный интерфейс RAPL, разработанный компанией Intel. Вычислительные эксперименты показали снижение энергопотребления ВС при выполнении ПС, реализующих полученные алгоритмы, до 3-4 раз.

На третьей итерации проектных процедур для указанного преобразования был реализован MBV. Была исследована метрика полученного ускорения для ПС, реализующих однокроходную вычислительную схему, обладающую лучшими показателями эффективности, полученных с помощью автоматической векторизации и векторизованных с использованием предложенной методологии. Вычислительные эксперименты показали, что реализация MBV дает большее

ускорение. Было также исследовано ускорение полученных ПС относительно не векторизованных версий программной реализации классического преобразования Хаусхолдера, было выявлено значительное ускорение, которое для однопроходной схемы растет с ростом размерности; это означает, что скорость уменьшения эффективности данной вычислительной схемы меньше скорости уменьшения эффективности классического преобразования.

Была также проанализирована эффективность реализации методологического подхода к процедурам BLAS уровня 2 на примере матрично-векторного умножения. Эксперименты показали, что применение MBV значительно ускоряет работу ПС, на небольших размерностях выигрыш составляет до 3,7 раз. Такое поведение объясняется тем, что кроме эффекта прологов и эпилогов, который сказывается в основном на небольших размерностях, MBV позволяет избежать невыровненного доступа к данным, что оказывает положительный эффект на обработку массивов любых размеров, формируя асимптотическое ускорение.

В целом необходимо отметить, что данный подход может быть использован для любой ширины SIMD-регистров, причем ожидается, что эффект будет усиливаться с ростом ширины регистров.

Были изучены реализации методологического подхода для задач обработки числовых последовательностей, проведены вычислительные эксперименты, доказана эффективность предложенной методологии по сравнению с реализациями, полученными в результате автоматической векторизации кода. Показана неэффективность традиционных общепринятых методов оптимизации программного обеспечения, применяемых без проведения дополнительного анализа, который должен предшествовать собственно разработке программы, осуществляться с использованием методов статического анализа исходного кода и предложенных методов МУСК и MBV.

5. АРХИТЕКТУРА И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СИСТЕМЫ ПОДДЕРЖКИ ЖИЗНЕННОГО ЦИКЛА ПРОЕКТИРУЕМЫХ ПС

Предложены концепция и архитектура системы поддержки жизненного цикла проектируемых ПС, которая позволяет выбирать проектные решения по повышению эффективности ПС. СП ЖЦ состоит из трех основных подсистем: подсистема формирования характеристик кода, подсистема оценки эффективности кода, подсистема поддержки проектирования ПС.

В разделе 5.1 приводится описание архитектуры системы, описываются модули системы, приводятся алгоритмы работы, скриншоты программной реализации системы.

В разделе 5.2 приводятся результаты апробации предложенной методологии проектирования эффективных ПС для решений инженерных задач. Приводится достигнутое повышение эффективности ПС согласно актам внедрения (использования) результатов диссертационной работы, описываются результаты внедрения на отечественных предприятиях.

В разделе 5.3 приведена реализация геометрического преобразования отражения для прототипа высокопроизводительного геометрического ядра, полученная на основе алгоритмов обработки разреженных мелко-блочных матриц и блоков векторов, полученных в процессе апробации методологии проектирования эффективных ПС.

В 5.4 приводятся основные результаты и выводы по главе 5.

5.1 Система поддержки жизненного цикла проектируемых ПС

В рамках диссертационного исследования была разработана концепция системы поддержки жизненного цикла (СП ЖЦ) проектируемых программных систем, предназначенной для автоматизации и сопровождения процесса создания эффективного программного обеспечения. Данная система реализует предложенную методологию проектирования, объединяя в едином

контуре модели и методы статического и имитационного анализа, оценки эффективности и поддержки принятия проектных решений

5.1.1 Архитектура системы

Архитектура системы представлена на рисунке 5.1 и соответствует предложенной методологии проектирования эффективных ПС (глава 3) и основана на взаимодействии СП ЖЦ с проектировщиком ПС.

Подсистема формирования характеристик кода предназначена для выполнения статического анализа исходного кода программы с последующим формированием метрик ПС в модуле оценки эффективности кода.

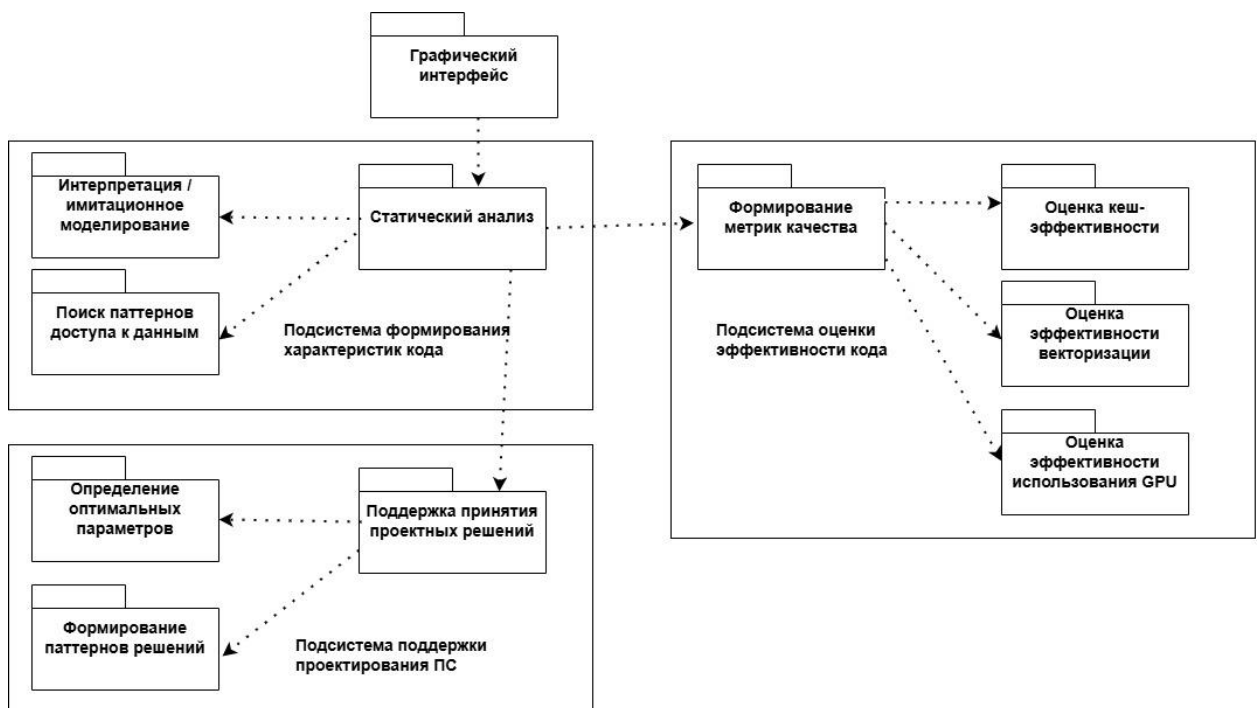


Рисунок 5.1 – Архитектура системы

На первом этапе осуществляется анализ абстрактного синтаксического дерева AST (рисунок 5.2).

В процессе анализа выполняются следующие операции с учетом допущений, сделанных в главе 3:

- получение и преобразование AST;
- поиск и формализация паттернов доступа в память;

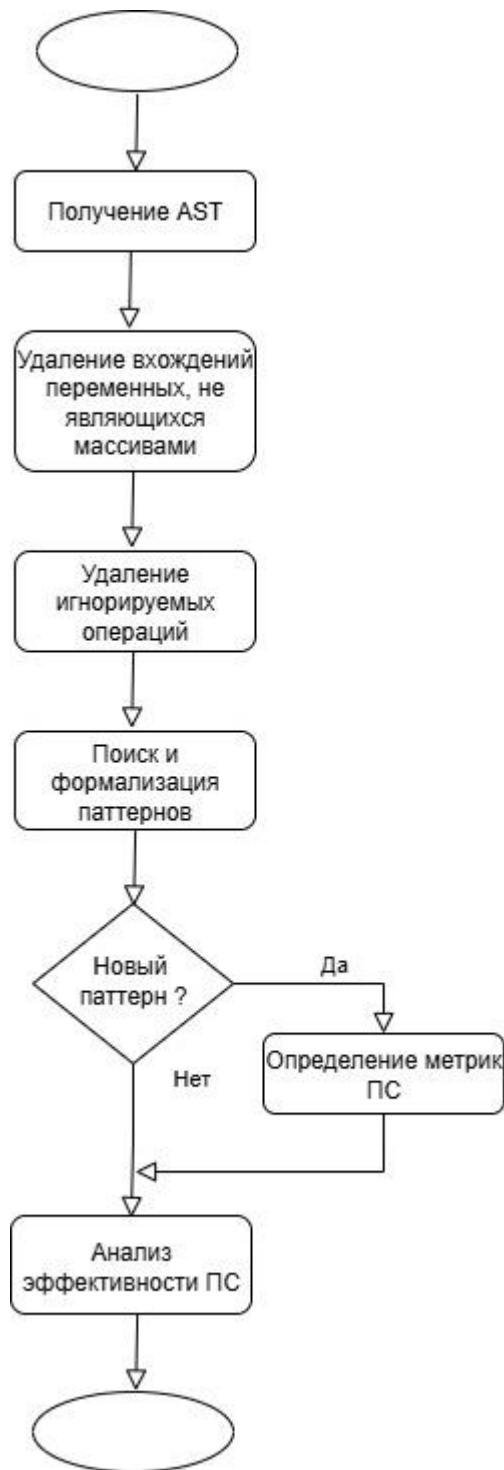


Рисунок 5.2 - Алгоритм анализа AST

- в случае выявления паттернов доступа, анализ которых для ЦВА осуществлялся ранее, и уже были получены необходимые метрики ПС, характеристики данных паттернов используются для анализа эффективности ПС;

- в случае выявления новых паттернов доступа, анализ которых для ЦВА не осуществлялся, и информация о которых отсутствует в системе; для

них производится имитационное моделирование с целью определения метрик ПС, полученные параметры запоминаются для повторного использования.

5.1.2 Подсистема интерпретации и имитационного моделирования

Имитационное моделирование осуществляется с использованием модуля «Интерпретация/имитационное моделирование» путем интерпретации исходного кода программы с формированием и последующей обработкой текущих трасс. Возможны два режима интерпретации исходного кода:

- упрощенный вариант интерпретации, используемый для формирования метрик кеш-эффективности ПС;
- полный вариант интерпретации, используемый для формирования метрик эффективности векторизации ПС.

Предложенная методология позволяет накапливать дополнительную информацию о выполняемом коде благодаря возможности передачи данной информации в трассах. В таблице 5.1 приведен формат трассы, используемый в модуле «Интерпретация/имитационное моделирование».

В результате разбора строки трассы подсистема «Интерпретация/имитационное моделирование» получает полное представление о выполняемой операции и возможность дальнейшей обработки данной информации. Необходимо отметить/, что в данном модуле предусмотрено:

- частичный анализ трасс; на рисунке 5.3 приведен пример работы модуля с анализом параметров «адрес» и «массив»;
- формирование различных вариантов трасс (таблица 5.2) с сохранением во внешнем файле трасс для использования стороннего кеш-симулятора.

Таблица 5.1. Формат трассы, используемый в модуле «Интерпретация/имитационное моделирование»

Трасса	Описание
100 c r func_1	адрес / массив / операция / функция
100 c r func_1	адрес – адрес строки кеш-памяти, к которой
0 a r func_1	осуществляется обращение
80 b r func_1	массив – название массива, к которому осуществляется
100 c w func_1	обращение
0 a r func_1	операция – чтение / запись
480 b r func_1	функция – название функции, в которой осуществляется
100 c r func_1	обращение

Таблица 5.2. Варианты различных форматов трасс

Трасса	Описание
100	Адрес
100 r	адрес / операция
100 c r	адрес / массив / операция

На рисунке 5.3 приведен пример работы модуля «Интерпретация/имитационное моделирование» с частичным анализом трасс. Рассчитываются следующие параметры:

- общее количество кеш-промахов;
- общее количество кеш-попаданий;
- количество кеш-промахов по каждому из массивов в отдельности (в примере использовался код с тремя массивами – a, b, c);
- общее количество обращений к памяти, представляющее собой сумму количества кеш-промахов и кеш-попаданий.

На основе рассчитанных параметров вычисляются значения метрик ПС. На рисунке 5.3 приведен пример формирования метрики ПС – усредненного значения вероятности кеш-промаха. Для формирования других метрик ПС в процессе моделирования рассчитываются необходимые параметры.


```

Кеш L1
Размер кеша 32 КБ 8-канальный
Размер кэш-линии 64
Обращения к кешу: 541712106
Попадания в кеш: 540001556 (запись - 54523726 , чтение - 485477830)
Промахи кеша: 1710550 (запись - 16877 , чтение - 1693673)
Частота промахов: 0.315767
Промахи кеша (массив a): 11249 (запись - 5625, чтение - 5624)
Промахи кеша (массив b): 1693675 (запись - 5626, чтение - 1688049)
Промахи кеша (массив c): 5625 (запись - 5625, чтение - 0)

Кеш L2
Размер кеша 256 КБ 8-канальный
Размер кэш-линии 64
Обращения к кешу: 1710550
Попадания в кеш: 1063 (запись - 0 , чтение - 1063)
Промахи кеша: 1709487 (запись - 16877 , чтение - 1692610)
Частота промахов: 99.9379
Промахи кеша (массив a): 11249 (запись - 5625, чтение - 5624)
Промахи кеша (массив b): 1692612 (запись - 5626, чтение - 1686986)
Промахи кеша (массив c): 5625 (запись - 5625, чтение - 0)

```

Рисунок 5.3 - Пример работы модуля с анализом параметров «адрес» и «массив»

Для подтверждения достоверности и оценки эффективности предложенной методологии были проведены вычислительные эксперименты, в ходе которых в СП ЖЦ метрики анализируемой ПС оценивались методом имитационного моделирования и сопоставлялись с результатами, выдаваемыми существующими инструментами анализа эффективности использования подсистемы памяти ПС.

В качестве анализируемой ПС была выбрана классическая реализация умножения матриц с тройным вложенным циклом.

Для исследования программных реализаций алгоритмов умножения матриц использовались:

- архитектурный симулятор gem5 для имитационного моделирования выполнения программ на целевой иерархии памяти;
- инструмент Cachegrind пакета Valgrind для динамического анализа эффективности работы кеш-памяти.

Все программы запускались в операционной системе Ubuntu 24.04.1 LTS, развернутой в подсистеме Windows Subsystem for Linux 2 (WSL2) под управлением Windows 10. Компиляция осуществлялась GCC 13.3.0. Эксперименты проводились на вычислительной системе на базе микропроцессора 11th Gen Intel(R) Core(TM) i9-11900 @ 2.50GHz.

В процессе анализа в обоих случаях использовалась следующая конфигурация подсистемы кеш-памяти:

- число уровней - 2;
- тип иерархии — инклюзивная (строка, присутствующая в кеше L1, считается одновременно находящейся и в кеше следующего уровня L2);
- алгоритм вытеснения — LRU (least recently used), т.е. при необходимости освобождения места из кеша удаляется строка, которая дольше всех не использовалась;
- способ записи — write-back, при котором модифицированные данные сначала накапливаются в кеше и записываются во внешнюю память только при вытеснении строки;
- поведение при промахе по записи — write-allocate, при котором при промахе по записи соответствующая строка сначала загружается в кеш, и только затем операция записи выполняется в кеше (а не напрямую в основную память или следующий уровень кеша).

Характеристики уровней следующие:

- L1: размер 32 Кб, 8-канальная ассоциативность;
- L2: размер 256 Кб, 8-канальная ассоциативность.

Для обоих уровней длина кеш-строки составляет 64 байта.

Запуск симуляции в gem5 для исследуемой программы осуществлялся командой:

```
gem5/build/ALL/gem5.opt gem5/configs/deprecated/example/se.py \
--cmd=./a \
--cpu-type=X86O3CPU \
--caches --l2cache \
```

```
--l1d_size=32kB --l1d_assoc=8 \
--l1i_size=32kB --l1i_assoc=8 \
--l2_size=256kB --l2_assoc=8 \
--mem-type=DDR3_1600_8x8 --mem-size=1GB
```

Выполнение команды приводит к запуску моделирования в симуляторе `gem5` в режиме эмуляции системных вызовов (SE-mode) с архитектурой x86 и следующей конфигурацией процессора, кеш-подсистемы и оперативной памяти:

- одно вычислительное ядро с моделью X86O3CPU (архитектура x86 из набора моделей `gem5`, out-of-order суперскалярное ядро);
- отдельные кешы первого уровня инструкций и данных объёмом 32 Кб, ассоциативностью 8 (размер строки 64 байта по умолчанию для `se.py`);
- общий кеш второго уровня объёмом 256 Кб, ассоциативностью 8;
- основная память типа DDR3-1600 объёмом 1 Гб.

Для проведения имитационного моделирования использовался стандартный конфигурационный сценарий `se.py`, входящий в состав дистрибутива `gem5` и предназначенный для режима эмуляции системных вызовов (SE-mode). Сценарий `se.py` формирует модель однопроцессорной системы архитектуры x86 и позволяет из командной строки задавать тип процессорного ядра, параметры кеш-подсистемы и оперативной памяти, что упрощает проведение серийных экспериментов с различными конфигурациями вычислительной системы.

Моделирование умножения матриц выполнялось с учётом различных вариантов размещения данных: автоматических массивов (на стеке), статических массивов и динамически выделяемых массивов (в куче), что иллюстрируется рисунком 5.4.

Симулятор `gem5` позволяет оценить только общее число кеш-промахов по категориям, зато сделать это достаточно точно для выбранной целевой архитектуры, т.к. при выполнении программы `gem5` пошагово эмулирует каждое обращение к памяти с учётом заданной конфигурации

кеш-подсистемы и алгоритма вытеснения, без аналитических допущений и упрощений.

int matrix_mul()	int a[10000];	int matrix_mul()
{	int b[10000];	{
int a[10000];	int c[10000];	int *a = (int*)malloc(10000 * sizeof(int));
int b[10000];	int matrix_mul()	int *b = (int*)malloc(10000 * sizeof(int));
int c[10000];	{	int *c = (int*)malloc(10000 * sizeof(int));

a)

б)

в)

Рисунок 5/4 - Варианты размещения данных: автоматические массивы а), статические массивы б), динамические массивы в).

В процессе моделирования собирались данные о промахх по чтению и записи на двух уровнях кеш-памяти для всех трёх вариантов размещения данных (автоматические, статические и динамические массивы). Для каждого параметра оценивалась разница между значениями, полученными СП ЖЦ в соответствии со следующим выражением.

$$diff = \frac{abs(V_{СП ЖЦ} - V_{gem5})}{V_{gem5}} \quad (5/1)$$

где

$V_{СП ЖЦ}$ - значение параметра, полученное СП ЖЦ;

V_{gem5} - значение параметра, полученное gem5.

Полученные результаты приведены на рисунках 5.5 и 5.6

На рисунке 5.5 видно, что при малых размерностях (100) для вариантов auto и static относительное отклонение оценок СП ЖЦ от результатов gem5 составляет порядка 5–7 %, а на размерности 400 разница возрастает до ≈10 %. На остальных размерностях расхождение не превышает 2 %. Увеличение отклонения при N=100 объясняется повышенной «зашумлённостью» измерений: из-за сравнительно небольшого числа вычислительных операций существенно возрастает относительный вклад служебных обращений к

памяти (стек, библиотечный код и т.п.), которые не моделируются в СП ЖЦ, но учитываются в симуляторе gem5.

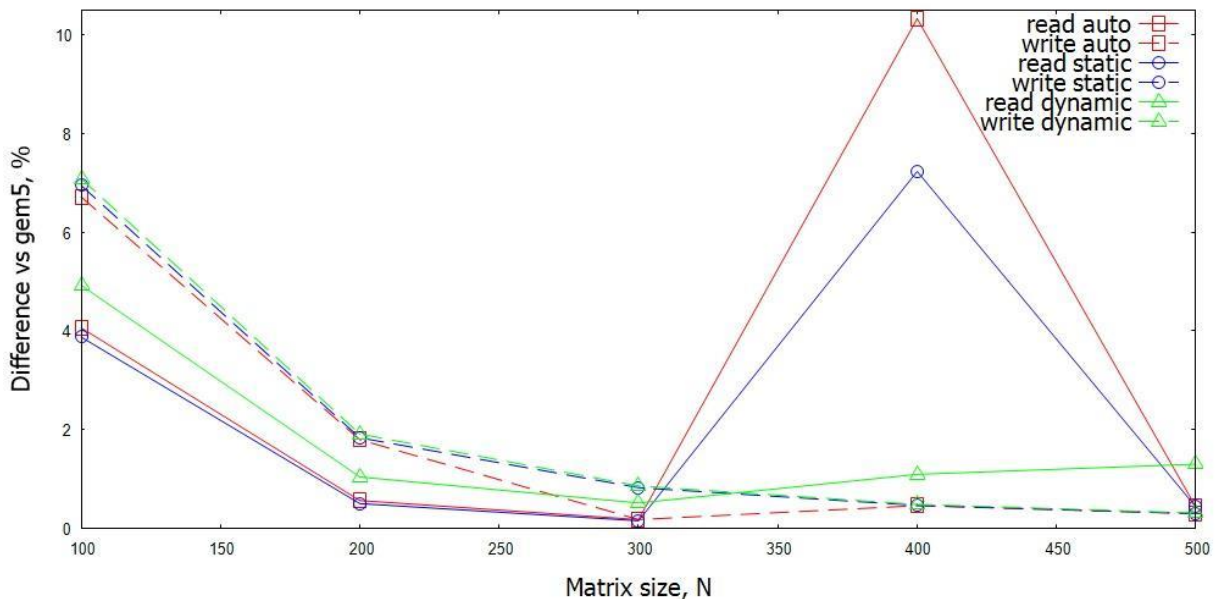


Рисунок 5.5 – Сравнение СП ЖЦ и gem5 по числу промахов в кеше L1.

На размерности 400 модель СП ЖЦ предсказывает существенно больше промахов L1 по массиву a, чем измеряется в gem5. Это связано с тем, что при данных размерностях длина строки матрицы (в строках кеша) и геометрия кеша (число наборов и ассоциативность) образуют неудачную с точки зрения модели комбинацию, при которой кеш-строки различных строк матрицы формально проецируются в ограниченное подмножество наборов и считаются часто конфликтующими. Фактическая последовательность обращений к памяти, моделируемая gem5 (с учётом реальных базовых адресов массивов, выравнивания и дополнительных обращений к стеку и другим данным), приводит к меньшему числу конфликтов, чем предполагает схема моделирования. Таким образом, на этих размерностях модель даёт завышенную, но консервативную оценку числа промахов.

Видно, что для динамических массивов за исключением малых размерностей (100) отклонение не превышает 1%. Такое отклонение является незначительным, поэтому можно считать, что для динамических массивов обе системы дают совпадающие оценки. В научных и инженерных приложениях в подавляющем большинстве случаев используются

динамические массивы, для которых расхождение между оценками СП ЖЦ и gem5 не превышает 1 %. Поэтому с учётом результатов gem5 можно считать, что система СП ЖЦ даёт достоверные оценки числа кеш-промахов для практически значимых вариантов размещения данных.

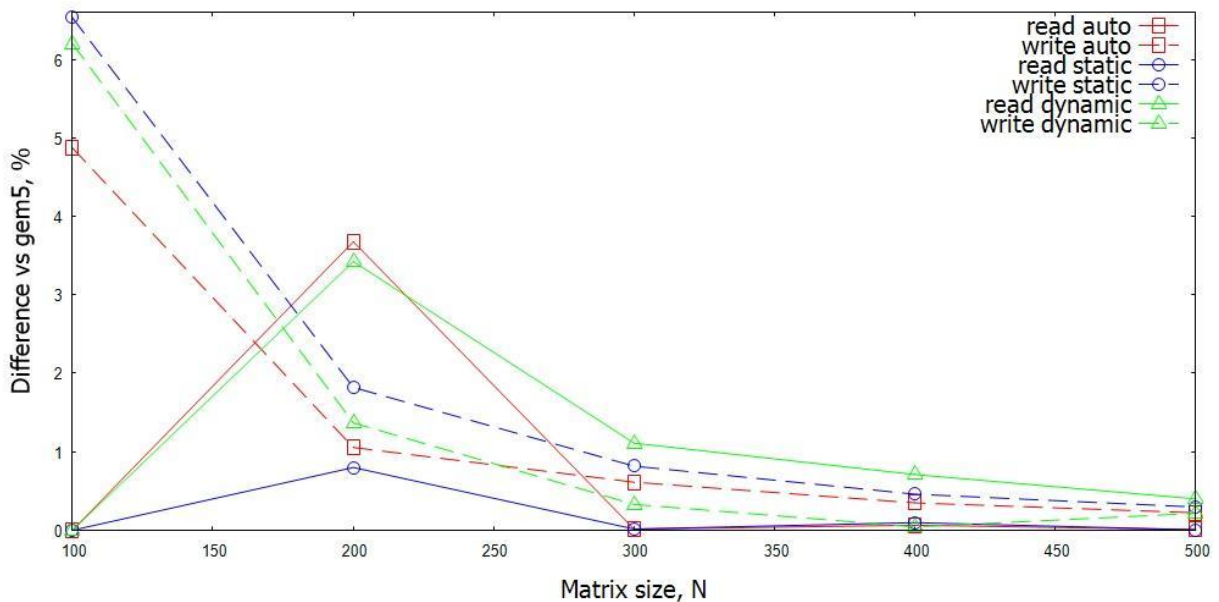


Рисунок 5.6 – Сравнение СП ЖЦ и gem5 по числу промахов в кеше L2.

Для кеша L2 результаты похожи для всех вариантов размещения данных: те же отклонения на малых размерностях, с дальнейшим выравниванием результатов, причем для размерности 300 и выше результаты практически идентичны и не превышают 1%..

Сопоставление результатов моделирования в системе СП ЖЦ с архитектурным симулятором gem5 показало, что для всех исследованных размерностей матриц и вариантов размещения данных оценки числа промахов L1/L2 и относительные тенденции совпадают практически полностью. В большинстве случаев расхождение не превышает нескольких процентов, а для отдельных «резонансных» размерностей носит объяснимый характер и, как правило, имеет консервативный характер (модель несколько завышает число конфликтных промахов).

Поскольку gem5 исполняет реальный бинарный код и детально моделирует микроархитектуру процессора и подсистемы памяти, его результаты можно считать более полными и точными и использовать в

качестве эталона. Стоит отметить, что моделирование с использованием gem5 осуществляется неприемлемо долго и не может быть использовано для оперативного анализа текущей программной реализации в ходе проектирования ПС. Ниже представлены графики зависимости времени моделирования в gem5 от размерности матриц, а также графики отношения времени моделирования в gem5 ко времени моделирования в СП ЖЦ.

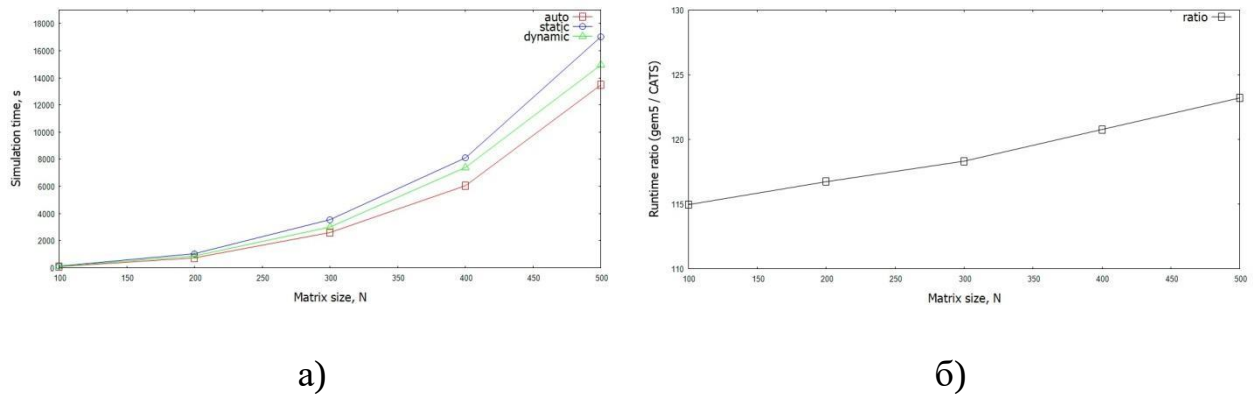


Рисунок 5.7 – Время моделирования в gem5 а) отношение времени моделирования gem5 к СП ЖЦ (динамические массивы) б)

На графике 5а) представлено время моделирования. Например, для размерности 500 (вариант static) gem5 потребовалось около 5 часов (17000 секунд) для завершения моделирования. На графике 5б) показано, что время моделирования СП ЖЦ на 2 порядка меньше (в среднем в 120 раз) аналогичного времени для gem5.

На этом фоне хорошее совпадение с gem5 подтверждает корректность предложенного подхода и позволяет использовать СП ЖЦ для быстрой и достаточно точной оценки эффективности алгоритмов без затратного архитектурного моделирования. Помимо того, СП ЖЦ позволяет анализировать исходный код программы (или любую часть исходного кода) без необходимости выполнения компиляции и отладки.

В таблице 5.3 приведено сравнение СП ЖЦ и gem5.

Таблица 5.3 - Сравнение СП ЖЦ и gem5

Показатель	СП ЖЦ	gem5
Точность моделирования	высокая	высокая
Время моделирования	быстро	долго
Статистика по отдельным массивам	да	нет
Статистика по отдельным функциям, блокам кода	да	нет
Возможность применения на этапе анализа исходного кода	да	нет

Резюмируя всё сказанное выше, можно сказать, что gem5, как и другие архитектурные симуляторы, является мощным средством детального исследования поведения программ на уровне микроархитектуры процессора и иерархии памяти, обеспечивающим высокую точность оценки производительности, но требующим значительных затрат времени на моделирование. На этом фоне СП ЖЦ позволяет получать близкие к gem5 результаты при существенно меньших вычислительных затратах, что делает её более удобной для оперативного анализа и сопровождения проектных решений. Кроме того, предлагаемая система позволяет получать более детальную статистику, которая может быть использована на начальных этапах проектирования ПС (статистика по массивам, функциям, блокам кода) методом статического анализа исходного кода программы.

Valgrind по сравнению с gem5 позволяет получить больше информации, в частности статистику обращений к памяти и промахов кеша для каждой строки исходного кода и каждой функции, с отдельным учётом чтений и записей, а также аннотированные профили, которые напрямую связывают показатели работы кеша и времени выполнения с конкретными участками программы. Это делает невозможным анализ кеш-промахов отдельно по различным массивам, но, путем изменения текста программы

можно примерно оценить значения данных параметров. Ниже приведён пример изменения исходного кода, необходимого для дальнейшего анализа результатов Valgrind.

$c[i*n+j] = 0;$ $for(k = 0; k < n; k=k+1) \{$ $c[i*n+j] = c[i*n+j] + a[i*n+k] * b[k*n+j];$	$\longrightarrow$	$c[i*n+j] = 0;$ $for(k = 0; k < n; k=k+1) \{$ $c[i*n+j] = \backslash$ $c[i*n+j] + \backslash$ $a[i*n+k] * \backslash$ $b[k*n+j];$

После выполнения моделирования необходимо проанализировать файл статистики и сложить показатели строк, содержащих обращения к элементам одних и тех же массивов. Например, в приведенном случае для анализа кеш-промахов по массиву с необходимо будет проанализировать содержимое трех строк. Для больших исходных кодов необходимо будет выполнять как предварительное преобразование текста программы, так и последующий анализ файла статистики, что может стать затратным или приведет к необходимости самостоятельной разработки дополнительных инструментальных средств. Кроме того, необходимо учитывать, что строки, содержащие, например, обращение $c[i*n+j]$, будут содержать суммарную информацию по обращениям к элементам массива c , а также переменным i , n и j .

Запуск Valgrind выполнялся с использованием следующей командной строки.

```
valgrind --tool=cachegrind \
    --cache-sim=yes \
    --I1=32768,8,64 \
    --D1=32768,8,64 \
    --LL=262144,8,64 \
    ./mm
```

Для постобработки результатов Valgrind и получения аннотированного отчёта по функциям и строкам исходного кода использовалась утилита `cg_annotate`, позволяющая детализировать распределение обращений к памяти и промахов кеша по участкам программы. Полученные результаты были сопоставлены с результатами, вычисленными системой СП ЖЦ для кешей L1 и L2, по той же процедуре сравнения, что и для данных, полученных в симуляторе `gem5`.

На рисунке 5.8, для каждого варианта размещения данных (автоматические, статические и динамические массивы) показаны кривые промахов по чтению и промахов по записи.

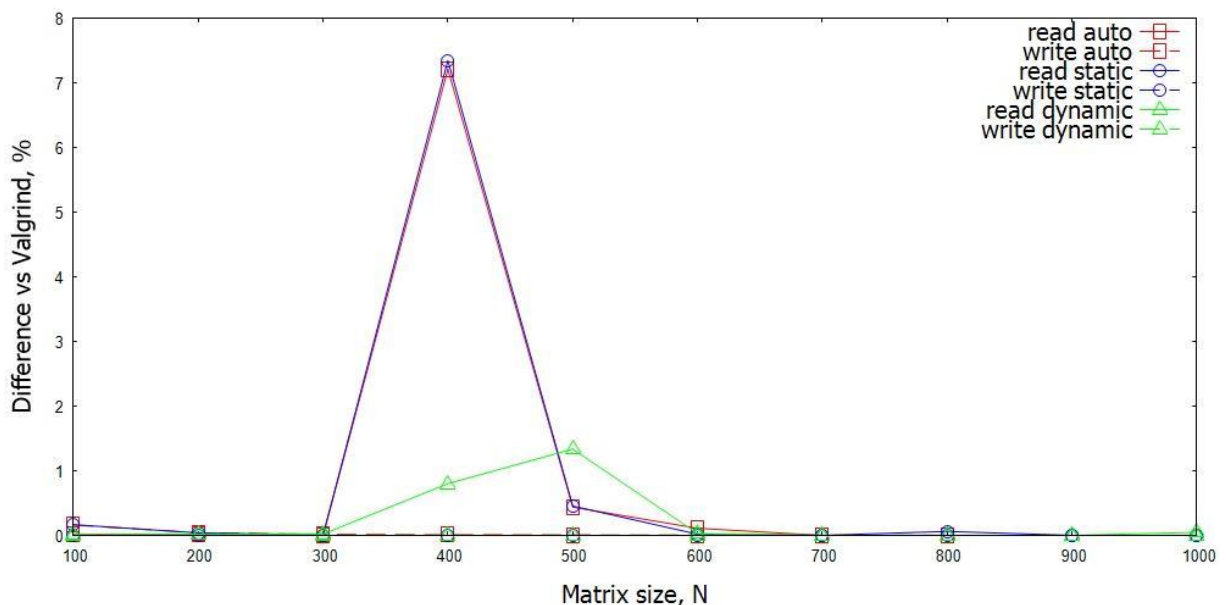


Рисунок 5.8 – Сравнение СП ЖЦ и Valgrind по числу промахов в L1.

Видно, что для всех вариантов размещения данных относительное отклонение результатов СП ЖЦ от результатов Valgrind в большинстве размерностей не превышает долей процента, и в этом смысле обе системы можно считать дающими совпадающие оценки. Однако на каждой кривой для двух размерностей (400 и 500) наблюдаются заметные расхождения. При этом для автоматических и статических массивов максимальное отклонение достигает около 7 %, тогда как для динамических массивов оно не превышает 1,4 %.

Причина здесь та же, что и в случае сравнения с `gem5`: на этих размерностях модель дает завышенную оценку значения кеш-промахов. Максимальное относительное отклонение составляет порядка 7 % для автоматических и статических массивов и 1,4 % для динамических массивов, что позволяет считать оценки СП ЖЦ достаточно точными для практических применений. При этом отклонение 1,4% является незначительным, поэтому можно считать, что для динамических массивов обе системы дают совпадающие оценки.

На рисунке 5.9 приведены результаты сопоставления результатов Valgrind с результатами, вычисленными системой СП ЖЦ для кеша L2 (верхний уровень иерархии в целевой системе).

Так же, как и в случае сравнения с `gem5`, для кеша L2 результаты похожи для всех вариантов размещения данных: небольшие отклонения (до 2%) на малых размерностях, с дальнейшим выравниванием результатов, причем для динамических массивов результаты для размерности 300 и выше практически идентичны и равны 0, для автоматических и статических массивов различия минимальны и также значения близки к нулю.

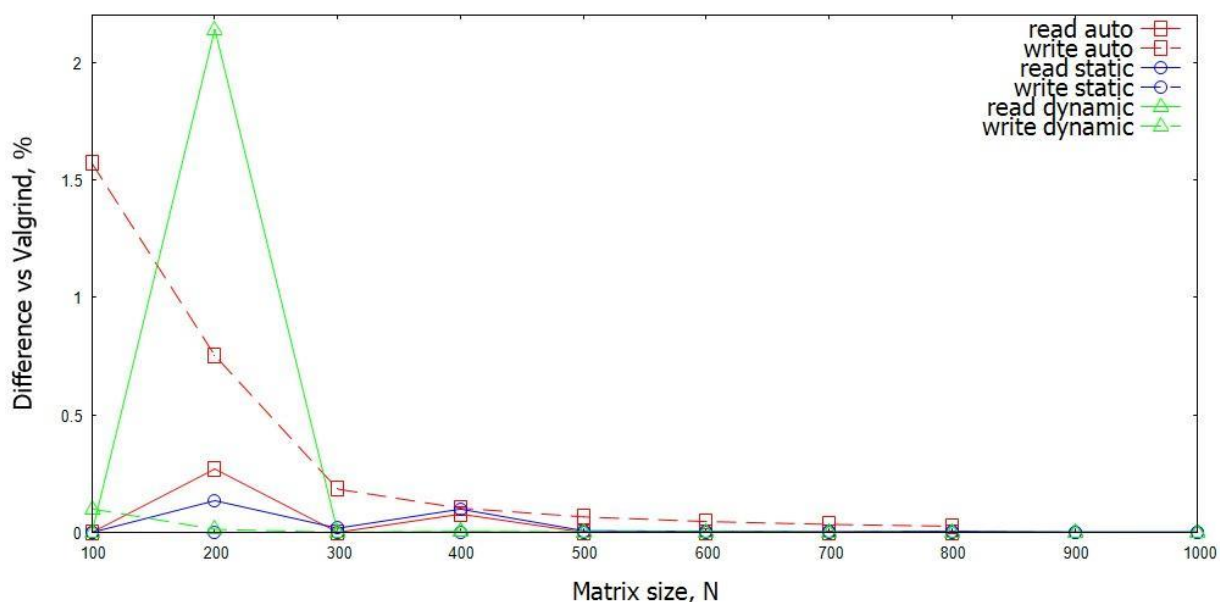
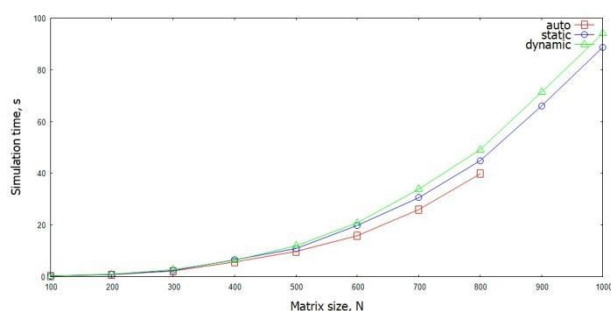


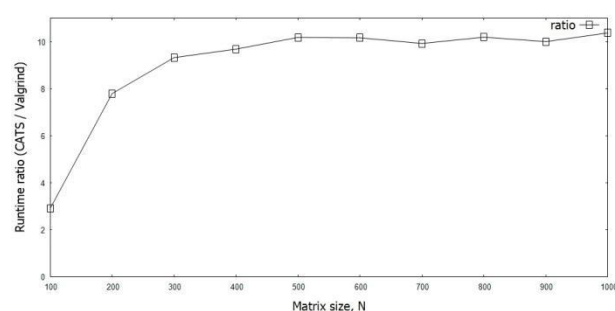
Рисунок 5.9 – Сравнение СП ЖЦ и Valgrind по числу промахов в L2.

Valgrind, в отличие от `gem5`, выполняет моделирование значительно быстрее. На рисунке 5.10 представлены графики зависимости времени

моделирования в Valgrind от размерности матриц, а также графики отношения времени моделирования в СП ЖЦ ко времени моделирования в Valgrind.



а)



б)

Рисунок 5.10 – Время моделирования Valgrind а) Отношение времени моделирования Valgrind к СП ЖЦ (динамические массивы) б)

Анализ графиков, представленных на рисунке 5.10 показывает, что в данном случае уже Valgrind выполняет моделирование быстрее, чем СП ЖЦ. Учитывая данный факт, Valgrind в большей степени, чем gem5, подходит для анализа проектируемой GC в процессе проектирования.

В таблице 5.4 приведено сравнение СП ЖЦ и Valgrind (Cachegrind).

Таблица 5.4 - Сравнение СП ЖЦ и Valgrind (Cachegrind)

Показатель	СП ЖЦ	Valgrind
Точность моделирования	высокая	высокая
Время моделирования	долго	быстро
Статистика по отдельным массивам	да	да/нет
Статистика по отдельным функциям, блокам кода	да	да/нет
Возможность применения на этапе анализа исходного кода	да	нет
Число уровней кеша	Любое	2

Продолжение таблицы 5.4

Показатель	СП ЖЦ	Valgrind
Алгоритм вытеснения	LRU, Random, FIFO, Pseudo-LRU	LRU
Политика записи	write-back, write-through	write-back
Политика размещения при промахе по записи	write-allocate, write-around	write-allocate
Тип иерархии кеша	инклюзивный, эксклюзивный	инклюзивный
Учет трафика памяти	да	нет
Предварительное изменение исходного кода	нет	да
Постобработка файла статистики	нет	да

Инструмент Cachegrind из пакета Valgrind является мощным средством динамического анализа и моделирования подсистемы кеш-памяти; моделирование в нём выполняется быстрее, чем в предлагаемой аналитической модели СП ЖЦ. Однако данный инструмент, как и gem5, не может использоваться на начальных этапах проектирования ПС, так как требует наличия скомпилированного и отлаженного исполняемого файла. Кроме того, Cachegrind присущ ряд ограничений (см. табл. 5.4), что существенно сужает область его применимости. Наконец, для получения приближённых статистических оценок отдельно по различным массивам приходится модифицировать исходный код программы и выполнять дополнительную постобработку сгенерированного файла статистики..

5.1.3 Подсистема оценки эффективности кода

В соответствии с предложенной методологией проектирования эффективных ПС проектировщик на основе анализа вычисленных метрик оценивает кеш-эффективность, эффективность векторизации и эффективность использования GPU, после чего принимает решение о выполнении очередной итерации проектирования или модификации ПС. На данной итерации в соответствии с требованиями, предъявляемыми к проектируемой ПС, а также полученным оценкам ее эффективности применяются методы повышения эффективности ПС (глава 3):

- улучшение стратегии кеширования путем реализации МУСК;
- повышение эффективности векторизации вычислений путем реализации MBV;
- повышение эффективности использования GPU путем реализации предложенного метода.

Реализация методов повышения эффективности ПС для разработки алгоритмов вычислительно-сложных процедур описана в главе 4, на рисунках 5.11 и 5.12 продемонстрированы вычислительные эксперименты с определением энергопотребления и времени выполнения в процессе реализации МУСК и MBV.

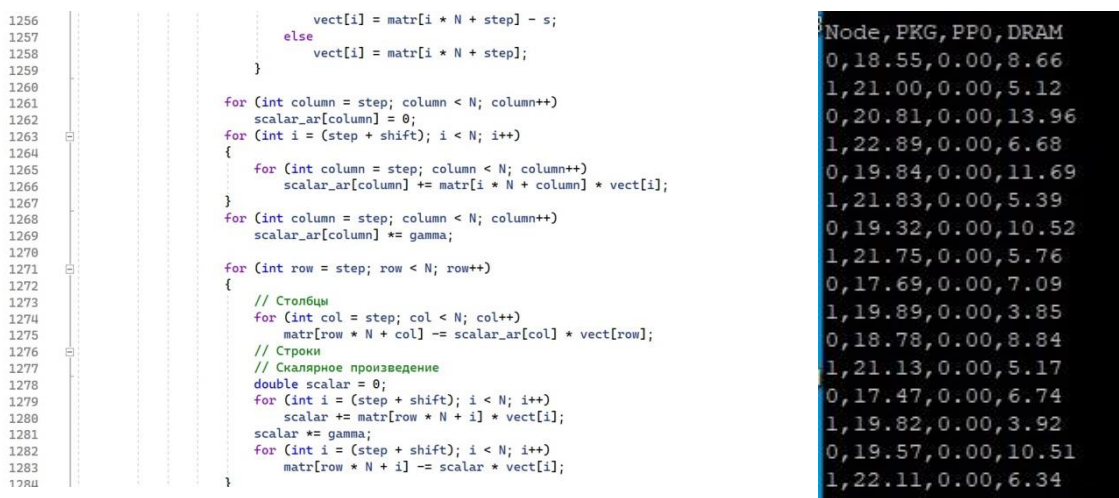


Рисунок 5.11 – Программная реализация МУСК, эксперимент с определением энергопотребления

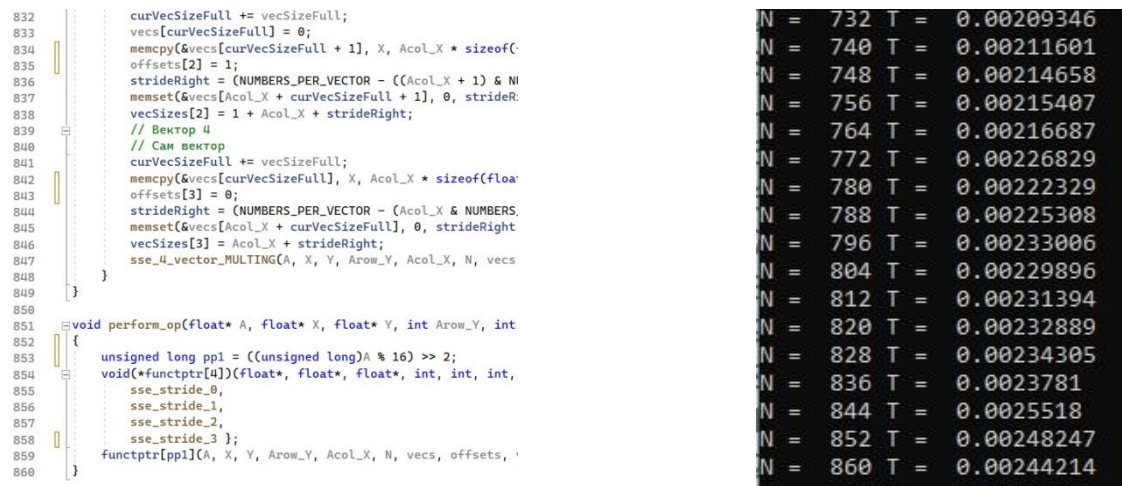


Рисунок 5.12 – Программная реализация MBV, эксперимент с определением времени выполнения

Для найденных паттернов доступа при необходимости реализуется метод определения количественных характеристик метрик ПС, который позволяет оценить вероятность кеш-промахов при обработке массивов заданного размера и максимальный размер блоков данных, при котором генерируются только «холодные» промахи и не происходит вытеснения строк из кеш-памяти.

Для СП ЖЦ был разработан графический интерфейс, на рисунке 5.13 приведена основная экранная форма.

Пользователю доступны несколько функций:

- загрузка исследуемого кода ПС;
- задание характеристик ВС;
- запуск анализа;
- сохранение результата.

В примере на рисунке 5.13 приведен анализ фрагмента ПС, реализующего скалярное умножение части вектора на строки матрицы.

Заданы следующие характеристики ВС:

- ВС имеет трехуровневую кеш-память;
- кеш-память L3 является разделяемой, L1 и L2 нет;
- тип кеш-памяти – множественно-ассоциативная;

- степень ассоциативности для L1 и L2 составляет 8, для L3 составляет 16;

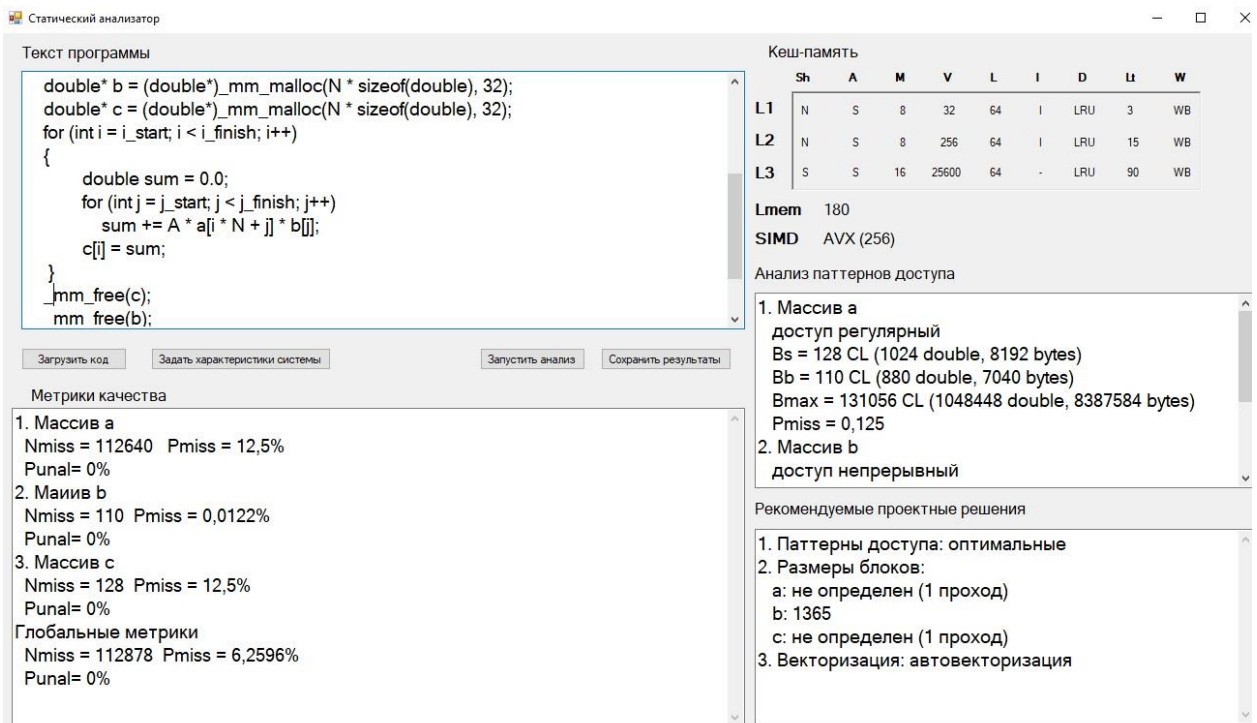


Рисунок 5.13 – Экранная форма СП ЖЦ

- размер кеш-памяти составляет 32К для L1, 256К для L2, 25600К для L3;

- длина кеш-линии для всех уровней кеш-памяти составляет 64 байта;

- уровни L1 и L2 построены по инклюзивному принципу;

- на всех уровнях используется алгоритм вытеснения Least Recently Used (LRU);

- латентность различных уровней кеш-памяти составляет 3, 15 и 90 тактов ядра;

- для всех уровней кеш-памяти используется политика записи write back;

- используется технология векторизации AVX.

В результате были получены значения метрик ПС, проанализированы используемые паттерны доступа к данным и рекомендуемые проектные решения. Перечень рассчитываемых метрик задается в разделе «Задать

характеристики системы». В примере на рисунке 5.13 рассчитаны следующие метрики:

- количество кеш-промахов;
- вероятность кеш-промаха;
- доля невыровненных обращений.

Значения метрик рассчитываются отдельно для каждого массива и в целом для всего анализируемого блока кода. Отдельно проанализированы паттерны доступа и сформулированы рекомендуемые проектные решения.

Характеристики паттернов доступа приводятся в нотации, использованной в главе 3. Кроме того, рассчитывается максимальная длина блока данных, считываемых из массива в цикле и значение вероятности «холодных» кеш-промахов в «заполненной» части паттерна.

В рекомендованных проектных решениях формулируются:

- оптимальность паттернов доступа или необходимость замены на более эффективные паттерны;
- максимальные размеры блоков для всех массивов (в примере на рисунке 5.13 оценивается размер блока только для массива *b*, для двух других массивов размер такого блока не определен, т.к. используется один проход вдоль массивов; для определения размера блока использовался метод определения максимального размера блока при регулярном доступе, глава 3);
- рекомендуется проведение автоматической векторизации без применения MBV.

В процессе разработки архитектуры и программного обеспечения СП ЖЦ были получены свидетельства о регистрации программы для ЭВМ, которые приведены в Приложении А.

5.2 Апробация методологии проектирования эффективных ПС

Предложенная методология проектирования эффективных ПС для решения инженерных задач была реализована в процессе создания и

оптимизации программного обеспечения, разрабатываемого компанией «ТЕСИС» для моделирования гидроаэродинамики и тепловых процессов, а также решения междисциплинарных задач, компанией «Топ Системы» для системы управления полным жизненным циклом изделий, компанией ОАО «Волжский абразивный завод» для автоматизации сбора и обработки данных с производственных участков электроплавильного цеха, а также многокритериального анализа факторов, влияющих на объем и качество выпускаемой продукции. Оценка эффективности реализации методологии приведена в таблице 5.5.

Таблица 5.5. Достигнутое повышение эффективности согласно актам внедрения (использования) результатов диссертационной работы.

Организация	Эффективность
АО «Топ Системы»	Уменьшение времени, необходимого для реализации базовых расчетных процедур до 18%, повышение точности вычислений до 7%.
ОАО «Волжский абразивный завод»	Повышение эффективности сбора и обработки данных производственных участков электроплавильного цеха на 17-20%
ООО «ТЕСИС»	Ускорение выполнения базовых операций до 10 раз относительно предыдущих программных реализаций при использовании технологии векторизации вычислений AVX2. Значительное, до 50-75%, уменьшение времени вычислений над плотными и разреженными матрицами при использовании графических сопроцессоров. Повышение скорости до 15% и точности до 5-7% расчетов при реализации оптимизированных алгоритмов решения систем линейных уравнений с разреженными или плохо обусловленными матрицами показало.

5.2.1 Внедрение в ООО «ТЕСИС»

Предложенные методы использованы в процессе создания и оптимизации программного обеспечения, разрабатываемого компанией «ТЕСИС» для моделирования гидроаэродинамики и тепловых процессов, а также решения междисциплинарных задач, в частности – при разработке прототипов расчетных библиотек. Исследования начинались в 2013 - 2015 годах в рамках ряда НИР, в том числе «Разработка и анализ алгоритмов реализации матричных операций с использованием векторных расширений систем команд процессоров», «Разработка и тестирование алгоритмов реализации блочных матричных операций с использованием векторных расширений систем команд процессоров» и «Модификация алгоритмов реализации матричных операций с использованием расширений систем команд, дополненных совмещенными сложениями – умножениями (FMA3), и архитектур процессоров с длинным командным словом», и нашли свое продолжение в 2024 году в рамках предпроектных исследований применения графических сопроцессоров.

Разработанные в рамках НИР алгоритмы использовались для выполнения базовых блочных операций линейной алгебры (умножение матриц, обращение и факторизация матриц) для разреженных мелко-блочных матриц и блоков векторов (плотных матриц определенной размерности) с применением векторных расширений системы команд микропроцессоров Intel и использованием возможностей графических сопроцессоров. На основе базовых операций компанией «ТЕСИС» были разработаны алгоритмы умножения разреженных мелко-блочных матриц на блоки векторов, LU-разложения, обращения матриц и решения систем линейных алгебраических уравнений.

Для повышения эффективности программных реализаций базовых операций были использованы векторные системы команд SSE, AVX, AVX2, AVX-512 и MIC (Xeon Phi), в качестве базовых типов данных использовались

float и double. Было проведено исследование возможной эффективной реализации базовых операций за счет рефакторинга программного кода и использования оптимизирующего компилятора процессоров архитектуры "Эльбрус".

В процессе выполнения работ были разработаны реализации операций линейной алгебры с использованием векторных регистров XMM, YMM, ZMM, проведено тестирование разработанных реализаций и их сравнение с не векторизованными версиями, в т.ч.: разработаны алгоритмы реализации базовых операций, проведено тестирование разработанных алгоритмов, проанализированы временные результаты тестов, получено ускорение относительно исходной реализации.

Реализация разработанных алгоритмов и методов позволила получить значительное ускорение выполнения базовых операций до 10 раз относительно предыдущих программных реализаций при использовании технологии векторизации вычислений AVX2. Разработанные алгоритмы и методы, использующие графические сопроцессоры, позволяют значительно, до 50-75%, уменьшить время вычислений над плотными и разреженными матрицами за счет распределения вычислений между центральным процессором и графическим сопроцессором, оптимизации вычислений под конкретные платформы, интеграции программного обеспечения с существующими высокопроизводительными программными библиотеками. Реализация оптимизированных алгоритмов решения систем линейных уравнений с разреженными или плохо обусловленными матрицами показало повышение скорости до 15% и точности до 5-7% расчетов.

Кроме того, полученные наработки в области векторизации вычислений и распределения вычислений между центральным процессором и графическим сопроцессором компания «ТЕСИС» предполагает использовать при построении вычислительных библиотек для программного комплекса «FlowVision» в 2025 – 2027 г.

Акт о внедрении (использовании) результатов диссертационной работы приведен в Приложении Б.

5.2.2 Апробация в АО «Топ Системы»

Предложенный подход к повышению эффективности ПС используется в АО «Топ Системы» с 2019 года в инструментальных компонентах, разработанных для системы управления полным жизненным циклом изделий для промышленных предприятий «САРУС».

Разработанные алгоритмы и методы использованы в библиотеках классов, предназначенных для решения геометрических задач прототипа CAD-системы 3-мерного моделирования:

- построения эквидистантных линий к проволочному телу;
- реализации операций вычисления масс-инерционных характеристик проволочных, твердых и листовых тел с использованием интегрирования по сетке;
- оптимизации ряда базовых расчетных классов библиотеки геометрических операций, связанных с реализацией булевых операций с телами; кинематических операций, построения пересечений и поиска самопересечений и оптимизацией частных случаев обрезки кривых и поверхностей.

Разработан алгоритм, выполняющий построение эквидистанты плоской кривой, состоящей из нескольких сегментов. Работа алгоритма основана на последовательной обработке пар смежных эквидистантных сегментов. Так для кривой, состоящей из n сегментов, будет образовано $n - 1$ пар. При этом кривая может быть замкнута, тогда число пар будет равно числу сегментов. При рассмотрении взаиморасположения смежных эквидистантных сегментов были выделены следующие базовые случаи:

- сопряжение;
- пересечение;

- разрыв;
- вырождение.

Были разработаны алгоритмы реализации как отдельных базовых случаев, так и их взаимосвязи. Выявление таких взаимосвязей важно не только по причине того, что их обработка требуется для корректного построения, но и потому что это может повысить производительность алгоритма за счёт упрощения обработки некоторых случаев.

Использование указанных результатов позволило повысить эффективность программного обеспечения, уменьшить время, необходимое для реализации базовых расчетных процедур до 18%, повысить точность вычислений до 7%.

Акт об использовании результатов диссертационной работы приведен в Приложении Б.

5.2.3 Внедрение в ОАО «Волжский абразивный завод»

Произведена автоматизация деятельности подразделений ОАО «Волжский абразивный завод», осуществляющих сбор и обработку данных с производственных участков электроплавильного цеха (ЭПЦ). Основными функциями разработанной программно-информационной системы являются:

- ведение баз данных оборудования ЭПЦ и исполнителей;
- настройка параметров информационной системы;
- импорт данных в базу данных из различных источников, формируемых в производственном процессе;
- ведение логов операций;
- проведение многокритериального анализа факторов, влияющих на объем и качество выпускаемой продукции;
- формирование отчетов и графиков в аналитической части системы.

Использование указанных результатов позволило повысить эффективность сбора и обработки данных ЭПЦ на 17-20%.

Разработанная программно-информационная система доказала свою работоспособность, полученные результаты утверждены заказчиком.

Акт о внедрении (использовании) результатов диссертационной работы приведен в Приложении Б.

5.3 Реализация геометрического преобразования отражения для прототипа высокопроизводительного геометрического ядра

Результаты, полученные в алгоритмах обработки разреженных мелко-блочных матриц и блоков векторов, были использованы для реализации геометрического преобразования отражения для прототипа высокопроизводительного геометрического ядра.

Под геометрическим ядром в данном случае понимается набор библиотек, предназначенных для реализации функций геометрического моделирования и используемых для решения задач вычислительной геометрии. К наиболее известным геометрическим ядрам можно отнести Parasolid, Open CASCADE. Из отечественных разработок можно отметить ядро C3D, которое лежит в основе Компас-3D, и RGK (Russian Geometric Kernel) [38, 191, 195]. Программные продукты для проектирования и компьютерного инжиниринга (CAD, CAM, CAE и пр.) являются базой для выполнения цифровой трансформации [188], которая успешно развивается в РФ как в последние годы, так и в настоящее время [139]. Для таких задач ключевым параметром является оценка сложности используемых алгоритмов [161].

Одной из важных характеристик геометрических ядер является возможность работать с разным представлением данных, что определяет возможности ядра по чтению и загрузке моделей, хранящихся в различных геометрических форматах, Речь здесь может идти как о форматах, разработанных специально для конкретной САПР, так и об общедоступных форматах. Из общедоступных форматов можно перечислить следующие:

STL, OBJ, COLLADA, FBX, STEP. Стоит отметить, что наиболее популярными являются STL [117] и COLLADA. У некоторых общедоступных форматов существует проблема универсальности, так как для них нет точного описания спецификации.

Полигональная модель описывается набором полигонов и нормалей [153]. В качестве полигона может выступать любой плоский многоугольник, однако на практике используется треугольный полигон. Так, полигональная модель с N полигонами содержит N нормалей и $N * 3$ вершин. Таким образом, полигональную модель можно представить в виде матриц, показанных на рисунке 5.14а) (широкая матрица) и рисунке 5.14б) (высокая матрица).

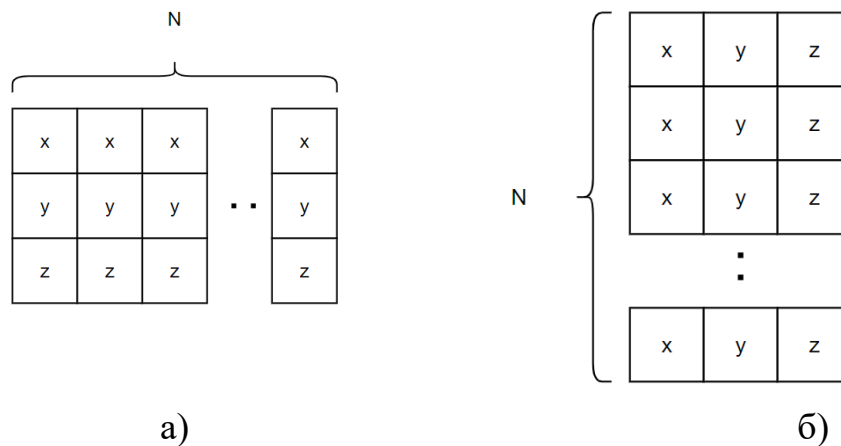


Рисунок 5.14 – Представление в виде матриц: а) $3 \times N$, (широкая матрица), б) $N \times 3$ (высокая матрица), N – число вершин

Оба матричных представления имеют свои достоинства и недостатки, поэтому в разных алгоритмах будет использоваться представление, описание которого по структуре и по расположению данных в памяти ВС является более предпочтительным для рассматриваемого алгоритма.

К примеру, для широкого представления в кеш-память попадёт массив однотипных координат (x , y или z), а для высокого представления в кеш-памяти будет находиться набор вершин, что для линейных преобразований таких, как вращение и отражение, может значительно снизить время выполнения алгоритма. Непосредственно само полигональное описание 3D-модели может использоваться во многих областях [6].

Далее в качестве оптимизируемой операции рассматривалось преобразование отражение Хаусхолдера (глава 4).. Данная операция выполняется путем умножения матрицы отражения на набор векторов, которые могут быть записаны в виде матрицы $3 \times N$ или $N \times 3$.

Для повышения производительности операция отражения была векторизована. Для векторизации использовались интринсики SSE. Поскольку работа осуществляется с вершинами в трёхмерном пространстве, то использование регистров xmm усложняется тем, что для типа float и double в них можно поместить по 4 и 2 элемента соответственно. Одна вершина описывается 3 точками. Это значит, что для типов float и double требуется одновременная обработка 4 и 2 вершин соответственно для заполнения векторов. Наиболее предпочтительным для векторизации является формат $N \times 3$, далее описана векторизация вычислений с использованием данного формата. На рисунке 5.15 показано распределение координат 4 вершин в трёх регистрах XMM для типа float в представлении $N \times 3$

x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4

Рисунок 5.15 – Распределение координат 4 вершин в трёх регистрах XMM для типа float в представлении $N \times 3$

Матрица Хаусхолдера симметрична, поэтому для описания алгоритма векторизации она будет описываться следующим образом:

$$H = \begin{bmatrix} a & f & e \\ f & b & g \\ e & g & c \end{bmatrix} \quad (5.2)$$

Всего требуется вычислить 3 квартета значений, что разбивает один проход на 3 отдельных шага, которые описаны на рисунках 5.16, 5.17 и 5.18. Вектора множителей (к примеру [a, f, e, a]), образованные из матрицы H, циклично сдвигаются на каждом шаге влево так, что последним элементом становится новый первый, то есть вектор [a, f, e, a] на первом шаге станет [f, e, a, f] на втором. Аналогичный сдвиг будет выполнен и на шаге 3. Выполнение векторизации для типа double осуществляется аналогично, за исключением формирования вектора множителей.

Вычислительные эксперименты показали эффективность предложенного алгоритма векторизации вычислений. Производилось сравнение времени выполнения преобразования с использованием полученных алгоритмов и с использованием функций LAPACKE пакета Intel MKL [62].

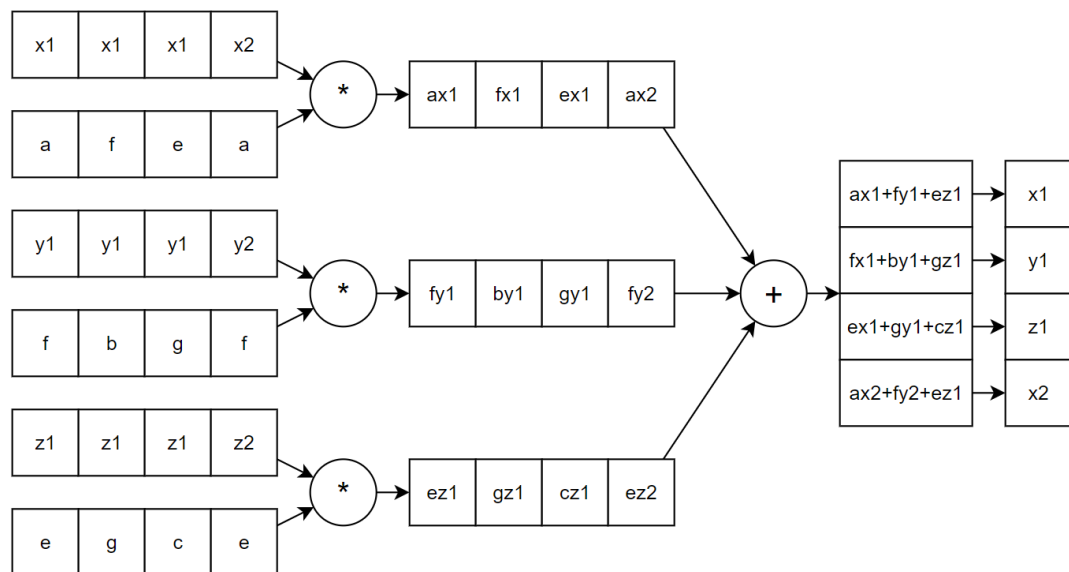


Рисунок 5.16 – Схема выполнение 1/3 части шага для векторизованной операции отражения. Тип данных – float

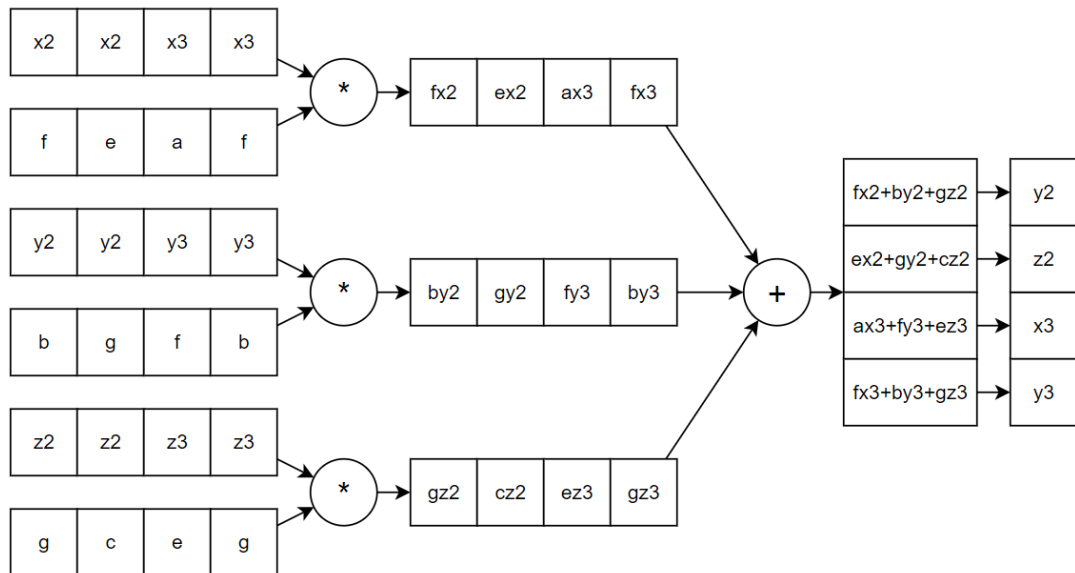


Рисунок 5.17 – Схема выполнение 2/3 части шага для векторизованной операции отражения. Тип данных – float

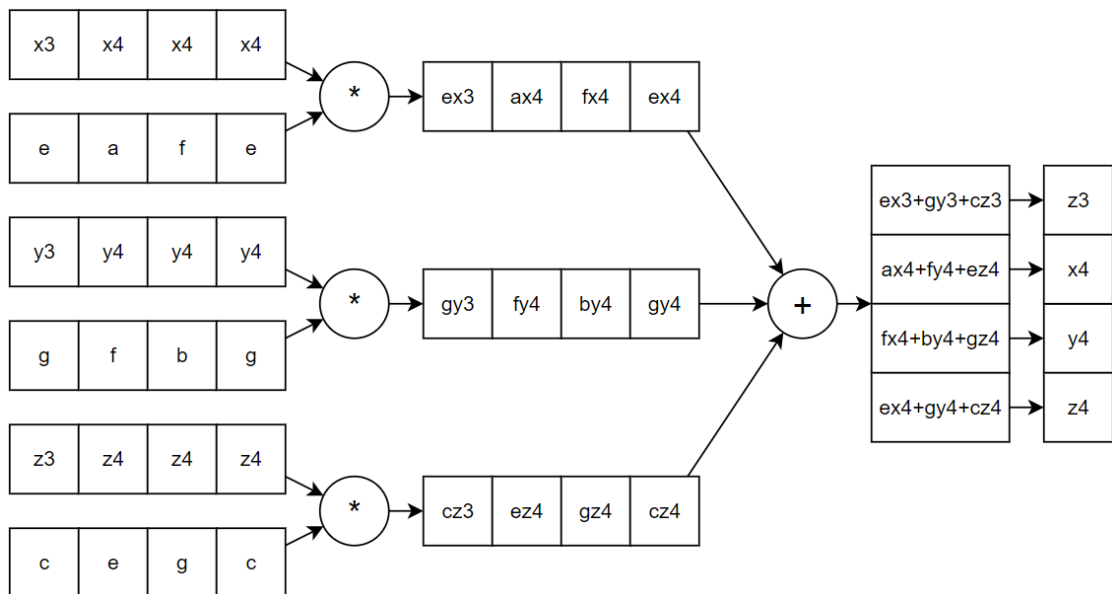


Рисунок 5.18 – Схема выполнение 3/3 части шага для векторизованной операции отражения. Тип данных – float

На рисунках 5.19 и 5.20 показана зависимость ускорения вычислений от количества полигонов относительно эталонного метода, в качестве которого использовалась матричная форма реализации отражения Хаусхолдера с форматом представления вершин $N \times 3$.

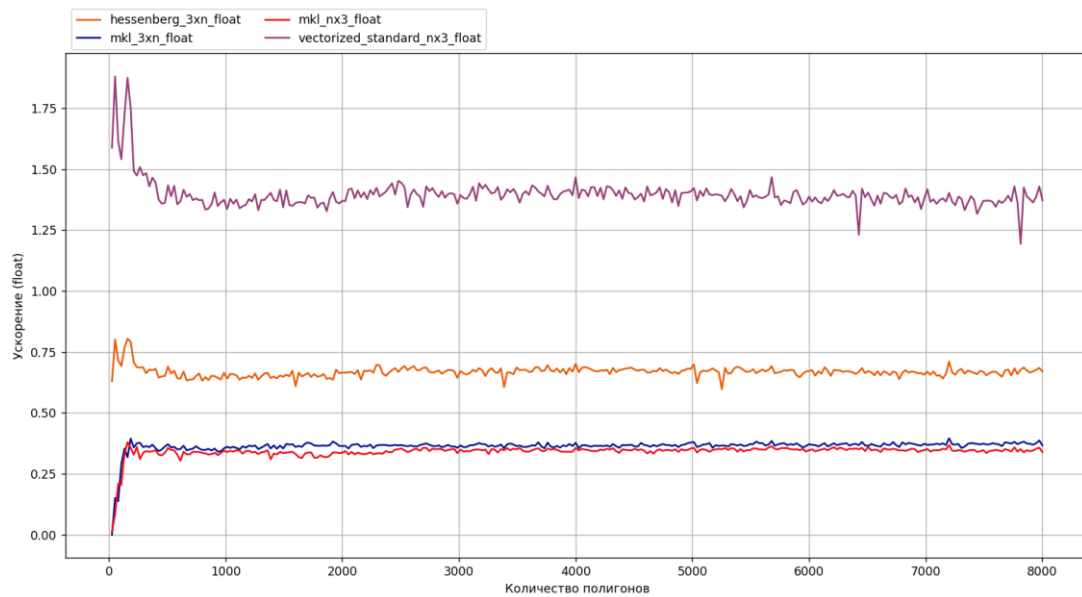


Рисунок 5.19 – Зависимость ускорения от количества полигонов. Тип float

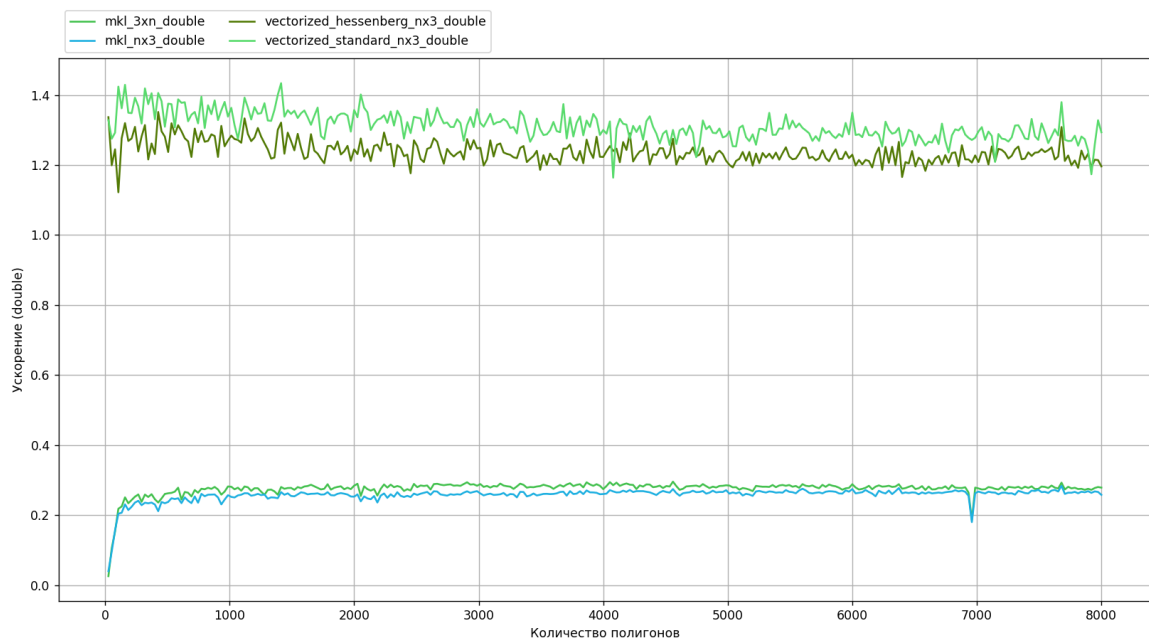


Рисунок 5.20 – Зависимость ускорения от количества полигонов. Тип double

На рисунках представлены обозначения в формате name_format_type. Здесь name – название алгоритма, format – формат хранения данных, type – тип используемых данных. Name принимает значения:

- standart – матричная форма реализации отражения Хаусхолдера;
- hessenberg – векторная форма реализации отражения Хаусхолдера;

- `vectorized_standart` – векторизованный вариант матричной формы реализации отражения Хаусхолдера;
- `vectorized_hessenberg` – векторизованный формат векторной формы реализации отражения Хаусхолдера;
- `mkl` – реализация Intel MKL.

`Format` может принимать значения $3 \times N$ и $N \times 3$, сами форматы описаны выше. `Type` может принимать значения `double` и `float`, что соответствует типам используемых данных. Например, `hessenberg_3xn_float` означает реализацию матричной формы реализации отражения Хаусхолдера, формат хранения – «широкая матрица», тип данных – `float`.

По результатам проведенных экспериментов можно заключить, что наибольшее ускорение достигается при реализации предложенной вычислительной схемы векторизации для формата представления вершин $N \times 3$. Полные результаты исследований представлены в [171]

5.4 Основные результаты и выводы по главе 5

Проведена экспериментальная апробация и внедрение научных результатов для повышения эффективности ПС.

Разработана архитектура и программная реализация СП ЖЦ, основанной на основе предложенной методологии проектирования эффективных ПС для решения инженерных задач. Система позволяет выбирать проектные решения по повышению эффективности ПС и состоит из трех основных подсистем: подсистемы формирования характеристик кода, подсистемы оценки эффективности кода, подсистемы поддержки проектирования ПС.

Система выполняет статический анализ исходного кода программы, на основе проведенного анализа:

- рассчитываются значения метрик ПС;
- анализируются используемые паттерны доступа к данным;

- выявляются необходимые проектные решения.

Описаны внедрение и апробация методологии проектирования эффективных ПС на отечественных предприятиях. Приведены полученные результаты в области повышения эффективности ПС согласно актам внедрения (Приложение Б).

ЗАКЛЮЧЕНИЕ

Главным результатом работы являются научно-методологические основы поддержки жизненного цикла проектируемых программных систем для решения инженерных задач в гетерогенных вычислительных средах, повышающие эффективность программных систем на различных этапах жизненного цикла.

Основные результаты, в которых отражена научная новизна и практическая ценность диссертационной работы, заключаются в следующем:

1. Исследованы существующие модели, методы и CASE-средства, использующиеся для проектирования ПС в области высокопроизводительных вычислений. Рассмотрены полные тексты более 40 публикаций, часть из этих публикаций представляют собой систематические исследования с общим числом исследованных работ более 600. Выявлены основные тенденции в области разработки моделей проектирования ПС, методов, основанных на этих моделях, выявлены основные недостатки данных моделей и методов.

2. Предложена модель улучшения показателей эффективности ПС, которая позволяет сократить время проектирования. Применение данной модели снижает затраты на разработку и модификацию ПС за счет значительного (в 2-3 раза) уменьшения числа итераций проектных процедур.

3. Предложена математическая модель оценки эффективности проектируемых ПС, позволяющая проектировщику ПС комплексно оценить ПС на различных этапах ЖЦ в ГВС.

4. Предложен аналитический метод определения максимального размера блока при регулярном доступе к данным, позволяющий производить исследование показателей эффективности использования подсистемы памяти проектируемых ПС. Сформулированы и доказаны 4 теоремы, формально описывающие отображение множества кешируемых данных на множества кеш-памяти. Сформулирован ряд следствий из теорем, позволяющих

аналитически оценить степень заполняемости множеств кеш-памяти с различными характеристиками при организации регулярного доступа к данным на основе различных шаблонов доступа. Реализация данного метода в процессе анализа ПС позволяет выбирать параметры алгоритмов, минимизирующие количество кеш-промахов при доступе к данным, тем самым значительно ускоряя работу ПС.

5. Предложена имитационная модель исполнения программы в ГВС с иерархической структурой памяти, позволяющая определять характеристики ПС на конкретной целевой вычислительной архитектуре, а также характеристики подсистемы памяти для проектируемой ПС. Сформулированы и доказаны четыре теоремы, устанавливающие достоверность моделирования в имитационной модели при выполнении ряда модельных предположений. Реализация модели дает проектировщику ПС возможность проведения имитационного моделирования процесса выполнения программы в подсистеме памяти, что позволяет оценивать ПС и принимать проектные решения по повышению эффективности ПС.

6. Предложен комплекс методов, позволяющий значительно повысить эффективность проектируемых ПС за счет значительного (до 4 раз) снижения времени экспериментальных вычислений и уменьшения энергопотребления ПС в 3-4 раза на этапе эксплуатации, снижения затрат на проектирование ПС:

- метод повышения эффективности использования подсистемы памяти, позволяющий улучшать стратегию кеширования данных в ГВС с иерархической структурой памяти;

- метод повышения эффективности векторизации вычислений, позволяющий избежать потерь производительности при доступе к данным за счет отказа от невыровненных обращений в память;

- метод повышения эффективности использования GPU, позволяющий повысить уровень загрузки и степень сбалансированности загрузки потоковых процессоров, что позволяет увеличить среднюю плотность

вычислений, снизить расхождение ветвей, увеличить долю использованных потоков.

7. Предложена методология проектирования эффективных ПС для решения инженерных задач в ГВС, реализующая предложенные модели и методы, позволяющая принимать проектные решения по эффективному использованию вычислительных ресурсов на начальных этапах жизненного цикла ПС. Реализация данной методологии позволяет повысить эффективность работы проектировщика, эффективность проектируемых ПС.

8. Выполнена верификация созданной методологии проектирования ЭПС на примере практически значимых вычислительно-сложных задач.

9. Предложены концепция и архитектура системы поддержки ЖЦ проектируемых ПС, позволяющей автоматизировать анализ эффективности возможных проектных решений о стратегиях разработки и модификации ПС, которая состоит из трех основных подсистем: подсистемы формирования характеристик кода, подсистемы оценки эффективности кода, подсистемы поддержки проектирования ПС.

В качестве перспектив исследования можно выделить развитие методологии проектирования ЭПС за счет анализа параллельных и распределенных вычислений, расширения перечня анализируемых вычислительных узлов в составе ГВС, а также развитие системы поддержки ЖЦ проектируемых ПС.

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

БПФ – быстрое преобразование Фурье
ВПВ – высокопроизводительные вычисления
ВС – вычислительная среда
ВУ – вычислительный узел
ГВС – гетерогенная вычислительная среда
ЖЦ – жизненный цикл
ИИ – искусственный интеллект
МВВ - метод ведущих векторов
МП – микропроцессор
МУСК - метод улучшения стратегии кеширования
ОС – операционная система
ПП – подсистема памяти
ПС - программная система
САПР – система автоматизированного проектирования
СЛАУ – система линейных алгебраических уравнений
СП ЖЦ - система поддержки жизненного цикла
ХФ – Хессенбергова форма
ЦВА – целевая вычислительная архитектура
ЭПС – эффективная программная система
ASIC - Application-Specific Integrated Circuit
AVX - Advanced Vector Extension
BLAS - Basic Linear Algebra Subprograms
BPMN – Business Process Model and Notation
CASE - Computer-Aided Software Engineering
CPU - Central Processing Unit
FPGA - Field-Programmable Gate Array
FPU - Floating Point Unit
GPGPU - General-Purpose Computing on Graphics Processing Units

GPU - Graphics Processing Unit

LAPACK - Linear Algebra Package

LRU - Least Recently Used

MMX - Multimedia Extensions

SIMD - Single Instruction, Multiple Data

SSE - Streaming SIMD Extensions

СПИСОК ЛИТЕРАТУРЫ

1. Al-Mouhamed M. A., Khan A. H., Mohammad N. A review of CUDA optimization techniques and tools for structured grid computing //Computing. – 2020. – V. 102. – №. 4. – P. 977-1003.
2. Agarwal, R. C., Gustavson, F. G., McComb, J., & Schmidt, S. Engineering and scientific subroutine library release 3 for IBM ES/3090 vector multiprocessors //IBM Systems Journal. – 1989. – V. 28. – №. 2. – P. 345-350.
3. Amdahl G. M. Validity of the single processor approach to achieving large scale computing capabilities //Proceedings of the April 18-20, 1967, spring joint computer conference. – 1967. – C. 483-485.
4. Anderson E. et al. LAPACK users' guide. – Society for industrial and applied mathematics, 1999.
5. Anderson D. et al. SETI@ home: internet distributed computing for SETI //Bioastronomy 99. – 2000. – T. 213.
6. Andreev, A., Andreeva, M., Drobotov, A., ...Shmeleva, A., Denisov, M. Development of a Discrete Slicer for Additive Manufacturing. Communications in Computer and Information Science, 2021, 1448 CCIS, pg. 267–281.
7. Arroyo M., Chiotta F., Bavera F. An user configurable clang static analyzer taint checker //2016 35th International Conference of the Chilean Computer Science Society (SCCC). – IEEE, 2016. – P. 1-12.
8. Babati B., Horváth G., Májer V., Pataki N.. Static analysis toolset with Clang //Proceedings of the 10th International Conference on Applied Informatics. – 2017. – P. 23-29.
9. Ballard, G., Carson, E., Demmel, J., Hoemmen, M., Knight, N., Schwartz, O. Communication lower bounds and optimal algorithms for numerical linear algebra //Acta Numerica. – 2014. – V. 23. – P. 1-155.
10. Barr, E.T., Marron, M.: Tardis: affordable time-travel debugging in managed runtimes. In: Proceedings of the 2014 ACM International Conference on

Object Oriented Programming Systems Languages and Applications (OOPSLA'14), pp. 67–82. ACM (2014)

11. Bán, D., Ferenc, R., Siket, I. et al. Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware. *J Supercomput* 75, 4001–4025 (2019). <https://doi.org/10.1007/s11227-018-2252-6>

12. Belter, G., Jessup, E. R., Karlin, I., Siek, J. G. Automating the generation of composed linear algebra kernels // *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. – 2009. – P. 1-12.

13. Bilmes J., Asanovic K., Chin C., Demmel J. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology // *The 11th International Conference on Supercomputing* (Vienna, Austria, July 7—11, 1997). New York, NY, USA, ACM, 1997. P.340—347.

14. Blackford L. S. et al. An updated set of basic linear algebra subprograms (BLAS) // *ACM Transactions on Mathematical Software*. – 2002. – T. 28. – №. 2. – C. 135-151.

15. Blair-Chappell S., Stokes A. *Parallel programming with Intel Parallel Studio XE*. – John Wiley & Sons, 2012.

16. Brais H., Kalayappan R., Panda P. R. A survey of cache simulators // *ACM Computing Surveys (CSUR)*. – 2020. – V. 53. – №. 1. – P. 1-32.

17. Bruce Jacob, Spencer W. Ng, David T. Wang, CHAPTER 1 - An Overview of Cache Principles, Editor(s): Bruce Jacob, Spencer W. Ng, David T. Wang, *Memory Systems*, Morgan Kaufmann, 2008, Pages 57-77, ISBN 9780123797513.

18. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., DeAntoni, J., Combemale, B.: Execution framework of the GEMOC studio (tool demo). In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE'16)*, pp. 84–89. ACM (2016)

19. Butko, A., Garibotti, R., Ost, L., Lapotre, V., Gamatie, A., Sassatelli, G., Adeniyi-Jones, C. A trace-driven approach for fast and accurate simulation of

manycore architectures //The 20th Asia and South Pacific Design Automation Conference. – IEEE, 2015. – P. 707-712.

20. Chambers B., Zaharia M. Spark: The definitive guide: Big data processing made simple. – " O'Reilly Media, Inc.", 2018.

21. Chen Y., Mendis C., Amarasinghe S. All you need is superword-level parallelism: systematic control-flow vectorization with SLP //Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. – 2022. – P. 301-315.

22. Chen, Y., Mendis, C., Carbin, M., Amarasinghe, S VeGen: a vectorizer generator for SIMD and beyond //Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. – 2021. – C. 902-914.

23. Chernykh, I., Kulikov, I., Glinsky, B., Vshivkov, V., Vshivkova, L., Prigarin, V. Advanced vectorization of PPML method for Intel® Xeon® scalable processors //Supercomputing: 4th Russian Supercomputing Days, RuSCDays 2018, Moscow, Russia, September 24–25, 2018, Revised Selected Papers 4. – Springer International Publishing, 2019. – P. 465-471.

24. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of UML models: a systematic review of research and practice. *Softw. Syst. Model.* 1, 1–48 (2018)

25. Combemale, B., Crégut, X., Pantel, M.: A design pattern to build executable DSMLs and associated V&V tools. In: *Proceedings of the 19th Asia–Pacific on Software Engineering Conference (APSEC)*, vol. 1, pp. 282–287. IEEE (2012)

26. Demmel J. Communication avoiding algorithms //2012 SC Companion: High Performance Computing, Networking Storage and Analysis. – IEEE, 2012. – P. 1942-2000.

27. Diefendorff K. et al. AltiVec extension to PowerPC accelerates media processing //IEEE Micro. – 2000. – V. 20. – №. 2. – P. 85-95.

28. Dongarra, J. J., Du Croz, J., Hammarling, S., & Duff, I. S. A set of level 3 basic linear algebra subprograms //ACM Transactions on Mathematical Software (TOMS). – 1990. – V. 16. – №. 1. – P. 1-17.
29. Dongarra, J. J., Du Croz, J., Hammarling, S., Hanson, R. J. An extended set of fortran basic linear algebra subprograms: Model implementation and test programs. – Argonne National Lab., IL (USA), 1987. – №. ANL/MCS-TM-81.
30. Dragos P. Overview of the Agile Rational Unified Process (Rup) in the Context of Software Development Projects //Journal of Business and Economics. – 2021. – T. 12. – №. 6. – C. 681-684.
31. Eeckhout L. Computer architecture performance evaluation methods. – Morgan & Claypool Publishers, 2010.
32. Egunov V.A., Andreev A.E. Implementation of QR and LQ decompositions on shared memory parallel computing systems // 2016 2nd International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) (Chelyabinsk, Russia, 19-20 May 2016). – [Publisher: IEEE], 2016. – 5 p. – DOI: 10.1109/ICIEAM.2016.7911607.
33. Egunov V. A., Kravets A. G. The New Method for Automatic Vectorization Efficiency Increasing //Cyber-Physical Systems: Data Science, Modelling and Software Optimization. – Cham : Springer Nature Switzerland, 2024. – P. 195-208.
34. Eichenberger A. E., Wu P., O'brien K. Vectorization for SIMD architectures with alignment constraints //Acm sigplan notices. – 2004. – V. 39. – №. 6. – P. 82-93.
35. Eljammaly, M., Karlsson, L., Kagstrom, B. On the tunability of a new Hessenberg reduction algorithm using parallel cache assignment, 2018. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10777 LNCS, c. 579-589.
36. Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Jeffrey C. Carver, Software engineering practices for scientific

software development: A systematic mapping study, *Journal of Systems and Software*, Volume 172, 2021, 110848, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2020.110848>

37. Eze, Val & Eze, Martin & Edozie, Enerst & Eze, Esther. (2023). Design and Development of Effective Multi-Level Cache Memory Model. *International Journal of Recent Technology and Applied Science (IJORTAS)*. 5. 54-64. DOI:10.36079/lamintang.ijortas-0502.515.

38. Filimonov, O.Y., Egunov, V.A., Nesterenko, E.N. Constructing Equidistant Curve for Planar Composite Curve in CAD Systems. *Communications in Computer and Information Science*, 2021, 1448 CCIS, pg. 296–309

39. Forlin B. et al. Lightweight Instrumentation for Accurate Performance Monitoring in RTOSes //2024 Design, Automation & Test in Europe Conference & Exhibition (DATE). – IEEE, 2024. – P. 1-2.

40. Frigo M. et al. Cache-oblivious algorithms //40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039). – IEEE, 1999. – P. 285-297.

41. Frigo M., Johnson S. G. FFTW: An adaptive software architecture for the FFT //Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181). – IEEE, 1998. – V. 3. – P. 1381-1384.

42. Frigo M., Johnson S. G. The design and implementation of FFTW3 //Proceedings of the IEEE. – 2005. – V. 93. – №. 2. – P. 216-231.

43. Garzón, Esteban & Hanhan, Robert & Lanuzza, Marco & Teman, Adam & Yavits, Leonid. (2024). FASTA: Revisiting Fully Associative Memories in Computer Microarchitecture. *IEEE Access*. PP. 10.1109/ACCESS.2024.3355961.

44. Goto K., Geijn R. A. Anatomy of high-performance matrix multiplication //ACM Transactions on Mathematical Software (TOMS). – 2008. – V. 34. – №. 3. – P. 1-25.

45. Goto K., Van De Geijn R. High-performance implementation of the level-3 BLAS //ACM Transactions on Mathematical Software (TOMS). – 2008. – V. 35. – №. 1. – P. 1-14.
46. Guzman, V., Pitkänen, T. & Takala, J. Use of compiler optimization of software bypassing as a method to improve energy efficiency of exposed data path architectures. J Embedded Systems 2013, 9 (2013). <https://doi.org/10.1186/1687-3963-2013-9>
47. Guocong Quan, Atilla Eryilmaz, Jian Tan, Ness Shroff, Prefetching and caching for minimizing service costs: Optimal and approximation strategies, Performance Evaluation, Volume 145, 2021, 102149, ISSN 0166-5316,
48. Gurung G., Shah R., Jaiswal D. P. Software development life cycle models-A comparative study //International Journal of Scientific Research in Computer Science, Engineering and Information Technology. – 2020. – T. 6. – №. 4. – C. 30-37.
49. Goto K., Van De Geijn R. High-Performance Implementation of the Level-3 BLAS // ACM Transactions on Mathematical Software. 2008. Vol. 35, no. 1. P. 4:1—4:14. DOI: 10.1145/1377603.1377607
50. Goto K., Geijn R.A. Anatomy of High-Performance Matrix Multiplication // ACM Transactions on Mathematical Software. 2008. Vol. 34, 158 no. 3. P. 12:1—12:25. DOI: 10.1145/1356052.1356053.
51. Johnson, J., Johnson, R. W., Padua, D. A., & Xiong, J.. Searching for the best FFT formulas with the SPL compiler //International Workshop on Languages and Compilers for Parallel Computing. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2000. – P. 112-126.
52. Hager, G., Treibig, J., Habich, J., Wellein, G. Exploring performance and power properties of modern multi-core chips via simple machine models //Concurrency and computation: practice and experience. – 2016. – T. 28. – №. 2. – P. 189-210.
53. Harris D., Harris S. Digital Design and Computer Architecture // 2nd Edition. NY, Morgan Kaufmann, 2012. 712 p.

54. Heaton D. , Carver J. C. "Claims about the use of software engineering practices in science: A systematic literature review", *Information and Software Technology*, 67, pp. 207-219, 2015
55. Hegedus, A., Bergmann, G., Ráth, I., Varró, D.: Replaying execution trace models for dynamic modeling languages. *Period. Polytechn. Electr. Eng. Comput. Sci.* 56(3), 71–82 (2013)
56. Hilken, F., Gogolla, M.: Verifying linear temporal logic properties in UML/OCL class diagrams using filmstripping. In: *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pp. 708–713. IEEE (2016)
57. Hockney R. W. and Curington I. J., “ $f1/2$: A parameter to characterize memory and communication bottlenecks,” *Parallel Computing*, vol. 10, no. 3, pp. 277–286, 1989
58. Hofgaard, Jacob. Static control flow graph analysis for inlined function detection and identification. MS thesis. NTNU, 2024.
59. Hojaji, F., Mayerhofer, T., Zamani, B. et al. Model execution tracing: a systematic mapping study. *Softw Syst Model* 18, 3461–3485 (2019). <https://doi.org/10.1007/s10270-019-00724-1>
60. H. van Vliet, “Software Engineering: Principles and Practice”, John Wiley, 2008.
61. Intel® C++ Compiler Classic Developer Guide and Reference. https://www.physics.udel.edu/~bnikolic/QTG/shared/docs/intel_c_user_and_reference_guide.pdf. (дата обращения 09.04.25)
62. Intel Math Kernel Library for Windows OS. Users’s Guide.. https://jp.xlsoft.com/documents/intel/mkl/11.0/mkl_userguide_win.pdf. (дата обращения 09.04.25)
63. Intel MPI Library Developer Guide for Windows OS. <https://jp.xlsoft.com/documents/intel/mpi/2021/mpi-devguide-windows.pdf>. (дата обращения 09.04.25) .

64. Intel® Threading Building Blocks.
<https://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf/> (дата обращения 09.04.25)
65. Intel® VTune™ Amplifier Installation Guide
https://jp.xlsoft.com/documents/intel/vtune/2018/amplxe_install_guide_windows_2018_Beta.pdf (дата обращения 09.04.25)
66. Jaleel, A., Cohn, R. S., Luk, C. K., Jacob, B. CMP \$ im: A Pin-based on-the-fly multi-core cache simulator //Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA. – 2008. – P. 28-36.
67. Jeffers J., Reinders J. Intel Xeon Phi coprocessor high performance programming. – Newnes, 2013.
68. Jouppi, Norman. (1998). Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache Prefetch Buffers.. Conference Proceedings - Annual Symposium on Computer Architecture. 18. 388-397. 10.1109/ISCA.1990.134547
69. Ivannikov, V.P., Grushin, D.A., Kuzyurin, N.N. et al. Software for improving the energy efficiency of a computer cluster. Program Comput Soft 36, 327–336 (2010). <https://doi.org/10.1134/S0361768810060022>
70. Kabir, K., Haidar, A., Tomov, S., & Dongarra, J. Performance analysis and design of a hessenberg reduction using stabilized blocked elementary transformations for new architectures // Simulation Series, 2015, 47(4) pp.135-142
71. Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou Rapl in action: Experiences in using rapl for power measurements //ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS). – 2018. – v. 3. – №. 2. – p. 1-26
72. Katal, A., Dahiya, S. & Choudhury, T. Energy efficiency in cloud computing data centers: a survey on software technologies. Cluster Comput (2022). <https://doi.org/10.1007/s10586-022-03713-0>

73. Kelefouras V., Kritikakou A., Goutis C. A Matrix–Matrix Multiplication methodology for single/multi-core architectures using SIMD //The Journal of Supercomputing. – 2014. – V. 68. – P. 1418-1440.
74. Kiselev, E.A., Kiselev, V.I., Shabanov, B.M. et al. The Energy Efficiency Evaluating Method Determining Energy Consumption of the Parallel Program According to Its Profile. Lobachevskii J Math 41, 2542–2551 (2020). <https://doi.org/10.1134/S1995080220120161>
75. Knijnenburg P.M.W., Kisuki T., O’Boyle M.F.P. Iterative Compilation // Lecture Notes in Computer Science. 2002. Vol. 2268. P.171—187.
76. Kozin I. N. Evaluation of clearspeed accelerators for hpc. – Science & Technology Facilities Council, 2009.
77. Kowarschik M., Weiß C. An overview of cache optimization techniques and cache-aware numerical algorithms //Algorithms for memory hierarchies: advanced lectures. – 2003. – P. 213-232.
78. Kravets, A.G., Egunov, V. The Software Cache Optimization-Based Method for Decreasing Energy Consumption of Computational Clusters. Energies 2022, 15, 7509. <https://doi.org/10.3390/en15207509>
79. Kudela P., Fiborek P. Vectorization of the code for guided wave propagation problems //European Workshop on Structural Health Monitoring: Special Collection of 2020 Papers-Volume 1. – Springer International Publishing, 2021. – P. 707-715.
80. Kusswurm, D. (2014). Advanced Vector Extensions (AVX). In: Modern X86 Assembly Language Programming. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-0064-3_12
81. Kusswurm, D. (2022). AVX-512 C++ Programming: Part 1. In: Modern Parallel Programming with C++ and Assembly Language. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-7918-2_7
82. Kusswurm, D. (2022). AVX-512 C++ Programming: Part 2. In: Modern Parallel Programming with C++ and Assembly Language. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-7918-2_8

83. Lattner C. A. LLVM: An infrastructure for multi-stage optimization. – 2002.
84. Larsen S., Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets //Acm Sigplan Notices. – 2000. – V. 35. – №. 5. – P. 145-156.
85. Lawson, C. L., Hanson, R. J., Kincaid, D. R., & Krogh, F. T. Basic linear algebra subprograms for Fortran usage //ACM Transactions on Mathematical Software (TOMS). – 1979. – V. 5. – №. 3. – P. 308-323.
86. Lee J., Kim H., Vuduc R. When prefetching works, when it doesn't, and why //ACM Transactions on Architecture and Code Optimization (TACO). – 2012. – T. 9. – №. 1. – P. 1-29.
87. Low, T. M., Igual, F. D., Smith, T. M., & Quintana-Orti, E. S. Analytical modeling is enough for high-performance BLIS //ACM Transactions on Mathematical Software (TOMS). – 2016. – T. 43. – №. 2. – C. 1-18.
88. Machupalli R., Hossain M., Mandal M. Review of ASIC accelerators for deep neural network //Microprocessors and Microsystems. – 2022. – T. 89. – C. 104441.
89. Majeed, A., Shah, M.A. Energy efficiency in big data complex systems: a comprehensive survey of modern energy saving techniques. Complex Adapt Syst Model 3, 6 (2015). <https://doi.org/10.1186/s40294-015-0012-5>
90. Maleki S. et al. An evaluation of vectorizing compilers //2011 International Conference on Parallel Architectures and Compilation Techniques. – IEEE, 2011. – C. 372-382.
91. Manual O. Intel Architecture Optimization Manual. – 1996.
92. Martínez-Davies, D., Veigas-Ramírez, S., Gallego-Torres, L., & Garcia, J. D. Code Rejuvenation: From Vector Compiler Intrinsics to Portable Standardized SIMD //International Journal of Parallel Programming. – 2025. – V. 53. – №. 3. – C. 20.
93. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: executable DSMLs based on fUML. In: Proceedings of the International

Conference on Software Language Engineering, Volume 8225 of Lecture Notes in Computer Science, pp. 56–75. Springer, Berlin (2013)

94. Merchant, F., Vawani, T., Chattopadhyay, A., (...), Nandy, S.K., Narayan, R., Efficient realization of householder transform through algorithm-architecture co-design for acceleration of QR Factorization, 2018, IEEE Transactions on Parallel and Distributed Systems, 29(8), p. 1707-1720

95. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: a framework for generating domain-specific property languages. In: Proceedings of the International Conference on Software Language Engineering (SLE), Volume 8706 of Lecture Notes in Computer Science, pp. 1–20. Springer, Berlin (2014)

96. Microsoft MPI. <https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi>. (дата обращения 09.04.25)

97. Nuzman D., Rosen I., Zaks A. Auto-vectorization of interleaved data for SIMD //ACM SIGPLAN Notices. – 2006. – V. 41. – №. 6. – P. 132-143.

98. OpenBLAS. User Manual. https://www.openmathlib.org/OpenBLAS/docs/user_manual. (дата обращения 09.04.25)

99. Porpodas V., Magni A., Jones T. M. PSLP: Padded SLP automatic vectorization //2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). – IEEE, 2015. – P. 190-201.

100. Profiling tools for hardware/software partitioning of embedded applications // Suresh, D. C., Najjar, W. A., Vahid, F., Villarreal, J. R., & Stitt, G. //Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems. – 2003. – P. 189-198.

101. Puschel M. et al. SPIRAL: Code generation for DSP transforms //Proceedings of the IEEE. – 2005. – V. 93. – №. 2. – P. 232-275.

102. Rahman S. M. F., Yi Q., Qasem A. Understanding stencil code performance on multicore architectures //Proceedings of the 8th ACM International Conference on Computing Frontiers. – 2011. – P. 1-10.

103. Ren G., Wu P., Padua D. A preliminary study on the vectorization of multimedia applications for multimedia extensions //Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16. – Springer Berlin Heidelberg, 2004. – P. 420-435.
104. Runday, F. (1986). Vector Computers and Vector Programming. In: O'Brien, J.J. (eds) Advanced Physical Oceanographic Numerical Modelling. NATO ASI Series, vol 186. Springer, Dordrecht. https://doi.org/10.1007/978-94-017-0627-8_17
105. Rotem-Gal-Oz A. Fallacies of distributed computing explained //URL <http://www.rgoarchitects.com/Files/fallacies.pdf>. – 2006. – T. 20.
106. Sanjuan, G., Tena, C., Margalef, T., & Cortés, A. Applying vectorization of diagonal sparse matrix to accelerate wind field calculation //The Journal of Supercomputing. – 2017. – V. 73. – №. 1. – P. 240-258.
107. Saravanos A., Curinga M. X. Simulating the Software Development Lifecycle: The Waterfall Model //Applied System Innovation. – 2023. – T. 6. – №. 6. – C. 108.
108. Schematic of the CUDA programming model (URL: https://www.researchgate.net/figure/Schematic-of-the-CUDA-programming-model_fig3_321958738)
109. Senarath U. S. Waterfall methodology, prototyping and agile development //Tech. Rep. – 2021. – C. 1-16.
110. Stallman R. M. Using the gnu compiler collection //Free Software Foundation. – 2003. – V. 4. – №. 02.
111. Shoch John, Hupp Jon, The 'Worm' Programs - Early Experience with a Distributed Computation // Communications of the ACM, Volume 25, Number 3, March 1982, pp. 172–180.
112. Schreiber R, VanLoan C. A Storage-Efficient WY Representation for Products of Householder Transformations // SIAM Journal on Scientific and Statistical Computing 10(1) · February 1989. Pp – 53 – 57.

113. Smith, T. M., Van De Geijn, R., Smelyanskiy, M., Hammond, J. R., Van Zee, F. G. Anatomy of high-performance many-threaded matrix multiplication //2014 IEEE 28th International Parallel and Distributed Processing Symposium. – IEEE, 2014. – P. 1049-1059.
114. Spampinato D. G., Püschel M. A basic linear algebra compiler //Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. – 2014. – P. 23-32.
115. SSE. (2011). In: Padua, D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-09766-4_2055
116. Stengel, H., Treibig, J., Hager, G., Wellein, G. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model //Proceedings of the 29th ACM on International Conference on Supercomputing. – 2015. – PP. 207-216.
117. Stroud I., Xirouchakis P.C. STL AND EXTENSIONS. Advances in Engineering Software. 2000. T. 31. № 2. C. 83-95.
118. Treibig, J., Hager, G., Hofmann, H. G., Hornegger, J., & Wellein, G. Pushing the limits for medical image reconstruction on recent standard multicore processors //The International journal of high performance computing applications. – 2013. – T. 27. – №. 2. – P. 162-177.
119. Treibig J., Hager G. Introducing a performance model for bandwidth-limited loop kernels //International Conference on Parallel Processing and Applied Mathematics. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. – C. 615-624.
120. Thesing T., Feldmann C., Burchardt M. Agile versus waterfall project management: decision model for selecting the appropriate approach to a project //Procedia Computer Science. – 2021. – T. 181. – C. 746-756.
121. Tian Y. et al. On the Caching Schemes to Speed Up Program Reduction //ACM Transactions on Software Engineering and Methodology. – 2023. – T. 33. – №. 1. – C. 1-30.

122. Tomas Dominguez, A.E., Quintana Orti, E.S., Fast Blocking of Householder Reflectors on Graphics Processors, 2018, Proceedings - 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2018, p. 385-393
123. Tsymbal, V., Kurylev, A., Profiling Heterogeneous Computing Performance with VTune Profiler // (2021) WOCL'21: International Workshop on OpenCL April 2021 Article No.: 10 <https://doi.org/10.1145/3456669.3456678>
124. Udoka E. V. H. et al. Design and development of effective multi-level cache memory model //International Journal of Recent Technology and Applied Science (IJORTAS). – 2023. – T. 5. – №. 2. – C. 54-64.
125. Von Hagen W. The definitive guide to GCC. – Apress, 2011.
126. Wang, Q., Zhang, X., Zhang, Y., Yi, Q AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs //Proceedings of the international conference on high performance computing, networking, storage and analysis. – 2013. – P. 1-12.
127. Wanko J. J., Venable C. H. Investigating Prime Numbers and the Great Internet Mersenne Prime Search //Mathematics Teaching in the Middle School. – 2002. – V. 8. – №. 2. – P. 70-76.
128. Whaley R. C., Dongarra J. J. Automatically tuned linear algebra software //SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing. – IEEE, 1998. – P. 38-38.
129. Whaley R. C., Petitet A., Dongarra J. J. Automated empirical optimizations of software and the ATLAS project //Parallel computing. – 2001. – T. 27. – №. 1-2. – P. 3-35.
130. White S. A., Miers D. BPMN modeling and reference guide: understanding and using BPMN. – Future Strategies Inc., 2008.
131. White T. Hadoop: The definitive guide. – " O'Reilly Media, Inc.", 2012.

132. S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009
133. Wilson G., "Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive", *Computing in Science & Engineering*, 8 (6), pp. 66-69, Nov.-Dec. 2006.
134. Wu, HT., Cho, HH., Wang, SJ. et al. Intelligent data cache based on content popularity and user location for Content Centric Networks. *Hum. Cent. Comput. Inf. Sci.* 9, 44 (2019). <https://doi.org/10.1186/s13673-019-0206-5>.
135. Wu P., Eichenberger A. E., Wang A. Efficient SIMD code generation for runtime alignment and length conversion // *International Symposium on Code Generation and Optimization*. – IEEE, 2005. – P. 153-164.
136. Yotov K. et al. An experimental comparison of cache-oblivious and cache-conscious programs // *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. – 2007. – P. 93-104.
137. Yotov, K., Li, X., Ren, G., Garzaran, M. J. S., Padua, D., Pingali, K., Stodghill, P. Is search really necessary to generate high-performance BLAS? // *Proceedings of the IEEE*. – 2005. – T. 93. – №. 2. – P. 358-386.
138. Zhang Y. et al. Models of Parallel Computation: a Survey and Classification // *Frontiers of Computer Science in China*. Higher Education Press, 2007. Vol. 1, No. 2. P. 156–165. DOI: 10.1007/s11704-007-0016-1
139. Абдрахманова Г. И., Васильковский С. А., Вишневский, М. А. Гершман, Л. М. Гохберг и др. Цифровая трансформация: ожидания и реальность: докл. к XXIII Ясинской (Апрельской) междунар. науч. конф. по проблемам развития экономики и общества, Москва, 2022 г. [Текст] / рук. авт. кол. П. Б. Рудник; Нац. исслед. ун-т «Высшая школа экономики». — М.: Изд. дом Высшей школы экономики, 2022. — 221 с. — ISBN 978-5-7598-2658-3 (в обл.). — ISBN 978-5-7598-2468-8 (e-book).
140. Абрамов Н. С., Абрамов С. М. Ноябрь 2022: состояние и перспективы развития суперкомпьютерной отрасли в мире и в России

//Программные системы: теория и приложения. – 2023. – Т. 14. – №. 2 (57). – С. 49-93.

141. Айдинян А. Р. Аппаратные средства вычислительной техники. – Directmedia, 2016.

142. Аль-згуль Мосаб Басам Гибридные алгоритмы в системах кэширования объектов // Advanced Engineering Research (Rostov-on-Don). 2008. №4-39. URL: <https://cyberleninka.ru/article/n/gibridnye-algoritmy-v-sistemah-keshirovaniya-obektov>.

143. Анализатор Clang Static Analyzer. <http://clang-analyzer.llvm.org>.

144. Анализатор CppCheck. <http://cppcheck.sourceforge.net>

145. Андреев А.Е., Егунов В.А., Жариков Д.Н., Малолетков В.А. Реализация вычислительно-интенсивных алгоритмов на гибридных системах с реконфигурируемыми сопроцессорами: монография / ВолгГТУ. - Волгоград, 2013. - 179 с.

146. Антонов А.А., Ключев А.О., Комар М.С., Кустарев П.В., Кучерявый Е.А., Молчанов Д.А., Петров В.И., Платунов А.Е. Разработка протокола множественного доступа для процессоров с многоуровневым кэшированием // Научно-технический вестник информационных технологий, механики и оптики. 2015. №3. URL: <https://cyberleninka.ru/article/n/razrabotka-protokola-mnozhestvennogo-dostupa-dlya-protssessorov-s-mnogourovnevym-keshirovaniem>.

147. Белеванцев А.А. Многоуровневый статический анализ исходного кода для обеспечения качества программ // диссертация на соискание ученой степени доктора физико-математических наук / Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук. 2018

148. Бетелин В. Б. Отечественные суперкомпьютерные технологии экзафлопсного класса–необходимое условие обеспечения технологической

конкурентоспособности России в XXI веке // Программные продукты и системы. – 2013. – №. 4. – С. 4-9.

149. Бурцев А. А. О возможности оптимизации некоторых функций библиотеки линейной алгебры с помощью векторного сопроцессора // Труды научно-исследовательского института системных исследований Российской академии наук. – 2014. – Т. 4. – №. 2. – С. 5-15.

150. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб: БХВ-Петербург, 2002. - 608 с.

151. Воронов Г.И., Трущин В.Д., Шумилин В.В., Ежов Д.В. Программный комплекс SMPI для обеспечения разработки, оптимизации и выполнения высокопараллельных приложений на суперкомпьютерных кластерных системах // Вопросы атомной науки и техники, сер. "Математическое моделирование физических процессов", 2013, №3, с.55-60.

152. Голуб Дж., Ван Лоан Ч., Матричные вычисления. – М.: Мир, 1999.

153. Гонахчян В.И. Обзор методов упрощения полигональных моделей на графическом процессоре. Труды Института системного программирования РАН. 2014. Т. 26. № 2. С. 159-174.

154. ГОСТ 23501.108-85. Системы автоматизированного проектирования. Классификация и обозначения.

155. ГОСТ 28806-90. Качество программных средств. Термины и определения.

156. ГОСТ Р ИСО 9000-2001. Системы менеджмента качества. Основные положения и словарь.

157. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств

158. ГОСТ Р ИСО/МЭК 14764-2002. Информационная технология. Сопровождение программных средств

159. ГОСТ Р ИСО/МЭК 25021:2014 «Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Элементы показателя<качества>».

160. ГОСТ Р ИСО/МЭК ТО 15271-2002. Информационная технология. Руководство по применению ГОСТ Р ИСО/МЭК 12207 (Процессы жизненного цикла программных средств)

161. Дешко И.П., Цветков В.Я. Оценка сложности алгоритма. Славянский форум. 2021. № 3 (33). С. 38-49.

162. Егунов В. А., Андреев А. Е. Векторизация алгоритмов выполнения собственного и сингулярного разложений матриц с использованием преобразования Хаусхолдера // Прикаспийский журнал: управление и высокие технологии. 2020. № 2 (50). С. 71–85.

163. Егунов В.А. Аппаратные методы решения задач линейной алгебры: монография / В.А. Егунов, В.С. Лукьянов // ВолгГТУ. - Волгоград, 2007. - 152 с.

164. Егунов, В.А., Векторизация алгоритмов выполнения собственного и сингулярного разложений матриц с использованием преобразования Хаусхолдера / В.А. Егунов, А.Е. Андреев // Прикаспийский журнал: управление и высокие технологии. - 2020. - № 2 (50). - С. 71-85.

165. Егунов В. А., Кравец А. Г. Метод улучшения стратегии кеширования для вычислительных систем с общей памятью // Программная инженерия. 2023. Том 14, № 7. С. 329—338. DOI: 10.17587/prin.14.329-338

166. Егунов, В.А. Кэш-оптимизация процесса вычисления собственных значений на параллельных вычислительных системах / В.А. Егунов // Прикаспийский журнал: управление и высокие технологии. - 2019. - № 1 (45). - С. 154-163.

167. Егунов, В.А. Новый метод повышения эффективности векторизации операций BLAS / В.А. Егунов, А.Г. Кравец // Информационные технологии. - 2024. - Т. 30, № 6. - С. 318-328. - DOI: 10.17587/it.30.318-328

168. Егунов В.А. О влиянии кэш-памяти на эффективность программной реализации базовых операций линейной алгебры / В.А. Егунов // Прикаспийский журнал: управление и высокие технологии. - 2018. - № 3. - С. 88-96.

169. Егунов В.А. Оценка эффективности программной реализации QR-разложения на многоядерных архитектурах / В.А. Егунов, С.И. Кирносенко, П.Д. Кравченя, О.О. Шумейко // Известия ВолгГТУ. Сер. Актуальные проблемы управления, вычислительной техники и информатики в технических системах. - Волгоград, 2017. - № 1 (196). - С. 56-59

170. Егунов, В.А. Технологии кеширования данных в современных микропроцессорах / В.А. Егунов, В.А. Шабаловский // Вестник Дагестанского государственного технического университета. Технические науки. - 2024. - Т. 51, № 3. - С. 60-71. - DOI: <https://doi.org/10.21822/2073-6185-2024-51-3-60-71>

171. Егунов В.А., Филимонов О.Ю. Реализация геометрического преобразования отражения для прототипа высокопроизводительного геометрического ядра / Цифровая экономика. - 2023. - № 1 (22). - С. 20-26. - DOI: 10.34706/DE-2023-01-03.

172. Ежова Н. А., Соколинский Л. Б. Обзор моделей параллельных вычислений //Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2019. – Т. 8. – №. 3. – С. 58-91.

173. Заборовский В. С., Мулюха В. А., Уткин Л. В. Интеллектуальная система диспетчерского управления гибридными высокопроизводительным вычислительными кластерами //XVI Всероссийская мультikonференция по проблемам управления (МКПУ-2023). – 2023. – С. 16-18.

174. ИСО/МЭК ТО 14471—99. Информационная технология. Программная инженерия. Руководство по адаптации CASE-средств

175. Каляев А. И. Теоретические основы создания самоорганизующихся диспетчеров распределенных систем на базе

мультиагентного социоинспирированного подхода //Известия Южного федерального университета. Технические науки. – 2021. – №. 4 (221). – С. 6-21.

176. Каляев И.А., Левин И.И. Реконфигурируемые вычислительные системы на основе ПЛИС. Ростов-на-Дону, 2022.

177. Киреев С.Е. Программирование процессора Cell. URL: https://mypresentation.ru/presentation/programmirovanie_processora_Cell_kireev_se (дата обращения 21.08.2025)

178. Компилятор Clang. <http://clang.llvm.org>

179. Кошпаев А. А., Васяева Е. С. Анализ алгоритмов замещения данных в кэш-памяти //ББК 60 П27. – 2016. – С. 70.

180. Левин И.И., Дордопуло А.И., Каляев И.А., Гудков В.А. Высокопроизводительные реконфигурируемые вычислительные системы на основе ПЛИС VIRTEX-7 // Программная инженерия. 2014. № 6. С. 3-7.

181. Мальковский С.И., Пересветов В.В. Исследование производительности четырехпроцессорных узлов в составе вычислительного кластера // Суперкомпьютеры: вычислительные и информационные технологии: материалы междунар. науч.- практ. конф., Хабаровск, 30 июня – 2 июля 2010 г. – Хабаровск: Изд-во Тихоокеанского гос. ун-та, 2010. – С. 77–84.

182. Мальковский С.И., Пересветов В.В. Зависимость производительности передачи сообщений MPI от порядка распределения процессов в четырехпроцессорном узле вычислительного кластера // Суперкомпьютеры: вычислительные и информационные технологии: материалы междунар. науч.-практ. конф., Хабаровск, 30 июня – 2 июля 2010 г. – Хабаровск: Изд-во Тихоокеанского гос. ун-та, 2010. – С. 85–92.

183. Малявко А., Менжулин С. Суперкомпьютеры и системы. Мультипроцессоры. – Litres, 2022.

184. Марпл-мл. С.Л. Цифровой спектральный анализ и его приложения: Пер. с англ. – М.: Мир, 1990. – 584 с, ил.

185. Метрики параллельных задач. URL: <https://en.ppt-online.org/440998> (дата обращения 21.08.2025)
186. МЦСТ Эльбрус. Компилятор LCC (ТВГИ.00500-01). <https://www.mcst.ru/lcc>. (дата обращения 09.04.2025).
187. Паттерсон Д. А., Хеннесси Д. Л. Архитектура компьютера и проектирование компьютерных систем. 4-е издание. Питер – 2012.
188. Полянсков Ю.В. Интеграция САПР-, PDM-, ERP-систем в единое информационное пространство производственного предприятия. / Ю.В. Полянсков, А.С. Кондратьева, М.С. Черников, А.А. Блюменштейн. – Известия Самарского научного центра Российской академии наук, т. 15, № 4(3), 2013.
189. Роби Р., Замора Дж. Параллельные и высокопроизводительные вычисления / пер. с англ. А В. Логунова. М.: ДМК Пресс, 2022. 704 с.
190. Список языков запросов свойств программ. <http://cs.nyu.edu/~lharris/content/programquerylangs.html>.
191. Стешина Д.А., Кочан И.Н. T-FLEX PLM: импортозамещение – это реально! Автоматизация в промышленности. 2015. № 1. С. 47-49.
192. Филисов Д.А. Стратегии оптимизации для высоконагруженных приложений: повышение общей производительности // Вестник науки. 2023. №7 (64). URL: <https://cyberleninka.ru/article/n/strategii-optimizatsii-dlya-vysokonagruzhennyh-prilozheniy-povyshenie-obschey-proizvoditelnosti>
193. Хорн Р., Джонсон Ч. Матричный анализ. – Мир, 1989. – Т. 656.
194. Хорошевский В.Г. Распределенные вычислительные системы с программируемой архитектурой // Вестник СибГУТИ. 2010. № 2. С.3-41.
195. Шаповалов О.В., Сергеев Е.С., Чалышев В.С., Катаев А.В., Крыжановский Д.И., Андреев А.Е.. Вопрос распараллеливания в разработке ядра геометрического моделирования. Научный сервис в сети Интернет: все грани параллелизма, 2013.

ПРИЛОЖЕНИЯ

Приложение А – Свидетельства о регистрации программ для ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2026667836

Облачная платформа гибридного анализа кеш-эффективности программных систем

Правообладатель: *Федеральное государственное бюджетное образовательное учреждение высшего образования «Волгоградский государственный технический университет» (RU)*

Авторы: *Егунов Виталий Алексеевич (RU), Кравец Алла Григорьевна (RU), Ковалев Иван Владимирович (RU), Петров Валерий Сергеевич (RU)*



Заявка № 2026667557

Дата поступления 10 июня 2026 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 10 июня 2026 г.

Руководитель Федеральной службы
по интеллектуальной собственности

документ подписан электронной подписью
Сертификат 00a570e47ad38d531b4b8818e75f29506
Владелец **Зубов Юрий Сергеевич**
Действителен с 04.09.2025 по 28.11.2026

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2026667837

Плагин расширенной аналитики кода для VS Code

Правообладатель: *Федеральное государственное бюджетное образовательное учреждение высшего образования «Волгоградский государственный технический университет» (RU)*

Авторы: *Егунов Виталий Алексеевич (RU), Кравец Алла Григорьевна (RU), Ковалев Иван Владимирович (RU), Петров Валерий Сергеевич (RU)*

Заявка № 2026667559

Дата поступления **10 июня 2026 г.**

Дата государственной регистрации

в Реестре программ для ЭВМ **10 июня 2026 г.**



*Руководитель Федеральной службы
по интеллектуальной собственности*

документ подписан электронной подписью
Сертификат 00a570e4f7add8d531b4b8818e75f29506
Владелец **Зубов Юрий Сергеевич**
Действителен с 04.03.2025 по 28.11.2026

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2026667841

**Платформа веб-мониторинга эффективности
программного кода**

Правообладатель: *Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Волгоградский государственный технический
университет» (RU)*

Авторы: *Егунов Виталий Алексеевич (RU), Кравец Алла
Григорьевна (RU), Ковалев Иван Владимирович (RU),
Петров Валерий Сергеевич (RU)*



Заявка № **2026667568**

Дата поступления **10 июня 2026 г.**

Дата государственной регистрации

в Реестре программ для ЭВМ **10 июня 2026 г.**

*Руководитель Федеральной службы
по интеллектуальной собственности*

ДОКУМЕНТ ПОДПИСАН ЭЛЕКТРОННОЙ ПОДПИСЬЮ
Сертификат 00a570e4f7ad38d531b4b8818e75f29506
Владелец **Зубов Юрий Сергеевич**
Действителен с 04.09.2025 по 28.11.2026

Ю.С. Зубов



о государственной регистрации программы для ЭВМ

№ 2025662909

Модуль расчет метрик качества программы на языке С

Правообладатель: **Федеральное государственное бюджетное образовательное учреждение высшего образования «Волгоградский государственный технический университет» (RU)**

Автор(ы): *Егунов Виталий Алексеевич (RU)*



Заявка № 2025662215

Дата поступления **23 мая 2025 г.**

Дата государственной регистрации

в Реестре программ для ЭВМ **23 мая 2025 г.**

Руководитель Федеральной службы
по интеллектуальной собственности

ДОКУМЕНТ ПОДПИСАН ЭЛЕКТРОННОЙ ПОДПИСЬЮ
Сертификат: SW926708-2006-0A240M70Bca2006
Иванов Зубов Юрий Сергеевич
Доверенность № 001 от 03.10.2025

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2024667441

**Программа для вычисления сингулярного разложения
симметричной матрицы с использованием CUDA**

Правообладатель: *Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Волгоградский государственный технический
университет» (RU)*

Авторы: *Безрученко Алексей Юрьевич (RU), Егунов Виталий
Алексеевич (RU), Кравец Алла Григорьевна (RU)*

Заявка № 2024666789

Дата поступления **24 июля 2024 г.**

Дата государственной регистрации

в Реестре программ для ЭВМ **24 июля 2024 г.**



*Руководитель Федеральной службы
по интеллектуальной собственности*

ДОКУМЕНТ ПОДПИСАН ЭЛЕКТРОННОЙ ПОДПИСЬЮ
Сертификат: 429646096.346136404f06f83673b4aa7
Владелец: **Зубов Юрий Сергеевич**
Действителен с 18.07.2023 по 02.08.2024

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2023664318

Модуль поддержки векторизации матричных преобразований

Правообладатель: *Федеральное государственное бюджетное образовательное учреждение высшего образования «Волгоградский государственный технический университет» (RU)*

Авторы: *Егунов Виталий Алексеевич (RU), Кравец Алла Григорьевна (RU)*

Заявка № 2023663723

Дата поступления 03 июля 2023 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 03 июля 2023 г.



*Руководитель Федеральной службы
по интеллектуальной собственности*

ДОКУМЕНТ ПОДПИСАН ЭЛЕКТРОННОЙ ПОДПИСЬЮ
Сертификат 429b6a0fe3853164ba96183b73b4aa7
Владелец **Зубов Юрий Сергеевич**
Действителен с 10.05.2023 по 02.08.2024

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2023689295

Программа обработки массивов данных случайных величин перевозочного процесса грузов и пассажиров с расчетом параметров закона Эрланга

Правообладатель: *Федеральное государственное бюджетное образовательное учреждение высшего образования «Волгоградский государственный технический университет» (RU)*

Авторы: *Куликов Алексей Викторович (RU), Симонова Ирина Эдуардовна (RU), Егунов Виталий Алексеевич (RU), Павлов Павел Алексеевич (RU), Куликов Андрей Алексеевич (RU)*

Заявка № 2023689163

Дата поступления 27 декабря 2023 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 27 декабря 2023 г.



Руководитель Федеральной службы
по интеллектуальной собственности

ДОКУМЕНТ ПОДПИСАН ЭЛЕКТРОННОЙ ПОДПИСЬЮ
Сертификат 429b6a09e3851164b996f83b73b4aa7
Владелец **Зубов Юрий Сергеевич**
Действителен с 19.01.2023 по 02.08.2024

Ю.С. Зубов

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2014618996

Автоматизированная система расчета подачи резца при
продольном точении нежестких валов на станках с ЧПУ

Правообладатель: **Федеральное государственное бюджетное
образовательное учреждение высшего профессионального
образования «Волгоградский государственный технический
университет» (RU)**

Авторы: **Жданов Алексей Андреевич (RU), Плотников Александр
Леонтьевич (RU), Чигиринский Юлий Львович (RU), Егунов
Виталий Алексеевич (RU)**

Заявка № 2014616776

Дата поступления 14 июля 2014 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 05 сентября 2014 г.



Врио руководителя Федеральной службы
по интеллектуальной собственности

Л.Л. Кирий

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2012614836

Программа управления сменными модулями масштабируемой
архитектуры мобильного робота

Правообладатель(ли): *Федеральное государственное бюджетное
образовательное учреждение высшего профессионального
образования «Волгоградский государственный
технический университет» (RU)*

Автор(ы): *Егунов Виталий Алексеевич,
Потапов Максим Игоревич (RU)*

Заявка № 2012612349

Дата поступления 2 апреля 2012 г.

Зарегистрировано в Реестре программ для ЭВМ
30 мая 2012 г.

Руководитель Федеральной службы
по интеллектуальной собственности

Б.П. Симонов



Приложение Б – Документы, подтверждающие внедрение результатов диссертации



Современные технологии - в промышленности

Программное обеспечение
Инженерный консалтинг
Вычислительная техника
Измерительное оборудование

АКТ о внедрении (использовании) результатов диссертационной работы Егунова Виталия Алексеевича

Результаты диссертационной работы Егунова В.А., тематика которой связана с разработкой моделей и методов проектирования программных систем, представленной на соискание ученой степени доктора технических наук, использованы в процессе создания и оптимизации программного обеспечения, разрабатываемого компанией «ТЕСИС» для моделирования гидроаэродинамики и тепловых процессов, а также решения междисциплинарных задач, в частности – при разработке прототипов расчетных библиотек. Исследования начинались в 2013 - 2015 годах в рамках ряда НИР, в том числе № 44/605-13 «Разработка и анализ алгоритмов реализации матричных операций с использованием векторных расширений систем команд процессоров», № 44/639-13 «Разработка и тестирование алгоритмов реализации блочных матричных операций с использованием векторных расширений систем команд процессоров» и № 44/776-15 «Модификация алгоритмов реализации матричных операций с использованием расширений систем команд, дополненных совмещенными сложениями – умножениями (FMA3), и архитектур процессоров с длинным командным словом», и нашли свое продолжение в 2024 году в рамках предпроектных исследований применения графических сопроцессоров.

Разработанные в рамках НИР алгоритмы использовались для выполнения базовых блочных операций линейной алгебры (умножение матриц, обращение и факторизация матриц) для разреженных мелко-блочных матриц и блоков векторов (плотных матриц определенной размерности) с применением векторных расширений системы команд микропроцессоров Intel и использованием возможностей графических сопроцессоров. На основе базовых операций компанией «ТЕСИС» были разработаны алгоритмы умножения разреженных мелко-блочных матриц на блоки векторов, LU-разложения, обращения матриц и решения систем линейных алгебраических уравнений

Реализация разработанных алгоритмов и методов позволила получить значительное ускорение выполнения базовых операций до 10 раз относительно предыдущих программных реализаций при использовании технологии векторизации вычислений AVX2. Разработанные алгоритмы и методы,



использующие графические сопроцессоры, позволяют значительно, до 50-75%, уменьшить время вычислений с плотными и разреженными матрицами за счет распределения вычислений между центральным процессором и графическим сопроцессором, оптимизации вычислений под конкретные платформы, интеграции программного обеспечения с существующими высокопроизводительными программными библиотеками. Реализация оптимизированных алгоритмов решения систем линейных уравнений с разреженными или плохо обусловленными матрицами показала повышение скорости расчетов до 15% и точности до 5-7%.

Кроме того, полученные наработки в области векторизации вычислений и распределения вычислений между центральным процессором и графическим сопроцессором компания «ТЕСИС» предполагает использовать при построении вычислительных библиотек для программного комплекса «FlowVision» в 2025 – 2027 г., что подтверждает перспективность результатов диссертационной работы Егунова В.А.

Генеральный директор
ООО «ТЕСИС»



С.Н. Курсаков



АО «Топ Системы»
ИНН/КПП 7726601967/772601001
115191, г. Москва, пер. Гамсоновский, д.2,
стр.3, офис 50.
Тел/факс: +7 (499) 973-20-34, 973-20-35
E-mail: tflex@topsystems.ru

Исх. № 01/0410/СТ от 10.04.2025 г.

АКТ
об использовании результатов
диссертационной работы
Егунова Виталия Алексеевича

Результаты научной работы Егунова В.А., полученные в рамках диссертационного исследования на соискание ученой степени доктора технических наук, тематика которого связана с разработкой моделей и методов проектирования программных систем, используются в ЗАО «Топ Системы» с 2019 года в инструментальных компонентах, разработанных для системы управления полным жизненным циклом изделий для промышленных предприятий «САРУС». Работы выполнялись в рамках НИР № 44/227-19 «Разработка специализированных объектно-ориентированных библиотек классов, предназначенных для решения геометрических задач прототипа САД системы, связанных с генерацией, модификацией и анализом трёхмерных математических моделей».

Разработанные методы и алгоритмы вычислительно-сложных процедур использованы в библиотеках классов, предназначенных для решения геометрических задач в прототипе САД-системы 3-мерного моделирования: построения эквидистантных линий к проволочному телу; реализации операций вычисления масс-инерционных характеристик проволочных, твердых и листовых тел с использованием интегрирования по сетке; оптимизации ряда базовых расчетных классов библиотеки геометрических операций, связанных с реализацией булевых операций с телами; выполнения кинематических операций, построения пересечений и поиска самопересечений и оптимизации частных случаев обрезки кривых и поверхностей.

Научно-техническая и практическая ценность полученных результатов, достигнутые технико-экономические показатели, степень научной новизны разработки – высокие. Полученные результаты позволяют повысить импортонезависимость получаемого программного обеспечения.

Использование указанных результатов позволило повысить эффективность программного обеспечения: уменьшить время, необходимое для реализации базовых расчетных процедур до 18%, повысить точность вычислений до 7%. Экономический эффект от внедрения решений, использующих результаты диссертационной работы Егунова В.А., оценивается в 1,8 млн. рублей в год.

С уважением,
Генеральный директор
АО «Топ Системы»



С.А. Кураксин



Открытое акционерное общество
«Волжский абразивный завод»



Улица им. Ф.Г. Логина, д. 169, г. Волжский Волгоградская область, 404119, Россия
ИНН: 3435000467; КПП: 343501001; ОКПО: 00220931; ОГРН: 1023402019596
E-mail: abraziv@vabz.ru; <http://www.vabz.ru>
Справка: (8443) 41-24-02; 41-04-33; Приемная: (8443) 52-53-39; Факс: (8443) 41-04-51

АКТ
о внедрении (использовании) результатов
диссертационной работы
Егунова Виталия Алексеевича

Результаты научной работы Егунова В.А., полученные в рамках диссертационного исследования на соискание ученой степени доктора технических наук, используются в ОАО «Волжский абразивный завод» с августа 2023 года.

Разработанные алгоритмы и методы использованы для реализации программных систем для сбора и обработки данных с производственных участков электро-плавильного цеха (ЭПЦ), а также для многокритериального анализа факторов, влияющих на объем и качество выпускаемой продукции.

Научно-техническая и практическая ценность полученных результатов, достигнутые технико-экономические показатели, степень новизны и оригинальность разработки – высокие. Полученные результаты готовы для практического использования.

Использование указанных результатов позволило повысить эффективность сбора и обработки данных ЭПЦ на 17-20%, снизить количество ошибок, обусловленных человеческим фактором на 7%. Годовой экономический эффект, полученный от внедрения результатов научной работы Егунова В.А., оценивается в 1 млн. рублей.

Заместитель генерального директора
по развитию цифровых технологий



В.Ю. Наумов
10.08.2023г.

В.Ю. Наумов