Южный федеральный университет

На правах рукописи

Юрушкин Михаил Викторович

Оптимизация размещения массивов в общей памяти

Специальность 05.13.11 —
Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
д. т. н., с.н.с.
Б.Я. Штейнберг

Ростов-на-Дону — 2016

Оглавление

Введение

Актуальность

Высокопроизводительные вычисления используются во многих отраслях экономики. Областями применения высокопроизводительных вычислений являются оборонная промышленность, создание новых лекарств, высокочастотная торговля фьючерсами, моделирование деталей автомобилей, визуализация киноэффектов, прогнозирование изменений климата и т.д.

Для того чтобы удовлетворить увеличивающийся спрос на высокую производительность современные компьютеры развиваются со стремительной скоростью. В 2005 году был выпущен первый двуядерный процессор. Теперь же количество ядер в современных процессорах достигает 16 и более. Появились новые виды вычислительных систем – видеокарты, количество вычислительных элементов которых измеряется в тысячах, а пиковая производительность измеряется в терафлопсах.

На фоне уменьшения времени выполнения арифметических операций за счет увеличения количества ядер, доступ к памяти становится все более узким местом в производительности вычислительных систем. С целью увеличения эффективности кеш-памяти давно используются блочные вычисления (тайлинг), которые позволяют подстроить алгоритм под иерархию памяти и, тем самым, уменьшить количество кеш-промахов. Также используются выравнивание данных и нестандартные размещения данных в памяти.

В некоторых случаях вместе с тайлингом используется блочное размещение массивов, позволяющее получить еще большее ускорение. Непосредственное использование блочного размещения массивов затруднительно, т.к. требует высокой квалификации программиста и много времени на разработку.

Степень разработанности темы

Ускорение программ с помощью перехода к блочным вычислениям исследовалось в работах Н.А. Лиходеда, В.Э Малышкина, Ф.Г. Густавсона, Моники Лэм, Майкла Вульфа, Д. Донгарры, Чарльза Ван Лоана. Дополнительное ускорение блочных вычислений за счет нестандартных размещений массивов в памяти рассматривалось в работах Д. Донгарры, В. Прасанна, Л. Карлсона.

Большинство работ, посвященных исследованию блочного размещения данных, связано с написанием вручную высокопроизводительных блочных алгоритмов, использующих блочно размещенные матрицы. Современные компиляторы и библиотеки на данный момент не поддерживают блочные размещения данных, ориентируясь только на стандарт языка программирования, который предписывает хранить матрицы либо по строкам (Си, ПАСКАЛЬ), либо по столбцам (ФОРТРАН). Исключение составляет пакет PLASMA для решения задач линейной алгебры. В нем реализованы алгоритмы некоторых важных задач линейной алгебры (умножение матриц, LU-разложение матрицы, QR-разложение матрицы, разложение Холецкого и т.д.), которые работают с блочно размещенными матрицами. Также в нем реализованы специальные функции, которые производят переразмещения из стандартного вида в блочный вид хранения матриц.

Описанные выше исследования показали высокую эффективность блочных вычислений и блочных размещений матриц. Но не рассмотрены возможности двойного блочного размещения матриц и не рассмотрены возможности использования блочных размещений матрицы компилятором. Не рассматривалась до сих пор основанная на модели задача прогноза времени выполнения программ, использующих блочное размещение матриц, на будущих процессорах.

Все это подчеркивает актуальность исследований, посвященных оптимизации размещений массивов в общей памяти и, в частности, автоматизации таких размещений.

Объект исследований

Программы, ориентированные на большие объемы вычислений и использующие общую память.

Цель работы

Разработка методов и средств ускорения параллельных программ на основе оптимизации размещения массивов в общей памяти.

Сформулированная цель может быть достигнута путем решения следующих **задач**:

1. Создание модели времени выполнения программ для систем с общей памятью.
2. Использование нестандартных размещений данных для построения высокооптимизированного алгоритма умножения матриц.
3. Автоматизация блочного размещения данных.
4. Модификация Оптимизирующей распараллеливающей системы.

Методология

Методология исследования основана на последовательной идентификации и анализе проблемы, а также анализе существующей литературы. При исследовании предлагаемой модели времени выполнения программ использовалась модель процессора, существенно приближенная к существующим процессорам. Для проверки корректности модели времени выполнения программ использовался метод численного эксперимента. При реализации высокопроизводительного алгоритма умножения матриц, а также алгоритма автоматизации блочного размещения матриц в обшей памяти использовались методы низкоуровневой оптимизации программ [50], [56], [80], теории графов [97], теории распараллеливающих/оптимизирующих преобразований программ [31], [32] линейной алгебры [92], [93]. Для проверки эффективности предлагаемых алгоритмов также использовался метод численного эксперимента.

Научная новизна

- Разработан новый алгоритм умножения матриц, который показывает бОльшую производительность по сравнению с известными пакетами OpenBLAS, Intel MKL и PLASMA.
- Разработана модель времени выполнения программ для систем с общей памятью, которая отличается от известных учетом латентности разных уровней иерархии памяти.
- Впервые разработана система поддержки автоматизированного блочного размещения многомерных массивов в оперативной памяти компилятором, которая позволяет получать ускорение для широкого класса программ.

Теоретическая значимость

Полученные в диссертации результаты могут стать основой для создания новых методов для эффективного отображения высокоуровневых программ на вычислительные архитектуры будущего.

Практическая значимость

Разработанные директивы блочного размещения данных позволяют компиляторам генерировать более быстрый код. Реализованный алгоритм умножения матриц может быть использован для ускорения многих задач линейной алгебры. Разработанная модель времени выполнения программ может быть использована для прогнозирования времени работы программ. Реализованный парсер языка ФОРТРАН позволяет использовать систему ОРС для оптимизации программ, написанных на языке ФОРТРАН. Полученные результаты могу быть использованы для создания новых инструментов разработки быстрых программ.

Личный вклад

Все результаты, приведенные в диссертации, разработаны автором лично. В исследованиях использовалась система ОРС, которая разработана на мехмате ЮФУ. Автор диссертации является одним из разработчиков ОРС.

Использование результатов работы

Часть результатов диссертации используются в учебном процессе мехмата Южного федерального университета в магистерской программе “Высокопроизводительные вычисления и технологии параллельного программирования”. Результаты диссертации могут использоваться предприятиями, разрабатывающими системное и высокопроизводительное программное обеспечение.

Гранты, поддерживавшие исследования диссертации

- Стипендия Правительства РФ 2014 г.
- ФЦП «Исследования и разработки по приоритетным направлениям научно-технологического комплекса России на 2009-2013 гг» «Распараллеливание программ с одновременной оптимизацией обращений к памяти» по государственному контракту от «10» октября 2013 г. № 14.514.11.4104
- Внутренний грант ЮФУ по Программе развития университета на период до 2021 года в целях повышения конкурентоспособности среди ведущих мировых научно-образовательных центров «Разработка библиотеки алгоритмов решения задач с теплицевыми матрицами и анализ их вычислительных характеристик» 01.05.2013 – 15.12.2013
- Лаборатория Ангстрем-ЮФУ.
- ФЦП «Научные и научно-педагогические кадры инновационной России», по теме «Создание биоинформационной технологии поиска взаимосвязанных сценариев организации в геномах животных и человека некодирующей ДНК и кодирующей белок ДНК» Государственный контракт № 14.740.11.0006 от 1 сентября 2010.

- ФЦП «Научные и научно-педагогические кадры инновационной России» на 2009-2013 годы по лоту «Проведение научных исследований коллективами научно-образовательных центров в области информатики» шифр «2009-1.1-113-050» по теме: «Диалоговый высокоуровневый оптимизирующий распараллеливатель программ и его приложения» (шифр заявки «2009-1.1-113-050-043») Государственный контракт № 02.740.11.0208 от 7 июля 2009 г.
- Внутренний грант Южного федерального университета «Учебно-научная лаборатория оптимизации и распараллеливания программ», 2007г.

Степень достоверности и апробация результатов работы

Достоверность результатов диссертации подтверждается математической строгостью изложения алгоритмов, многими численными экспериментами.

Результаты, изложенные в диссертации, опубликованы в 13 научных работах [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74] включая 6 в журналах перечня ВАК, один из которых индексируется в БД «Scopus», 1 монографию и 6 материалов научных конференций. Имеется свидетельство о государственной регистрации программы ЭВМ. Эти результаты также докладывались на семинарах и конференциях:

1. Национальный суперкомпьютерный форум (НСКФ) 2015.
2. Московский суперкомпьютерный форум (МСКФ) 2014.
3. Параллельные вычислительные технологии (ПаВТ) 2014.
4. Московский суперкомпьютерный форум (МСКФ) 2013.
5. YSC-2013 "High Performance Computing and Simulation", Барселона, Испания. 2013.
6. VI сессия научной школы-практикума молодых ученых и специалистов «Технологии высокопроизводительных вычислений и компьютерного моделирования: технологии eScience».НИУ ИТМО. 09.04.13-12.04.13
7. XIV молодежная конференция-школа с международным участием "Современные проблемы математического моделирования", п. Абрау-Дюрсо, 12-17 сентября 2011 года.

8. На научном семинаре ФГУП Квант, весна 2011 г.
9. «Научный сервис в сети Интернет», г. Новороссийск, 20-26 сентября 2010 г.
10. На постоянно действующем научном семинаре мехмата ЮФУ «Автоматическое распараллеливание программ».

Основные положения, выносимые на защиту

1. Высокопроизводительный алгоритм умножения матриц, использующий нестандартное размещение данных.
2. Автоматизация блочного размещения данных в оперативной памяти.
3. Модель времени выполнения программ для систем с общей памятью.

Структура и объем диссертации

Диссертация состоит из введения, четырех глав, заключения, списка сокращений и списка литературы. Текст диссертации изложен на 127 страницах и содержит 15 примеров, 23 рисунка и 16 таблиц. Список литературы содержит 104 наименования.

Основное содержание работы

Первая глава носит вспомогательный характер. В ней описываются понятия необходимые для изложения результатов диссертации. В **параграфе 1.1** приведены принципы работы иерархии памяти современных вычислительных систем. Дается определение кеш-памяти, виртуальной памяти, кеш-памяти TLB. **Параграф 1.2** посвящен методу тайлинга, который предназначен для оптимизации временной локальности данных. В **параграфе 1.3** приведено определение пространственной локальности данных, а также различным метода размещения данных. В частности, описывается блочное размещение данных, автоматическую поддержку которого реализовал автор диссертации в системе OPC. В **параграфе 1.4** приводится понятие выравнивания данных в памяти. На примерах представлено как производится выравнивание данных современными компиляторами. **Параграф 1.5** содержит обзор компиляторов, которые содержат

в своем составе нестандартные оптимизации работы с памятью. В частности, приводится описание проекта TALC, который позволяет программисту размещать структуры и массивы согласно определенной схемы.

Вторая глава посвящена модели времени выполнения программ для иерархии памяти. Приводится абстрактная формула времени выполнения программ. На основе данной формулы в **параграфах 2.1-2.2** строятся специализированные формулы времени выполнения программ, реализующих алгоритм блочного умножения матриц, а также алгоритма Флойда-Уоршалла. В **параграфе 2.3** описывается метод статического определения количества кеш-промахов, возникших во время выполнения алгоритма.

Третья глава посвящена построению высокопроизводительного алгоритма умножения матриц. В **параграфе 3.1** описываются существующие высокопроизводительные алгоритмы умножения матриц. В частности, упоминаются пакеты Atlas, Intel MKL, PLASMA и OpenBLAS. В **параграфе 3.2** приводится описание алгоритма умножения матриц. Его особенностью является то, что данные хранятся в памяти небольшими блоками нестандартным образом, а не стандартным образом. В **параграфе 3.3** приводятся результаты численных экспериментов, в которых сравнивается производительность программной реализации разработанного алгоритма, а также существующих других высокопроизводительных реализаций (Intel MKL, PLASMA, OpenBLAS). Согласно результатам численных экспериментов представленный в диссертации алгоритм по производительности превосходит остальные алгоритмы, и его производительность составляет в среднем 98-99% от теоретической производительности процессора.

В четвертой главе приводится алгоритм автоматизации блочных размещений данных в системе ОРС. Такая автоматизация реализована на уровне директив компилятора, описание которых приводится в **параграфе 4.2.** Реализованные директивы позволяют блочно размещать массивы произвольной размерности. В **параграфах 4.3-4.4** приводятся программные проекты, в которых можно протестировать работу данных директив. В **параграфах 4.5-4.5** строится

алгоритм обработки компилятором директив блочного размещения данных. **Параграф 4.7** содержит результаты численных экспериментов, в которых сравнивается время работы программ, скомпилированных в двух режимах – без директив блочного размещения данных и без них. Результаты численных экспериментов подтверждают, что реализованная автоматизация блочных размещений дает дополнительное ускорение для некоторых классов задач.

В заключении подводятся итоги исследований, представленных в диссертации. Описываются основные полученные результаты, решенные задачи, обсуждается их новизна и практическая значимость. Обосновывается достижение цели диссертации.

Глава 1

Используемые методы оптимизации работы с данными программы

1.1. Дисбаланс производительности процессора и памяти.

Вычислительные системы развиваются на протяжении последних десятилетий с большой скоростью. Согласно постоянно обновляющимся спискам Top500 производительность компьютеров увеличивается каждый год на 30 процентов. Такой рост до 2005 года был обусловлен наращиванием частоты процессоров. С 2005 года по наши дни наблюдается больший уклон в сторону увеличения количества ядер внутри одного процессора. Это связано с тем, что увеличивать частоту еще больше уже невозможно на практике [57]. Разрыв между временем обработки данных процессором и временем считывания данных из памяти в процессор увеличивается с каждым годом. Поэтому оптимизация работы с памятью является важной задаче. Память современных процессоров по степени удаленности от процессора имеет иерархическую структуру и включает в себя такие уровни как оперативная память, многоуровневая кеш-память и регистровая память (Рисунок 1).

Рис. 1: Иерархия памяти многих современных процессоров.

Чем ближе уровень памяти к процессору, тем больше скорость считывания данных из этой памяти, но, в то же время, меньше ее размер. Иерархическая структура памяти продиктована тем, что многие алгоритмы в определенный промежуток времени обращаются к фиксированному набору данных несколько раз.

1.1.1. Кеш-память процессора

Кеш-память процессора – это память, которая находится на одном кристалле с процессором. В современных процессорах кеш-память по степени удаленности от процессора разделяется на несколько уровней. В частности, некоторые процессоры Intel [58], [59] включают в себе 3 уровня кеш-памяти, которые называются кеш-памятью L1, кеш-памятью L2 и кеш-памятью L3. Кеш-память L1 расположена ближе всего к процессору. Соответственно, скорость доступа к ней выше чем скорость доступа к остальным уровням кеш-памяти. Размер кеш-памяти L1 составляет десятки килобайт. Кеш-память L3 является самым медленной, но, в то же самое время, и самой большой. Ее размер исчисляются в мегабайтах. В некоторых многоядерных процессорах [58], [59] кеш-память L3 является общей для всех ядер, в то время как каждое ядро имеет свою кеш-память меньших уровней.

Кеш-память процессора может хранить не только данные, но и инструкции (код исполняемых в данный момент программ). Например, в [58] кеш-память L3 позволяет хранить как данные, так и исполняемый код программ. В кеш-памяти L1 и кеш-памяти L2 данные и инструкции хранятся в двух разных уровнях кеш-памяти, которые называются кеш-памятью данных (L1d, L2d) и кеш-памятью инструкций (L1i, L2i) соответственно. В данной работе речь в большей степени будет идти о кеш-памяти данных.

В тот момент времени, когда данное считывается из оперативной памяти, производится проверка на наличие этого данного в кеш-памяти процессора. Если данное находится в кеш-памяти, то обращение к оперативной памяти не осуществляется. Такая ситуация называется попаданием в кеш-память (cache hit). В противном случае происходит кеш-промах (cache miss) - считывание данного из оперативной памяти. Т.к. время считывания данных из оперативной памяти намного выше чем из кеш-памяти, то возникает задача минимизации кеш-промахов. Один из методов минимизации кеш-промахов описан в разделе 1.2.

Рассмотрим более подробно процесс сохранения данных в кеш-памяти. Будем предполагать, что адрес данного представляет собой 64-битное целое. Физически кеш-память представляет собой набор банков памяти (Рисунок 2).

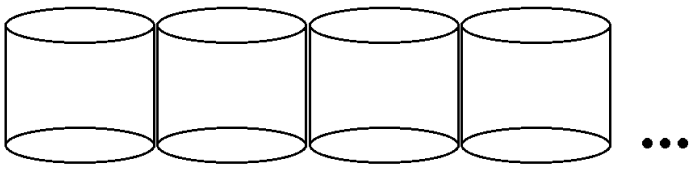

Рис 2: Банки кеш-памяти.

Данные пересылаются из памяти в кеш-память и наоборот небольшими блоками - кеш-линейками. Размер кеш-линейки обычно равен 32 либо 64 байт и зависит от процессора. Сохранение кеш-линейки в кеш-памяти означает сохранение этой кеш-линейки в определенном банке памяти кеш-памяти. Для того чтобы определить номер банка памяти, в который сохраняется кеш-линейка,

необходимо проанализировать адрес запрашиваемого данного, Его можно разделить на 3 части (Рисунок 3)

- Тэг (T)
- Номер банка памяти, в который попадает данное (S).
- Смещение относительно начала кеш-линейки (O).

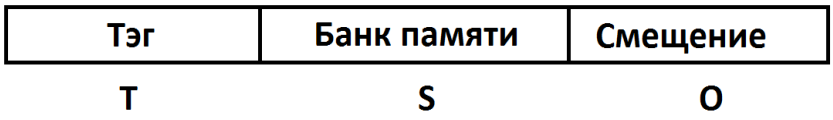

Рис 3: Адрес элемента.

Очевидно, что T + S + O = 64 (размер адреса).

O = log(CacheLineSize).

Число S определяет номер банка памяти, в который попадет запрашиваемая кеш-линейка. Так, например, при последовательном чтении массива данные будут сохраняться в последовательных банках памяти. Соответственно, при достижении последнего банка памяти следующий банк памяти будет иметь номер 0. При таком методе считывания данных реально используемое количество банков памяти равно максимальному значению, кеш-память используется максимально эффективно.

Размер банка памяти, т.е. количество кеш-линеек, которые он может вместить, называется *ассоциативностью кеш-памяти*. Увеличение ассоциативности кеш-памяти приводит к усложнению ее проектирования. Это связано с тем, что усложняется схема устройства, производящего проверку на наличие запрашиваемой кеш-линейки в банке памяти.

Если в кеш-памяти присутствует всего один банк памяти (S равен 0), то кеш-память называется *полностью ассоциативной.* Если банк памяти может хранить одну кеш-линейку, то кеш-память называется *кеш-памятью прямого отображения*. Кеш-память прямого отображения является самой простой с точки зрения проектирования. Большим ее минусом является то, что при ее

использовании вероятность возникновения кеш-промахов намного больше, чем при использовании полностью ассоциативной кеш-памяти. На практике в качестве компромисса между полностью ассоциативной кеш-памятью и кеш-памятью прямого отображения часто используется кеш-память с k-канальной ассоциативностью, при которой каждый банк памяти может хранить k кеш-линеек. Число k обычно варьируется в диапазоне 4-16. В некоторых случаях кеш-память L1 является полностью ассоциативной.

Последний элемент адреса (T) называется тэгом. По тэгу кеш-линейки производится проверка на ее наличие в банке памяти. Если кеш-линейки с таким тегом нет, то возникает ситуация кеш-промаха.

При попытке сохранения кеш-линейки в банке памяти, происходит операция вытеснения из него другой кеш-линейки. Существует несколько алгоритмов вытеснения данных из банков памяти, и каждый из них имеет свои плюсы и минусы. На данный момент наиболее часто используемыми методами вытеснения данных является метод вытеснения наиболее давно использовавшихся данных (LRU - Least Recently Used), а также метод вытеснения наиболее недавно использовавшихся данных (MRU - Most Recently Used).

Кеш-память делится на *инклюзивную (включающую)* и *эксклюзивную (исключающую)* кеш-память. В *инклюзивной кеш-памяти* выполняется условие: если кеш-линейка хранится в младшем уровне кеш-памяти, то она хранится также и в старших уровнях кеш-памяти. В эксклюзивной кеш-памяти наоборот: кеш-линейки хранятся только в одном из уровней кеш-памяти.

Важной особенностью работы кеш-памяти является также то, что программист не имеет возможности адресовать находящиеся в ней данные. Единственная возможность явно повлиять на работу кеш-памяти состоит в программной предвыборке данных (software data prefetching). Программная предвыборка данных заключается в заблаговременной перекачке данных из оперативной памяти в кеш-память с целью уменьшения времени простаивания процессора. Такая обработка данных зачастую позволяет эффективно распараллелить вычисления и загрузку данных.

Набор операций предвыборки данных зависит от архитектуры процессора. Так, например, в некоторых процессорах Intel с инклюзивной кеш-памятью [58], [59] поддерживается программная предвыборка данных в нескольких режимах (Таблица 1).

Таблица 1. Набор операций предвыборки данных в архитектуре x86.

PREFETCH0	Закачка кеш-линейки во все уровни кеш-памяти
PREFETCH1	Закачка кеш-линейки во все уровни кеш-памяти за исключением кеш-памяти L1
PREFETCH2	Закачка кеш-линейки во все уровни кеш-памяти за исключением кеш-памяти L2
PREFETCHNTA	Закачка кеш-линейки в кэщ-память с учетом того, что ее не следует сохранять в кеш-памяти на долгое время

Стоит отметить, что команды предвыборки данных имеют лишь рекомендательный для процессора характер, т.к. они не влияют на результат работы программы.

Для того чтобы произвести программную предвыборку данных программист обязан вставить в код программы вызовы специальной функции, осуществляющей предвыборку данных. Функция, позволяющая программисту произвести программную предвыборку данных, часто реализуется в компиляторах в виде встроенной (intrinsic) в компилятор функции. Так, например, в компиляторе MSVC 2010 функция предвыборки кеш-линейки объявлена следующим образом:

```
void _mm_prefetch(char * p , int type);
```

где аргумент p является адресом кеш-линейки, а аргумент type принимает одно из следующих значений:

- _MM_HINT_T0 - закачка кеш-линейки во все уровни кеш-памяти.

- _MM_HINT_T1 - закачка кеш-линейки в L2 и L3 уровни кеш-памяти.
- _MM_HINT_T2 - закачка кеш-линейки в L3-уровень кеш-памяти.
- _MM_HINT_NTA – особый режим предвыборки данных, позволяющий закачивать данные в кеш-память L1 в обход остальных уровней кеш-памяти. Этот режим стоит использовать только при подкачке данных, обрабатываемых только один раз.

Многие существующие процессоры являются многоядерными. В таких процессорах кеш-память L1 и кеш-память L2 реализуются локальными для каждого из ядер в отдельности. Кеш-память L3 является общей для всех ядер и может использоваться для синхронизации данных различных уровней кеш-памяти L2. Такая кеш-память должна обладать свойством когерентности. Когерентность кеш-памяти – это свойство кеш-памяти, означающее целостность данных, хранящихся в локальных уровнях кеш-памяти.

Существует много протоколов обмена данными между локальными уровнями кеш-памяти, которые обеспечивают когерентность кеш-памяти (MSI, MESI, MOSI и т.д.). Протокол MESI - широко используемый в современных процессорах протокол поддержки когерентности кеш-памяти. В соответствии с ним каждая кеш-линейка имеет одно из четырех состояний:

1. Modified
2. Exclusive
3. Shared
4. Invalid

Состояние Modified означает, что кеш-линейка загружена в кеш-память текущего ядра, и данные, хранящиеся внутри нее, изменены.

Кеш-линейка принимает состояние Exclusive, если она присутствует в кеш-памяти только одного ядра и неизменена.

1.1.2. Аппаратная поддержка виртуальной памяти. Кеш-память TLB.

Виртуальная память – это технология управления памятью, которая разработана для многозадачных операционных систем. Для данных каждого процесса, выполняемого на такой системе, выделяется диапазон адресов, называемых виртуальными адресами. Процесс не имеет непосредственного доступа к данным другого процесса.

При обращении к данным в программе, написанной для многозадачной системы, происходит отображение виртуальных адресов в физические. За этот процесс отображения адресов отвечает операционная система. Процесс трансляции виртуального адреса в физический является дорогостоящей по времени операцией. Для его оптимизации соседние виртуальные адреса группируются в виртуальные страницы, размер которых определяется настройками ОС и может варьироваться от нескольких килобайт до нескольких гигабайт. В процессе трансляции виртуального адреса в физический, нужно определить физический адрес виртуальной страницы, в которой он находится, и прибавить к нему смещение относительно начала страница. Таким образом, при обращении к данным, находящимся в одной виртуальной странице происходит только одна операция трансляции виртуального адреса в физический адрес.

В современных вычислительных системах производится кеширование результатов трансляции адресов наиболее часто используемых виртуальных страниц. Устройство, в котором хранятся результаты трансляции адресов виртуальных страниц, называется кеш-памятью TLB (или буфером ассоциативной трансляции). Перед трансляцией адреса виртуальной страницы в физический адрес производится проверка на его наличие в кеш-памяти TLB. Если адрес виртуальной страницы найден в кеш-памяти TLB, то такая ситуация называется попаданием в кеш-память TLB. При попадании в кеш-память TLB трансляция адреса виртуальной страницы не производится. В случае, если искомый адрес виртуальной страницы не найден в кеш-памяти TLB, то возникает ситуация промаха к кеш-памяти TLB, и начинается процесс трансляции виртуального адреса в физический. Ввиду того, что размер кеш-памяти TLB невелик (128-512

записей), физически он часто реализуется в виде полностью ассоциативной кеш-памяти. Особенностью кеш-памяти TLB является то, что при обработке промаха к кеш-памяти TLB процессор не может выполнять другие инструкции и находится в состоянии ожидания. Более того время обработки промаха к кеш-памяти TLB в несколько раз больше чем время обработки промаха к кеш-памяти.

1.2. Увеличение временной локальности данных. Метод тайлинга.

Большинство современных процессоров способны эффективно задействовать кеш-память при выполнении программ, имеющих высокую временную локальность данных [4]. Временная локальность данных заключается в повторном использовании одних и тех же данных за короткий промежуток времени. Если программа имеет высокую временную локальность данных, то данные, к которым производится повторное обращение, не успевают вытесняться из кеш-памяти. В результате количество обращений к основной памяти снижается.

Одним из основных способов увеличения временной локальности данных является метод тайлинга (tiling) [6], [2], [3]. Тайлинг [78] – это преобразование алгоритма для получения макроопераций. Получаемые в результате преобразования макрооперации называются зерном вычислений или тайла. Множество операций алгоритма, составляющих зерно вычислений, выполняется атомарно, как одна единица вычислений.

При использовании тайлинга возникает проблема выбора оптимального размера блока, при котором блочная программа будет работать максимально быстро. Существуют алгоритмы ([7]), которые асимптотически максимально эффективно используют кеш-память без учета точных размеров кеш-памяти и кеш-линеек. Их преимуществом является то, что они легко переносимы на разные вычислительные системы, поддерживающие кеш-память. Тем не менее было показано [8], что такие алгоритмы имеет меньшую производительность чем алгоритмы, в которых параметры кеш-памяти используются явно.

В качестве примера использования тайлинга рассмотрим задачу умножения матриц.

Листинг 1. Стандартный алгоритм умножения матриц

```
for (i=0; i<N; ++i)
  for (k=0; k<N; ++k)
    for (j=0; j<N; ++j)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Результатом применения тайлинга является следующий блок кода:

Листинг 2. Блочное умножение матриц.

```
for (ii = 0; ii < SIZE; ii += BLOCK_SIZE)
for (kk = 0; kk < SIZE; kk += BLOCK_SIZE)
for (jj = 0; jj < SIZE; jj += BLOCK_SIZE)
  for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++)
  for (k = kk; k < kk + BLOCK_SIZE && k < SIZE; k++)
  for (j = jj; j < jj + BLOCK_SIZE && j < SIZE; j++)
    C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Впервые тайлинг был использован в середине 80-х годов [5]. В статье подчеркивается, что использование тайлинга в последние десятилетия стало критически важным для достижения высокой производительности программ линейной алгебры.

Недостатком тайлинга является то, что при его использовании меняется порядок выполнения операций. Поэтому не всегда очевидно, является ли блочная программа эквивалентной исходной.

Количество умножений в стандартном алгоритме умножения квадратных матриц равно n^3, где n – размер матриц. Или $(N/2)^{3/2}$, где N – количество входных данных (количество элементов двух квадратных матрицы размерности n).

Если матрица размера n*n не помещается в кеш-памяти, то в кеш-память попадут строка матрицы A и столбец матрицы B (все, что необходимо для выполнения внутреннего цикла). Над данными, попавшими в кеш-память, программа выполняется частью, вычисляющей скалярные произведения. Процессор недогружен, ожидает поступления данных.

Оценим количество пересылок данных между оперативной памятью и кеш-памятью. Если в кеш-памяти находится вектор-строка матрицы **A,** и к нему подкачиваются векторы-столбцы матрицы **B**, то каждый столбец матрицы **B** будет подкачиваться **n** раз. Итого, чтений из оперативной памяти будет порядка n^3.

Теперь рассмотрим блочный алгоритм умножения квадратных матриц. Сложность по умножениям блочного алгоритма такая же, как и у стандартного n^3. Размер блоков подбирается так, чтобы умножаемые блоки поместились в кеш-памяти. Оценим количество пересылок данных между оперативной памятью и кеш-памятью. Если в кеш-памяти находится блок матрицы **A** и к нему подкачиваются блоки матрицы **B** , то каждый блок матрицы **B** будет подкачиваться **n/m** раз. Итого, чтений блоков матрицы **B** из оперативной памяти будет порядка $\mathbf{(n/m)^3}$. Поскольку каждый блок содержит $\mathbf{m^2}$ чисел, общее количество чтений чисел матрицы **B** из оперативной памяти будет порядка $\mathbf{n^3/m}$ - это в **m** раз меньше, чем при обычном умножении.

В [41] подробно описаны методы подсчета локальности данных, а также алгоритм выбора цепочки преобразований, приводящий к увеличению локальности данных.

Метод разбиения задачи на подзадачи – это метод создания эффективных программ для компьютеров с иерархией памяти. Этот метод можно рассматривать, как обобщение перехода к блочному коду.

Иерархия архитектуры предполагает адекватное ей разбиение задачи на подзадачи – на вычислительном устройстве каждого уровня архитектуры решается подзадача соответствующего уровня иерархии подзадач. Разбиение задач на подзадачи необходимо даже для простой иерархической структуры памяти, состоящей из большой, но медленной, и из быстрой, но малой по объему

памяти. Адекватное разбиение задачи на подзадачи определяет эффективность всей высокопроизводительной системы. Компиляторы и процессоры выполняют такое разбиение автоматически, но, практически, без минимизации обращений к памяти – хорошую оптимизацию программист должен выполнять сам.

1.3. Увеличение пространственной локальности данных. Переразмещение данных

В предыдущем разделе было введено понятие временной локальности данных. Другим важным понятием является пространственная локальность данных [4]. Принцип пространственной локальности данных заключается в частом обращении к данным, хранящимся в одном месте памяти. Как следует из определения пространственной локальности данных, временная локальность данных является ее частным случаем.

1.3.1. Нестандартные размещения данных

В современных языках программирования используются стандартные размещения массивов - по строкам (row-major) либо по столбцам (column-major), которые еще называются каноническими размещениями (Canonical layouts) [10].

Существует также мортоновское размещение матриц. При таком методе размещения матрица разбивается на 4 одинаковых блока. Элементы одного блока хранятся в памяти рядом. Далее каждый блок также разбивается на 4 подблока, элементы которого хранятся рядом. Этот процесс разбиения блоков на подблоки выполняется рекурсивно до тех пор, пока размер разбиваемого блока не станет равен 1. Из определения следует, что мортоновское размещение применимо только к квадратным матрицам, размер которых является степенью 2. На Рисунке 4 представлен пример использования мортоновского размещения для матрицы 8x8. Числа, которыми отмечены элементы матрицы, означают адреса хранения элементов в линейной памяти после размещения.

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

Рис. 4: Мортоновское размещение матрицы 8x8. На месте элементов. матрицы расположен относительный адрес соответствующего элемента.

Мортоновское размещение может использоваться в целях уменьшения количества кеш-промахов [20] для различных приложений, выполняющих много операций над данными. В частности, в приложениях обработки графики [21] и приложениях баз данных [22].

В [9] приводится результаты оптимизации ссылочных структур хранения. Предложены 3 методики оптимизации: clustering (кластеризация), сжатие (compression) и раскраска (coloring). В методе кластеризации используется нестандартное размещение данных, при котором используемые вместе элементы структур находятся в одной кеш-линейке. В частности, в случае кластеризации элементов дерева в одной кеш-линейке стоит хранить поддерево. Если к элементам производится параллельный доступ, то такие элементы размещаются в разных банках кеш-памяти. Использование данных методик позволило ускорить пакеты RADIANCE на 42% и VIS на 27%.

Важность поддержки нестандартных размещений в языках программирования отмечается в [10]. Исследуется производительность 4D- и мортоновского размещений. При использовании нестандартных размещений

добавляются незначительные временные расходы (2-5% от времени работы программы), в то время как скорость выполнения самой программы увеличивается в 1.1-2.5 раза (блочное умножение матриц, алгоритм Штрассена умножения матриц, разложение Холецкого, сжатие изображения с помощью вейвлета Хаара).

В [12] приводятся результаты сравнения блочно-рекурсивного размещения (Morton data layout) данных со стандартными размещениями. Было взято несколько тестовых программ (Alternating Direction Implicit Kernel; умножение матриц; решение двумерной задачи Дирихле уравнения Лапласа методом Якоби; разложение Холецкого). Тестирование проводилось на квадратных матрицах размером от 100x100 до 2048x2048. Результаты экспериментов показали, что блочно-рекурсивное распределение проигрывает стандартным распределениям на архитектуре x86. В случае процессоров Alpha и Sparc блочно-рекурсивное размещение имеет преимущество по сравнению с одним из стандартных размещений.

В [11] авторы ускорили алгоритм, реализующий преобразование Уолша-Адамара. За счет переразмещения данных в оперативной памяти удалось уменьшить количество возникающих кеш-промахов.

В [13] представлена методика *разделения массивов* (array partitioning), при использовании которой кеш-память разделяется на несколько одинаковых по размеру частей. Массивы размещаются таким образом, что они попадают в различные части кеш-памяти.

В статье [35] приводятся результаты ускорения программ на GPGPU за счет использования нестандартных размещений данных. Тестирование проводилось 15 примерах, на которых достигается ускорение в 4.3 раза.

1.3.2. Блочное размещение данных

Блочное размещение – это способ хранения массива в памяти, при котором массив разбивается на блоки одинакового размера. Блоки матрицы хранятся в памяти последовательно без промежутков (Рисунок 5). Элементы, находящиеся

внутри одного блока, хранятся в памяти стандартным образом, например, по строкам:

0	1	2	3	16	17	18	19
4	5	6	7	20	21	22	23
8	9	10	11	24	25	26	27
12	13	14	15	28	29	30	31
32	33	34	35	48	49	50	51
36	37	38	39	52	53	54	55
40	41	42	43	56	57	58	59
44	45	46	47	60	61	62	63

Рис. 5: Блочное размещение элементов матрицы 8x8. На месте элементов матрицы расположен относительный адрес соответствующего элемента.

Общая формула вычисления адреса элемента с индексом (i,j) матрицы X имеет следующий вид:

$$addr(X,i,j) = X + (i/d_1)*\lceil N_2/d_2 \rceil * d_2 + (j/d_2)*d_1*d_2 + (i\%d_1)*d_2 + j\%d_2 \qquad (1)$$

Где

- N2 – количество элементов в строке матрицы X
- d1xd2 – размер блока.

В некоторых блочных алгоритмах блочное размещение массивов дает увеличение производительности [15], [16], [18] , [19], [12], [28], [63]. Так в задаче блочного умножения матриц блочное размещение матриц позволяет существенно

сократить количество промахов к кеш-памяти и кеш-памяти TLB. Подробнее об этом излагается в Главе 2.

В [15] представлены реализации оптимизированных алгоритмов транспонирования матрицы, умножения матриц и генерации полигонов в методе конечных элементов. Данные алгоритмы используют блочные вычисления и блочное размещение данных в совокупности. Тестирование производительности проводилось на процессорах UltraSPARC II, Alpha 21264 и Pentium III. В задаче транспонирования матрицы авторы достигли 50% увеличения производительности, 22% ускорения умножения матриц по сравнению с алгоритмами CBLAS. В блочном алгоритме генерации полиэдров в методе конечных элементов блочное размещение дает 1.7 раза по сравнению со стандартным размещением.

В [16] исследовано влияние нестандартных видов размещения данных, таких как блочное и мортоновское размещения, на эффективность использования кеш-памяти и кеш-памяти TLB в блочных программах. Представлены методики определения оптимального размера блока, используемого при размещении матриц блочным образом. Эффективность данной методики была проверена на задачах LU-разложения матрицы, разложении Холецкого матрицы и умножения матриц. Результаты численных экспериментов показали, что блочное размещение в совокупности с блочными вычислениями дает дополнительную производительность более чем на 50% по сравнению со стандартными размещениями.

В [23] рассматривается вопрос оптимизации алгоритма двумерного быстрого преобразования Фурье. Авторы статьи исследовали зависимость времени работы алгоритма от метода размещения данных. Результаты численных экспериментов показали, что при использовании блочного размещения время работы алгоритма меньше чем при стандартном размещении. Это связано с тем, что в данной задаче вычисление преобразования Фурье для столбцов матрицы, размещенной по строкам, является дорогостоящей операцией, в силу ассоциативности кеш-памяти.

В некоторых задачах выбор способа размещения данных сильно влияет на эффективность векторизации. Так, в алгоритме умножения матриц в библиотеках GotoBLAS [86] и OpenBlas [30] для реализации эффективной векторизации (register blocking) используется блочное размещение матриц. В работе [46], [47], [48] предложены специфические способы распределения массивов для трафаретных программ (stencil codes). Результаты численных экспериментов показали, что предложенные способы размещения массивов позволяют увеличить эффективность результирующей программы (10-200% в зависимости от вычислительной архитектуры и задачи).

В [14] представлен высокопроизводительный блочный алгоритм переразмещения матриц из стандартного формата в блочный формат и наоборот. Особенностью данного алгоритма является то, что он не требует большого количества дополнительной памяти, и поэтому он может использоваться для переразмещения матриц большого размера.

Изложенные выше методы оптимизации увеличивают эффективность использования памяти за счет переразмещения элементов в рамках каждого из массивов. В случае, когда в программе производится обращение к элементам нескольких массивов, то может возникнуть ситуация, при которой запрашиваемые элементы разных массивов будут отображаться в одни и те же банки памяти, что снизит эффективность работы кеш-памяти. Эта проблема может быть решена, если разместить массивы так, чтобы запрашиваемые в одно и то же время элементы разных массивов хранились в памяти близко друг относительно друга. В [13] приводятся результаты численных экспериментов, которые подтверждают, что подобное размещение массивов может дать дополнительную производительность.

1.4. Выравнивание данных в памяти

При разработке программ следует помнить о том, что данные должны быть выровнены в памяти. Это требование связано с тем, что процессоры

обрабатывают выровненные данные намного эффективнее, чем невыровненные. Некоторые процессоры [60] при невыровненном доступе к данным аварийно прекращают выполнение программы.

Для того чтобы переменная, занимающая несколько байт памяти, была выровнена, необходимо, чтобы ее адрес был кратен ее размеру. Так, например, если объявленная в программе переменная имеет тип int32 (4 байта), то для того, чтобы она была выровнена, она должна быть размещена по адресу кратному 4.

Если процессор поддерживает невыровненный доступ, то может возникнуть ситуация, когда невыровненное данное находится в соседних словах. В этом случае процессору придется считывать из памяти соседние слова, в которых находится данное, а также произвести “склейку” этих частей. Таким образом, при доступе к невыровненным данным производятся дополнительные арифметические операции чтения данных. Невыровненный доступ порождает дополнительные кеш-промахи в случае, когда считываемая переменная хранится на границе кеш-линеек или на границе виртуальных страниц.

Современные компиляторы способны выравнивать данные автоматически. Поэтому программисту в большинстве случаев не приходится думать о выравнивании данных. Рассмотрим примеры того, как происходит выравнивание данных компиляторами.

Пример 1**.** Для того, чтобы в структуре

Листинг 3. Объявление структуры, в которой размер первого поля меньше размера второго поля.

```
struct Foo1
{
  short a;
  int64 b;
}
```

поле b было выровнено, компилятор после размещения поля ‘a’ вставляет дополнительные 6 байт. Поэтому размер такой структуры равен 16 байт, а не 10.

Пример 2. Рассмотрим следующую структуру:

Листинг 4. Объявление структуры, в которой размер первого поля больше размера второго поля.

```
struct Foo2
{
  int64 b;
  short a;
}
```

В данной структуре размер поля b больше размера поля a. Поэтому компилятор размещает поле a сразу после поля b без добавления промежуточных байт. После байт, относящихся к полю ‘a’ компилятор вставляет дополнительные 6 байт, чтобы размер структуры был кратен размеру максимального типа, используемого при объявлении полей этой структуры. Дополнительные байты вставляются в конец структур для того, чтобы обеспечить автоматическое выравнивание данных внутри массивов структур.

В некоторых компиляторах реализована возможность ручного контроля над выравниванием полей структур. Так, например, в компиляторах C/C++ MSVC поддерживаются директивы **pragma pack push/pop**.

1.5. Оптимизация работы с данными программ современными компиляторами

Система компиляторов ROSE [44], [89], [88] предназначена для построения source-to-source трансляторов. В качестве входных языков поддерживаются Fortran 2003, C, C++. Система ROSE имеет высокоуровневое внутреннее

представление, а также большой набор высокоуровневых оптимизирующих преобразований (оптимизации гнезд циклов, inlining, Partial Redundancy Elimination).

В языке Fortran DVM [39] реализована поддержка ручного размещения данных в распределенной памяти. Для эффективной работы программы, работающей на многопроцессорной системе, программист должен правильно разместить данные в оперативной памяти каждого из узлов системы.

Автоматическое выравнивание данных реализовано во всех компиляторах, поддерживающих автоматическую векторизацию (GCC [85], Intel [38], LLVM [83], MSVC [104]). В некоторых компиляторах семейства PGI [45] реализованы операции копирования массивов (copyin) на графический ускоритель и выгрузка результатов с ускорителя в оперативную память (copy).

Существуют оптимизаторы кода Polly LLVM [81], [82] и PLUTO [87], [43], которые ускоряют программы за счет автоматического применения метода тайлинга и распараллеливания программы для многоядерных процессоров.

Массивы данных в современных языках программирования могут быть определены как массивы структур, структуры массивов либо как комбинация этих подходов. Выбор конкретного способа определения массивов влияет на эффективность использования памяти, а также на операции доступа к данным. Соответственно, при внесении изменений в метод размещения массивов, программисту приходится изменять все операции доступа к элементам массива, что является достаточно трудоемким процессом.

Проект TALC [33], [34] предназначен для того, чтобы избавить программиста от необходимости переписывать программу, в случае изменения метода размещения массивов. TALC входит в систему компиляторов ROSE и представляет собой расширение языков C/C++. С помощью него программист посредством специальных *схем данных* может определять массивы структур, а также способ размещения полей внутри структур. Поля внутри структур могут также быть массивами. По схемам данных TALC генерирует соответствующие им структуры данных языка Си. Также генерируются операции доступа к этим

структурам внутри программы. Тестирование производительности TALC велось на нескольких программах решения задач гидродинамики и продемонстрировало ускорение в 40-60% [33].

В работе [36] отмечается важность использования переразмещения данных для увеличения уровня SLP (Superword Level Parallelism). При правильном выборе метода размещения данных уменьшается количество дополнительных операций упаковки/распаковки данных. С этой целью в компиляторе SUIF [42], [40] была реализована автоматизация переразмещения переменных. В результате применения оптимизатора на 16 программах было получено среднее ускорение в 15.2% за счет увеличения SLP.

1.6. Выводы к первой главе

В первой главе описаны понятия необходимые для изложения результатов диссертации. В **параграфе 1.1** приведены принципы работы иерархии памяти современных вычислительных систем. Дается определение кеш-памяти, виртуальной памяти, кеш-памяти TLB. **Параграф 1.2** посвящен методу тайлинга, который предназначен для оптимизации временной локальности данных. В **параграфе 1.3** приведено определение пространственной локальности данных, а также различные методы размещения данных. В частности, описывается блочное размещение данных, автоматическую поддержку которого реализовал автор диссертации в системе OPC. В **параграфе 1.4** приводится понятие выравнивания данных в памяти. На примерах представлено как производится выравнивание данных современными компиляторами. **Параграф 1.5** содержит обзор компиляторов, которые содержат в своем составе нестандартные оптимизации работы с памятью. В частности, приводится описание проекта TALC, который позволяет программисту размещать структуры и массивы согласно определенной схемы.

Глава 2

Модель времени выполнения программ для иерархической памяти

Скорость доступа к данным, находящимся в памяти, для многих задач остается слабым местом в работе компьютера: процессоры обрабатывают данные намного быстрее, чем подкачивают их из памяти. Память современных процессоров имеет иерархическую структуру. Чаще всего она состоит из регистрового файла процессора, иерархической кеш-памяти и оперативной памяти. Данные между оперативной памятью и кеш-памятью пересылаются блоками (кеш-линейками) фиксированной длины. В некоторых моделях процессоров Intel [58], [59] реализована трехуровневая кеш-память (Рисунок 6). Замыкает иерархию оперативная память.

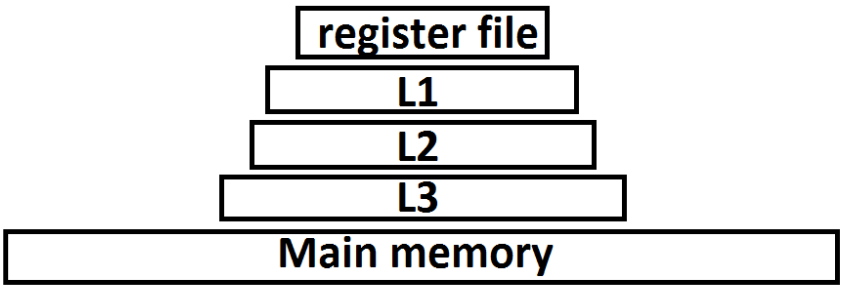

Рис. 6: Иерархия памяти процессоров Intel Sandy Bridge, IvyBridge, Haswel [58, 59]

В то же самое время современные процессоры имеют большое количество параметров. Например, в Таблице 2 приведены некоторые важные параметры процессора Intel Core i5-2410m.

Таблица 2. Параметры процессора Intel Core i5-2410m

Cache line size	64 bytes
L1 size	32 KB
L1 banks count	64
L1 associativity	8
L1 latency	3 ops
L2 size	256 KB
L2 banks count	512
L2 associativity	8
L2 latency	10 ops
L3 size	3072 KB
L3 banks count	4096
L3 associativity	12
L3 latency	22 ops
TLB cache size	512 items
TLB cache associativity	4
CPU frequency	2.3 GHz

Каждый из данных параметров должен учитываться во время оптимизации программы. Общая формула времени выполнения программы имеет следующий вид:

$$T(n) = T_{execution}(n) + T_{load}(n) \quad (2)$$

где n – размер входных данных, *T(n)* – время выполнения программы, $T_{execution}(n)$ - общее время выполнения арифметических операций, $T_{load}(n)$ - общее время перекачки данных между оперативной памятью и кеш-памятью. Точная формула времени выполнения зависит от алгоритма.

Пусть исходный алгоритм можно представить в виде нескольких подзадач. В этом случае формула времени выполнения алгоритма будет иметь следующий вид:

$$T(n) = R \cdot (T_{task_{execution}} + T_{task_{load}}) \quad (3)$$

Где $T_{task_{load}}$ — время подкачки данных, используемых подзадачей, из оперативной памяти в кеш, $T_{task_{execution}}$— время выполнения подзадачи, R – количество подзадач. Предположим, что функция сложности алгоритмов решения исходной задачи и подзадач одинаковая и равна F (количество операций, как функция от количества входных данных), объем данных исходной задачи D1 и объем данных всех подзадач одинаков и равен D2. В этом случае количество подзадач

$$R = \frac{F(D1)}{F(D2)} \quad (4)$$

Пусть подзадачи запрограммированы оптимально ($T_{task_{execution}}$ минимально). Для того чтобы уменьшить время выполнения алгоритма нужно уменьшить $T_{task_{load}}$.

Рассмотрим подзадачу в виде совокупности подзадач меньшего размера. В этом случае к ней также будет применима формула (3). Сведение задачи к нескольким подзадачам выполняется рекурсивно. Согласно модели, оптимальное количество шагов рекурсии равно количеству уровней памяти, при условии, что исходную задачу можно разбить на подзадачи таким образом. Ниже представлено несколько примеров использования формулы (3). Для упрощения будем предполагать, что кеш данных является полностью ассоциативным.

Пример 3. *Вычисление формулы времени выполнения для задачи однородного доступа к памяти.* Рассмотрим случай, когда в качестве задачи выступает чтение из памяти элементов массива X с i1 по i2 с шагом h. Предполагается, что

- Данные выровнены по типу данных, т.е. адрес данного кратен размеру типа этого данного.
- Размер типа данного меньше размера кеш-линейки.
- Первый элемент массива, к которому производится обращение, смещен относительно начала кеш-линейки на p0 байт, а относительно начала виртуальной страницы — на p1 байт.

Подсчитаем количество кеш-линеек S0 загруженных из оперативной памяти в кеш: Если $h < \left\lceil \frac{Lk}{d} \right\rceil$, то

$$S0 = \left\lceil \frac{d \cdot (i2 - i1 + 1) + po}{Lk} \right\rceil \tag{5}$$

иначе

$$S0 = \left\lceil \frac{i2 - i1 + 1}{h} \right\rceil$$

где Lk — размер кеш-линейки, d — размер типа данных. Значение выражения $\lceil x \rceil$ равно минимальному целому числу большему или равному чем x (соответственно, значение выражения $\lfloor x \rfloor$ равно максимальному целому числу меньшему или равному чем x). Количество виртуальных страниц S1, задействованных в данной задаче, вычисляется схожим образом. Если $h < \left\lceil \frac{Vk}{d} \right\rceil$, то

$$S1 = \left\lceil \frac{d \cdot (i2 - i1 + 1) + p1}{Vk} \right\rceil \tag{6}$$

иначе

$$S1 = \left\lceil \frac{i2 - i1 + 1}{h} \right\rceil$$

Здесь *Vk* – размер виртуальной страницы. Таким образом,

$$T_{task_{load}} = k0 \cdot S0 + k1 \cdot S1,$$

где *k0* – время перекачки кеш-линейки из оперативной памяти в кеш-память, а *k1* – время трансляции виртуальной страницы. Данный пример демонстрирует, что для того, чтобы данные считывались максимально быстро из памяти, они должны храниться рядом друг с другом.

Пример 4. Эффективное разбиение задачи умно*жения матриц на подзадачи. Использование блочного кода и блочного размещения данных.* Пусть решается задача умножения матриц C=A*B. Будем предполагать, что матрицы – квадратные, размер матрицы равен N, каждая из них не помещается в кеш-памяти. Рассмотрим стандартный алгоритм умножения матриц:

Листинг 5. Стандартный алгоритм умножения матриц.

```
for(i=0; i<N; ++i)
for(j=0; j<N; ++j)
for(k=0; k<N; ++k)
  C[i*N+j]+=A[i*N+k]*B[k*N+j];
```

В данном алгоритме общее время подкачки данных T зависит в большей степени от операций, проводимых над матрицей B. Это связано с тем, что доступ к элементам B происходит с шагом N. Поэтому время выполнения алгоритма равно

$$T_1(n) = k_0 \cdot N^3 + k_1 \cdot N^3 + k_2 \cdot N^3 \qquad (7)$$

где *k0* – время перекачки кеш-линейки из оперативной памяти в кеш-память, *k1* – время трансляции виртуальной страницы, *k2* – время выполнения арифметической операции.

Время работы алгоритма (7) можно уменьшить, если представить исходную задачу в виде набора подзадач – умножений блоков матриц. Для этого модифицируем исходный алгоритм умножения матриц, используя тайлинг:

Листинг 6. Алгоритм умножения матриц после применения к нему метода тайлинга.

```
for(di=0; di<N; di+=d)
for(dj=0; dj<N; dj+=d)
for(dk=0; dk<N; dk+=d)
  for(i=di; i< MIN(N,di*d); ++i)
  for(j=dj; j< MIN(N,dj*d); ++j)
  for(k=dk; k< MIN(N,dk*d); ++k)
    C[i*N+j]+=A[i*N+k]*B[k*N+j];
```

Время работы данного алгоритма равно

$$T_2(n) = \left(\frac{N}{d}\right)^3 \cdot (k_0 \cdot d \cdot \left\lceil \frac{d}{Lk} \right\rceil + k_1 \cdot d \cdot \left\lceil \frac{d}{Vk} \right\rceil + k_2 \cdot d^3) \quad (8)$$

Где *Lk* — размер кеш-линейки, *VK* – размер виртуальной страницы *d* — размер типа данных. Предположим, что *d* нацело делится на *LK*, и *d* < *VK*. Тогда

$$T_2(n) = N^3 \cdot (\frac{k_0}{d \cdot Lk} + \frac{k_1}{d^2} + k_2) \quad (9)$$

Разделим $T_1(n)$ на $T_2(n)$ и получим ускорение, полученное за счет разбиения на подзадачи:

$$\frac{T_1(n)}{T_2(n)} = \frac{k_0 + k_1 + k_2}{\frac{k_0}{d \cdot Lk} + \frac{k_1}{d^2} + k_2} \quad (10)$$

Скорость доступа к данным можно увеличить еще больше, если матрицы разместить блочно, выбрав размер блока равным d. В этом случае формула (8) примет вид

$$T_3(n) = \left(\frac{N}{d}\right)^3 \cdot (k_0 \cdot \frac{d^2}{Lk} + k_1 \cdot \frac{d^2}{Vk} + k_2 \cdot d^3) \quad (11)$$

Как видно из формул (8) и (11) имеет место следующая теорема:

Теорема. Применение блочного размещения матриц к блочному алгоритму позволяет уменьшить количество промахов к кеш-памяти в $\frac{\left\lceil \frac{d}{Lk} \right\rceil}{\frac{d}{Lk}}$ раз, а к кеш-памяти TLB – в $\frac{\left\lceil \frac{d}{Vk} \right\rceil}{\frac{d}{Vk}}$ раз.

2.1. Анализ блочного алгоритма умножения матриц

Рассмотрим задачу определения оптимального уровня кеш-памяти для которого следует производить тайлинг.

2.1.1. Определение уровня кеш-памяти, для которого оптимально производить тайлинг

Будем предполгалать, что кеш-память каждого уровня является полностью ассоциативной. Формула времени выполнения (2) блочного алгоритма умножения матриц имеет следующий вид:

$$T(n, L) = T_{execution}(n) + T_{load}(n, L) \quad (12)$$

Где L – уровень кеш-памяти под который производится тайлинг, $T_{execution}(n)$ - время выполнения всех арифметических операций, $T_{load}(n, L)$ - время загрузки используемых данных в уровень кеш-памяти L. Время подкачки данных выражается через время подкачки одного блока следующим образом:

$$T_{load}(n,L)=\left(\frac{n}{d_L}\right)^3 * T_{load}(d_L) \quad (13)$$

Если кеш-память каждого уровня является полностью ассоциативной, то

$$d_L = \sqrt{\frac{LSize}{DataTypeSize}} \quad (14)$$

.

Это связано с тем, что в кеш-память должен помещаться только блок матрицы B.

Теорема: Отношение времен загрузки данных в уровни кеш-памяти обратнопропорционально корню квадратному отношения размеров уровней кеш-памяти и прямопропорционально отношению латентностей уровней кеш-памяти:

$$\frac{T_{load}(n,L_1)}{T_{load}(n,L_2)} = \frac{\left(\frac{n}{d_1}\right)^3 * \left(\frac{L_1 Size}{DataTypeSize} * \lambda_1\right)}{\left(\frac{n}{d_2}\right)^3 * \left(\frac{L_2 Size}{DataTypeSize} * \lambda_2\right)} = \left(\frac{L_2 Size}{L_1 Size}\right)^{\frac{1}{2}} * \frac{\lambda_1}{\lambda_2} \quad (15)$$

Параметр λ_1 определяет время закачки данных из кеш-памяти L1 в регистры процессора. Параметр λ_2 соответственно определяет время закачки данных из кеш-памяти L2 в регистры. Найдем теперь уровень кеш-памяти, для которого время загрузки данных будет минимально при использовании тайлинга:

$$\frac{T_{load}(n,L_2)}{T_{load}(n,L_3)} = \left(\frac{3*2^{10}}{2^8}\right)^{\frac{1}{2}} * \frac{10}{22} = \sqrt{12} * \frac{10}{22} > 1$$

$$\frac{T_{load}(n,L_1)}{T_{load}(n,L_3)} = \left(\frac{3*2^{10}}{2^5}\right)^{\frac{1}{2}} * \frac{3}{22} = 12\sqrt{2} * \frac{3}{22} > 1$$

$$\frac{T_{load}(n,L_1)}{T_{load}(n,L_2)} = \left(\frac{2^8}{2^5}\right)^{\frac{1}{2}} * \frac{3}{10} = \sqrt{2^3} * \frac{3}{10} < 1$$

$$T_{load}(n,L_3) < T_{load}(n,L_1) < T_{load}(n,L_2)$$

Таким образом, минимальное общее время закачки данных достигается при тайлинге, проводимом для кеш-памяти L3. Следующий уровень кеш-памяти, для которого следует производить тайлинг, является кеш-память L1.

Рассмотрим теперь задачу определения оптимального размера блока для конкретного уровня кеш-памяти.

2.1.2. Определение оптимального размера блока для блочного алгоритма умножения матриц

Предположим, что тайлинг производится под кеш-память *L3*. В этом случае, если кеш-память *L3* является полностью ассоциативной, то оптимальный размер блока равен

$$d_L = \sqrt{\frac{L_3 Size}{DataTypeSize}} \tag{16}$$

Рассмотрим теперь случай, когда кеш-память *L3* не является полностью ассоциативной. Пусть *step* – минимальное положительное число, такое что

$$(step \cdot N \cdot DataTypeSize) \vdots (L_3 BankCount \cdot CacheLineSize) \tag{17}$$

В этом случае элементы каждой строки матрицы с номером кратным числу *step* будут отображаться в одни и те же кеш-линейки.

Если $N = 2048$, то $step = 16$ и при размере блока $d > L_3 Assoc * step = 12 \cdot 16 = 192$ элементы одного блока уже начнут вытеснять друг друга из кеш-памяти, и тем самым существенно увеличится количество кеш-промахов. Таким образом, оптимальный размер блока $d \le 192$.

Для блочного умножения матриц со стандартным размещением данных имеет место следующая формула времени выполнения:

Если $d \le 192$, то

$$T(n) = \left(\frac{n}{d}\right)^3 \cdot \frac{1}{\nu}\left(\frac{d^2 \cdot 3 \cdot memory\_latency}{\text{CacheLineSize}} + \frac{d^3 \cdot cache\_latency}{\text{CacheLineSize}} + d^3 \cdot 2 \cdot k\right) \quad (18)$$

, иначе

$$T(n) = \frac{n^3}{\nu}\left(\frac{memory\_latency}{\text{CacheLineSize}} + \frac{cache\_latency}{\text{CacheLineSize}} + 2 \cdot k\right) \quad (19)$$

Где υ - частота процессора, k - время выполнения арифметических операций. Точность данной формулы подтверждается результатами численного эксперимента. В данном эксперименте подсчитывается время выполнения программы, реализующей блочное умножение матриц для разных размеров блока:

Таблица 3. Результаты профилировки блочного умножения матриц

N	**d**	**LL3 cache miss impact**	**L2 cache miss impact**	**L1 cache miss impact**	**Elapsed (sec)**	**Theoretical time (sec)**
2048	64	0.71	3.6	0.014	64.443805	27.39792
2048	96	0.43	1.9	0.69	63.763994	26.6684
2048	**128**	0.51	1.8	0.6	**63.267705**	26.3037
2048	160	0.61	1.4	0.68	63.409075	26.0849
2048	**192**	2.3	0.67	0.94	65.798332	**25.9390**
2048	224	4.1	0.25	0.86	69.210627	71.894

Данный эксперимент, а также последующие эксперименты данной главы проведены с помощью инструмента Intel Vtune Amplifier [
53]. На базе собранных с помощью него профилей работы программ были рассчитаны следующие характеристики программы:

1. Влияние кеш-промахов к L1-кешу.

$$L1\ miss\ impact = \frac{12 * \text{MEM\_LOAD\_UOPS\_RETIRED.L2\_HIT}}{\text{CPU\_CLK\_UNHALTED.THREAD}}$$

2. Влияние кеш-промахов к L2-кешу.

$$L2\ miss\ impact = \frac{\begin{matrix}((26 * \text{MEM\_LOAD\_UPOS\_RETIRED.LLC\_HIT\_PS}) \\ + (43 * \text{MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRED.XSNP\_HIT\_PS}) \\ + (60 * \text{MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRED.XSNP\_HITM\_PS}))\end{matrix}}{\text{CPU\_CLK\_UNHALTED.THREAD}}$$

3. Влияние кеш-промахов к L3-кешу.

$$L3\ miss\ impact = \frac{180 * \text{MEM\_LOAD\_UOPS\_MISC\_RETIRED.LLC\_MISS\_PS}}{\text{CPU\_CLK\_UNHALTED.THREAD}}$$

Каждая из характеристик показывает относительное количество кеш-промахов к определенному уровню кеш-памяти, произошедших во время выполнения программы. Чем больше это значение, тем больше влияние кеш-промахов к конкретному уровню кеш-памяти на общее падение производительности программы.

Из эксперимента видно, что оптимальный размер блока равен 128. В Таблице 3 приведено также теоретическое время работы программы, которое вычисляется с использованием формул 18-19. Теоретическое значение размера блока близко к практическому значению и равно 192.

Для того чтобы уменьшить влияние ассоциативности кеш-памяти на размер блока, разместим матрицы блочно. В Таблице 4 приведены результаты численного эксперимента, в котором замерялось время работы блочного алгоритма умножения матриц, размещенных блочно.

Таблица 4. Результаты профилировки блочного умножения матриц с блочным размещением

N	**d**	**LL3 cache miss impact**	**L2 cache miss impact**	**L1 cache miss impact**	**Elapsed (sec)**	**Theoretical time (sec)**
2048	64	0.16	0.027	1.4	53.200717	27.3979
2048	256	0.25	0.37	0.85	48.364546	25.7566
2048	384	0.18	0.43	0.71	47.757125	25.5743
2048	**512**	0.29	0.38	0.75	**47.244979**	25.4831
2048	568	0.52	0.38	0.74	48.283772	25.4561
2048	**624**	1.3	0.26	0.7	49.350072	**25.4340**
2048	768	2.6	0.09	0.73	49.797658	71.8940
2048	1024	2.5	0.023	0.34	50.419416	71.8940

Из эксперимента видно, что оптимальный размер блока равен 512. Оптимальное теоретическое значение размера блока близко к практическому значению и равно 624.

2.2. Определение оптимального размера блока для блочного алгоритма Флойда-Уоршалла

В данном разделе будут приведены оценки времени выполнения для алгоритма Флойда-Уоршалла поиска кратчайших расстояний между вершинами графа. Стандартная реализация алгоритма Флойда-Уоршалла, представлена ниже:

Листинг 7. Стандартный алгоритм Флойда-Уоршалла.

```
for (int k = 0; k < N; k++)
for (int i = 0; i < MIN(dl+d, N); i++)
for (int j = 0; j < MIN(dl+d, N); j++)
   A[i*N+j] = min(A[i*N+j], A[i*N+k]+A[k*N+j]);
```

Ниже приведена блочная реализация алгоритма Флойда-Уоршалла:

Листинг 8. Блочной алгоритм Флойда-Уоршалла.

```
for (dl = 0; dl < N; dl += d) {
   for (k = dl; k < MIN(dl+d, N); k++)
   for (i = dl; i < MIN(dl+d, N); i++)
   for (j = dl; j < MIN(dl+d, N); j++)
      A[i*N+j] = min(A[i*N+j], A[i*N+k]+A[k*N+j]);
   for (dk = 0; dk < N; dk += d) {
     if (dk == dl)
       continue;
     for (k = dl; k < MIN(dl+d, N); k++)
     for (i = dl; i < MIN(dl+d, N); i++)
     for (j = dk; j < MIN(dk+d, N); j++)
        A[i*N+j] = min(A[i*N+j], A[i*N+k]+A[k*N+j]);
     for (k = dl; k < MIN(dl+d, N); k++)
     for (i = dk; i < MIN(dk+d, N); i++)
     for (j = dl; j < MIN(dl+d, N); j++)
        A[i*N+j] = min(A[i*N+j], A[i*N+k]+A[k*N+j]);
   }
   for (dk = 0; dk < N; dk += d)
   for (ds = 0; ds < N; ds += d) {
     if (dk == dl || ds == dl) continue;
     for (k = dl; k < MIN(dl+d, N); k++)
     for (i = dk; i < MIN(dk+d, N); i++)
     for (j = ds; j < MIN(ds+d, N); j++)
        A[i*N+j] = min(A[i*N+j], A[i*N+k]+A[k*N+j]);
   }
}
```

Предположим, что тайлинг производится под *L3*-кеш. В этом случае, если *L3*-кеш является полностью ассоциативным, то оптимальный размер блока равен

$$d_L = \sqrt{\frac{L_3 Size}{DataTypeSize}} \tag{20}$$

Рассмотрим теперь случай, когда *L3*-кеш не является полностью ассоциативным. Пусть *step* – минимальное положительное число, такое что

$$(step \cdot N \cdot DataTypeSize) \vdots (L_3 BankCount \cdot CacheLineSize) \tag{21}$$

В этом случае элементы каждой строки матрицы с номером кратным числу *step* будут отображаться в одни и те же кеш-линейки.

Если $N = 2048$, то $step = 16$ и при размере блока $d > L_3 Assoc * step = 12 \cdot 16 = 192$ элементы одного блока уже начнут вытеснять друг друга из кеш-памяти, и тем самым существенно увеличится количество кеш-промахов. Таким образом, оптимальный размер блока $d \le 192$.

Для блочного алгоритма Флойда со стандартным размещением данных имеет место следующая формула времени выполнения:

Если $d \le 192$, то

$$T(n) = \left(\frac{n}{d}\right)^3 \cdot \frac{1}{\nu}\left(\frac{d^2 \cdot 3 \cdot memory\_latency}{\text{CacheLineSize}} + \frac{d^3 \cdot cache\_latency}{\text{CacheLineSize}} + d^3 \cdot 2 \cdot k\right) \tag{22}$$

, иначе

$$T(n) = \frac{n^3}{\nu}\left(\frac{memory\_latency}{\text{CacheLineSize}} + \frac{cache\_latency}{\text{CacheLineSize}} + 2 \cdot k\right) \tag{23}$$

Где υ - частота процессора, k - время выполнения арифметических операций. Точность формул 22-23 подтверждается результатами численного

эксперимента. В эксперименте подсчитывается время выполнения программы, реализующей блочный алгоритм Флойда-Уоршалла для разных размеров блока:

Таблица 5. Результаты профилировки блочного алгоритма Флойда-Уоршалла

N	**d**	**LL3 cache miss impact**	**L2 cache miss impact**	**L1 cache miss impact**	**Elapsed (sec)**	**Theoretical time (sec)**
2048	64	1.2	6.9	0.22	47.919996	27.39792
2048	96	1.0	5.2	0.44	46.652864	26.6684
2048	128	0.92	4.4	0.7	45.936372	26.30375
2048	**160**	0.92	3.1	0.95	**45.513963**	26.08492
2048	176	3.0	2.5	0.79	47.382814	26.00534
2048	**192**	6.0	1.7	1.0	49.601719	**25.93903**
2048	256	11.0	0.41	1.2	50.248607	71.89401

Из эксперимента видно, что оптимальный размер блока равен 160. В Таблице 5 приведено также теоретическое время работы программы, которое вычисляется с использованием формул 22-23. Теоретическое значение размера блока близко к практическому значению и равно 192.

Так же, как и в эксперименте, посвященному блочному умножению матриц, для того чтобы уменьшить влияние ассоциативности кеш-памяти на размер блока, разместим матрицу блочно. В Таблице 6 приведены результаты численного эксперимента, в котором замерялось время работы блочного алгоритма Флойда-Уоршалла с блочным размещением.

Таблица 6. Результаты профилировки блочного алгоритма Флойда-Уоршалла с блочным размещением

N	d	LL3 cache miss impact	L2 cache miss impact	L1 cache miss impact	Elapsed (sec)	Theoretical time (sec)
2048	272	0.26	1.1	0.59	43.762151	25.72449
2048	336	0.28	0.85	0.61	43.48941	25.62641
2048	400	0.31	0.87	0.51	43.375812	25.55972
2048	496	0.44	0.84	0.47	43.320609	25.49195
2048	**528**	0.53	0.68	0.5	**43.144292**	25.47484
2048	560	0.88	0.68	0.42	43.352389	25.45968
2048	**624**	2.7	0.48	0.41	44.037547	**25.43403**
2048	688	3.9	0.1	0.56	44.883257	71.89401

Из эксперимента видно, что оптимальный размер блока равен 528. Теоретическое оптимальное значение размера блока близко к практическому значению и равно 624.

2.3. Анализ масштабируемости параллельного блочного алгоритма Флойда

Важной характеристикой параллельного алгоритма является его масштабируемость. Параллельный алгоритм является масштабируемым, если при росте количества ядер он обеспечивает увеличение ускорения при сохрании эффективности использования ядер. Эффективность использования вычислительного ядра определяется соотношением $S_P = \frac{T(1,N)}{P \cdot T(P,N)}$, где $T(p,N)$ - время работы последовательной программы, запущенной на процессоре с количеством ядер равным p.

Согласно формуле предлагаемой модели времени выполнения (3):

$$S_P = \frac{T_{execution}(1,N) + T_{load}(1,N)}{P \cdot (T_{execution}(P,N) + T_{load}(P,N))}$$

Рассмотрим, чему равны значения $T_{execution}(p,N)$ и $T_{load}(p,N)$ в случае параллельного блочного алгоритма Флойда. Время выполнения арифметических операций равно $T_{execution}(p,N) = \frac{2 \cdot N^3}{p \cdot \nu}$, где ν – это частота одного ядра. Время загрузки определяется следующей формулой: $T_{load}(p,N) = \frac{N}{d(p)} \cdot \frac{\left(\frac{N}{d(p)}\right)^2}{p\mu} \cdot p \cdot 3d(p)^2 = \frac{3N^3}{d(p) \cdot \mu}$, где

- $d(p)$- это оптимальный размер блока, при котором данные, используемые различными ядрами, не вытесняются из общей кеш-памяти. Если предположить, что кеш-память является полностью ассоциативной, то оптимальный размер блока определяется следующим образом: $d(p) = \sqrt{\frac{CacheSize}{3 \cdot p \cdot sizeof\,(double)}} = \frac{1}{2}\sqrt{\frac{CacheSize}{6p}}$
- μ - частота подкачки данных из оперативной памяти в кеш-память.

Удобно определить величину $\gamma = \frac{\nu}{\mu}$, которая показывает во сколько раз быстрее одно вычислительное ядро обрабатывает данные чем их подкачивает.

Из приведенный соотношений следует, что

- начиная с $p = \left(\frac{\sqrt{CacheSize}}{3\sqrt{6}\gamma}\right)^{\frac{2}{3}}$ время загрузки данных будет занимать больше времени чем время выполнения арифметических операций.
- Минимальное теоретическое время работы алгоритма достигается при $P_{opt} = \left(\frac{\sqrt{2 \cdot CacheSize}}{3\sqrt{3}\gamma}\right)^{\frac{2}{3}}$.
- Эффективность использования ядер монотонно убывает, начиная с $p = 1$.

2.4. Статическое определение количества кеш-промахов

Как было сказало ранее, время обращения к данным является намного большим, чем время выполнения арифметических операций. Поэтому программист при реализации программы должен представлять, сколько обращений к памяти производит его программа во время своего выполнения. Ситуация усложняется тем, что одна и та же программа на разных компьютерах может производить различное количество обращений к памяти. Это связано с тем, что существуют компьютеры с различными параметрами кеш-памяти (ассоциативность, размер, количество уровней, политика замещения элементов), данная задача становится очень трудоемкой.

Информация по количеству кеш-промахов, возникших за время работы программы, хранится в специальных регистрах – счетчиках производительности процессора (hardware performance counters). На основе данных счетчиков работают несколько известных динамических профилировщиков [
53], [54], позволяющих подсчитать количество кеш-промахов, возникающих в программе. Они основаны на том, что входная программа запускается в специальном режиме на целевом компьютере, и во время ее работы берутся замеры счетчиков производительности.

Основным недостатком такого подхода является то, что для анализа программы необходимо ее запустить (собственно, поэтому такие профилировщики называются динамическими). Некоторые приложения требуют много времени для своей работы. И поэтому профилировка таких программ становится очень долгой. Другим недостатком является зависимость от платформы, на которой производится запуск программы. В качестве исключения можно привести продукт Valgrind [55], позволяющий эмулировать требуемый процессор и выполнять программу на нем.

Последним важным недостатком динамической профилировки является то, что невозможно за один проход получить оценку количества кеш-промахов как функцию от входных данных, т.к. за один проход можно получить количество

кеш-промахов только для конкретного набора входных данных. Поэтому требуется запускать программу на разных наборах входных данных.

В данном разделе приводятся методы статического определения количества кеш-промахов. Изложенные методы одинаково работают как для иерархии уровней кеш-памяти, так и для кеш-памяти TLB. Поэтому с их помощью возможно получить численные оценки количества обращений к основной памяти, количество промахов к промежуточным уровням кеш-памяти, количество промахов к виртуальным страницам. Запуск исследуемых программ не требуется.

Рассмотрим задачу определения количества промахов к уровню кеш-памяти верхнего уровня. Будем предполагать, что политикой замещения элементов в кеш-памяти является политика LRU (Last Recently Used). Остальные параметры кеш-памяти:

1. CacheAssoc - ассоциативность кеш-памяти A
2. BankCount - количество банков кеш-памяти B.
3. CacheSize = BankCount*Assoc

Пример 5. Во фрагменте программы

Листинг 9. Чтение элементов одномерного массива с шагом.

```
x = 0;
for (j=0; j<N; j++)
  x += A[a*j+b]);
```

на каждой итерации происходит чтение с шагом a элементов массива A. Количество кеш-промахов ограничено сверху числом N. Т.к. физически чтение данных из оперативной памяти происходит блоками, то более точная формула количества кеш-промахов имеет вид:

$$CacheMissCount = \begin{cases} .N, a * DataTypeSize >= BlockSize \\ \dfrac{N}{\dfrac{BlockSize}{a * DatTypeSize}}, a * DataTypeSize < BlockSize \end{cases}$$

Пример 6. Фрагмент программы

Листинг 10. Повторное чтение элементов одномерного массива с шагом.

```
for (i=0; j<N1; i++) {
  x = 0
  for j=0; j<N; j++
    x += A[a*j+b];
}
```

отличается от фрагменты программы из Примера 5 тем, что в нем присутствует внешний цикл, который производит повторное обращение к считанным данным. Рассчитаем необходимые параметры кеш-памяти при которых повторные кеш-промахи производиться не будут. Введем несколько понятий:

Определение 1**.** Эффективное количество банков памяти ($EffectiveBankCount$) - количество банков кеш-памяти, в которые отображаются считываемые данные. Для вхождения массива A из Примера 6 оно равно

$$\frac{BankCount}{\left\lceil \frac{a * DataTypeSize}{BloclSize} \right\rceil} \le EffectiveBankCount \le BankCount.$$

Определение 2**.** Эффективная ассоциативность кеш-памяти ($EffectiveCasheAssoc$) – необходимая минимальная ассоциативность кеш-памяти, при которой повторное обращение к уже считанным в кеш данным не приводит к кеш-промаху. Для вхождения массива A из Примера 6 она равна

$$EffectiveCasheAssoc = \frac{EffectiveBankCount}{N}$$

Если в Примере 5 для вхождения A $EffectiveCacheAssoc > CacheAssoc$, то при каждой новой итерации по i данные массива A будут повторно считываться в кеш-память и количество кеш-промахов возрастет в N1 раз.

Пример 7. Подсчитаем количество кеш-промахов для стандартного алгоритма Флойда нахождения матрицы кратчайших расстояний:

Листинг 11. Стандартный алгоритм Флойда-Уоршалла.

```
for (int k=0; k < N; ++k)
for (int i=0; i < N; ++i)
for (int j=0; j < N; ++j)
  A[F(I, j)]  (1) += MIN(A[F(i, j)]  (2), A[F(i,k)]  (3) +  A[F(k,j)]  (4))
```

Рассмотрим случай, при котором элементы матрицы A хранятся по строкам, т.е.

$$F(i, j) = i * N + j$$

Подсчитаем количество кеш-промахов, которые произойдут во время выполнения данного алгоритма. Сначала заметим, что внутри гнезда циклов производится обращение только к элементам матрицы A. Поэтому, если размер матрицы A не превосходит размера кеш-памяти ($N^2 * DataTypeSize \le CacheSize$), то количество кеш-промахов легко подсчитать и оно не превосходит число $\frac{N^2 * DataTypeSize}{BlockSize}$. Поэтому будем предполагать, что размер матрицы A больше чем размер кеш-памяти.

Подсчитаем количество кеш-промахов, возникающих при обращении к вхождениям (1), (2), (3), (4) независимо:

$$CacheMissCount = CacheMissCount(1) + CacheMissCount(2) + CacheMissCount(3) + CacheMissCount(4)$$

На одной и той же итерации к вхождениям (2), (3), (4) при исполнении программы происходят обращения раньше, чем вхождению (1). Из того, что индексные выражения при вхождениях (1) и (2) совпадают, следует, что количество промахов к (1) равно нулю:

$$CacheMissCount(1) = 0$$

Во вхождении (2) обращение к одним и тем же данным происходит, при изменении счетчика k. Между двумя последовательными обращениями к одному и тому же элементу матрицы A во вхождении (1) производит обращение ко всем элементам матрицы A. Следовательно, при повторном обращении к элементу Aij будет всегда происходить кеш-промах.

$$CacheMissCount(2) = \frac{N^3 * DataTypeSize}{BlockSize}$$

Количество кеш-промахов, порождаемых вхождением (3), не существенно, т.к. оно ограничено $O(N^2)$.

Между двумя последовательными обращениями к одному и тому же элементу матрицы A во вхождении (4) происходит счиsstывание строки матрицы A. Подсчитаем, эффективное количество банков памяти (EffectiveBankCount) и эффективную ассоциативность кеш-памяти (EffectiveCacheAssoc) для этого вхождения. В случае, когда

EffectiveCacheAssoc > CacheAssoc,

выполняется равенство

$$CacheMissCount(4) = \frac{N^2}{BlockSize}$$

, в противном случае выполняется равенство

$$CacheMissCount(4) = \frac{N}{BlockSize}.$$

Таким образом:

$$CacheMissCount = \frac{N^3 * DataTypeSize}{BlockSize} + O(N^2)$$

Пример 8. Рассмотрим алгоритм Флойда-Уоршалла нахождения матрицы кратчайших расстояний, в котором матрица размещена блочно:

Листинг 12. Алгоритм Флойда-Уоршалла с блочным размещением матрицы кратчайших расстояний.

```
for (dk=0; dk<N; dk+=d)
for (k=MAX(dk*d, 0); i < MIN((dk+1)*d, N); ++k)
    for (di=0; di<N; di+=d)
    for (di=0; di<N; di+=d)
        for (i=MAX(di*d, 0); i < MIN((di+1)*d, N); ++i)
        for (j=MAX(dj*d, 0); j < MIN((dj+1)*d, N); ++j)
            A[F2(di, dj, i, j)]  (1) += MIN(A[F2(di, dj, i, j)]  (2),
                A[F2(di, dk, i, k)]  (3) + A[F2(dk, dj, k, j)]  (4))
```

Функция *F2* принимает следующий вид:

$$F2(di, dj, i, j) = d_i * d * N + d_j * d^2 + i * d + j$$

Как и в предыдущем примере подсчитаем количество кеш-промахов, возникающих при обращении к вхождениям (1), (2), (3), (4) независимо:

$$CacheMissCount = CacheMissCount(1) + CacheMissCount(2) + CacheMissCount(3) + CacheMissCount(4)$$

$$CacheMissCount(1) = 0$$

$$CacheMissCount(2) = \frac{N^3 * DataTypeSize}{BlockSize}$$

$$CacheMissCount(3) = \frac{N^3}{d}$$

$$CacheMissCount(4) = \frac{N^2 DataTypeSize}{BlockSize}$$

2.5. Выводы ко второй главе

Во **второй главе** представлена формула модели времени выполнения программ для иерархии памяти. Тем самым решается **задача 1**.

В начале главы приводится абстрактная формула времени выполнения программ. На основе данной формулы в **параграфах 2.1-2.2** строятся специализированные формулы времени выполнения программ, реализующих алгоритм блочного умножения матриц, а также алгоритма Флойда-Уоршалла. В **параграфе 2.3** описывается метод статического определения количества кеш-промахов, возникших во время выполнения алгоритма.

Глава 3

Умножение матриц рекордной производительности

На протяжении последних 40 лет активно развиваются различные алгоритмы умножения матриц [17], [26]. Это, в частности, связано с тем обстоятельством, что к решению задачи умножения матриц сводятся алгоритмы решения других задач (например, один из алгоритмов решения СЛАУ [94], QR разложение матрицы [24], [25], LU-разложение матрицы [27]).

На настоящее время различные алгоритмы умножения матриц реализованы во многих пакетах численных методов (Lapack, MKL, OpenBLAS, PLASMA и т.д.). Такое разнообразие библиотек обусловлено тем обстоятельством, что сложно охватить все множество существующих вычислительных систем. Учитывая тот факт, что вычислительные системы с каждым годом стремительно усложняются, особый интерес представляют новые высокопроизводительные алгоритмы, а также используемые ими методы оптимизации.

Одним из главных факторов, сдерживающим достижение высокой производительности, является скорость доступа к оперативной памяти, которая существенно ниже скорости выполнения арифметических операций. По этой причине некоторые алгоритмы, которые использовались 20-30 лет назад [95], [96] сейчас практически не находят применения [76], [100]. В работе [71] показано, что алгоритм Штрассена, по сравнению со стандартным алгоритмом умножения матриц, требует большее количество операций с памятью, что делает его неприменимым на практике. Кроме того, очевидно, что он хуже по сравнению со стандартным алгоритмом распараллеливается.

В данной работе представлено описание нового алгоритма умножения матриц, в основе которого лежит использование нестандартного размещения

матриц в оперативной памяти. В программной реализации задействованы различные методы оптимизации для процессора, имеющего кеш-память и поддержку векторных вычислений. Исследуемая программа была протестирована на процессоре с поддержкой 256-битных векторных регистров AVX. В заключительной части главы представлены результаты численных экспериментов, в которых проводилось сравнение реализованного алгоритма с функцией DGEMM из библиотек MKL, OpenBLAS, PLASMA.

3.1. Высокопроизводительные пакеты программ линейной алгебры

Статья [26] посвящена построению высокопроизводительного алгоритма умножения матриц. В ней рассматривается семейство блочных алгоритмов умножения матриц и с помощью аналитических выкладок находит оптимальный алгоритм, при котором время подкачки блоков сравнимо со временем умножения блоков. Исходная реализация алгоритма представлена в библиотеке GotoBLAS [86].

На данный момент существует много высокопроизводительных библиотек линейной алгебры. Среди них стоит выделить OpenBLAS [30], MKL [84], PLASMA [28], ATLAS [29].

Библиотека OpenBLAS [30] является библиотекой с открытым программным кодом, в основе которой лежит библиотека GotoBLAS. Она оптимизирована под процессоры архитектуры Sandy Bridge, Haswell и Loongson. В частности, в функции умножения матриц данной библиотеки реализована методика упаковки регистров, позволяющая эффективно задействовать векторные вычисления.

Библиотека PLASMA [28] является библиотекой решения задач линейной алгебры, специально оптимизированной для работы на современных многоядерных процессорах. Библиотека PLASMA отличается от своих аналогов (OpenBLAS, MKL) тем, что все алгоритмы, входящие в ее состав, оптимизированы для эффективного использования памяти. Одной из причин высокой производительности этих алгоритмов является то обстоятельство, что

они работают только с матрицами, размещенными в памяти блочным образом. На настоящее время поддерживаются алгоритмы решения нескольких наиболее важных задач линейной алгебры, таких как QR-разложение матрицы, LU-разложение матрицы, умножения матриц. Следует однако отметить, что в этой библиотеке нет поддержки эффективной работы с разреженными матрицами (sparse matrices) и матрицами-полосами (band matrices).

Библиотека ATLAS (Automatically Tuned Linear Algebra Software) является библиотекой с открытым исходным кодом. Главной ее особенностью является то, что она не использует низкоуровневую оптимизацию. Таким образом, ее легче переносить на новые платформы. В основе библиотеки ATLAS лежат алгоритмы, эффективно использующие память за счет блочных вычислений. Размеры блоков на конкретном компьютере подбираются эмпирически.

3.2. Высокопроизводительный алгорит умножения матриц, использующий двойное блочное размещение данных

Будем предполагать, что умножаются матрица $\mathbf{A} \in M_{MxK}$ и матрица $\mathbf{B} \in M_{\mathrm{KxN}}$, а результат умножения сохраняется в матрицу $\mathbf{C} \in M_{MxN}$. В основе программно реализованного автором алгоритма умножения лежит следующий блочный алгоритм умножения матриц, представленный на листинге 1.

Листинг 13. Основа алгоритма умножения матриц рекордной производительности

```
void MatrixMult(double* A, double* B, double* C) {
  for (dk=0; dk<K; dk+=Dk)
  for (di=0; di<M; di+=Di)
  for (dj=0; dj<N; dj+=Dj)
    BlockAAddr = … // вычисление адреса блока матрицы A
    BlockBAddr = … // вычисление адреса блока матрицы B
    BlockCAddr = … // вычисление адреса блока матрицы C
    BlockMult(BlockAAddr, BlockBAddr, BlockCAddr);
}
```

Внутри функции *BlockMult* производится умножение блоков матрицы **A** и матрицы **B**, результат сохраняется в блок матрицы **C**. Матрицы при реализации такого алгоритма умножения разбиваются на блоки согласно схеме представленной на Рисунке 7.

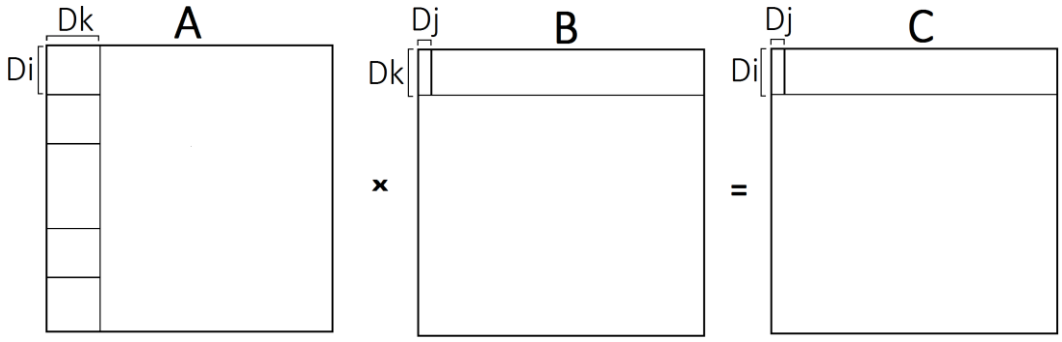

Рис. 7: Схема разбиения матриц для высокоуровневой части алгоритма умножения матриц

Значение D_k является одновременно шириной полосы матрицы **A** и высотой полосы матрицы **B**. Для того, чтобы ускорить время подкачки данных из оперативной памяти в кеш-память, данные в матрицах изначально размещаются блочно. Матрица **A** размещается в оперативной памяти блоками размера $D_i . \times D_k$, которые хранятся по столбцам (Рисунок 8).

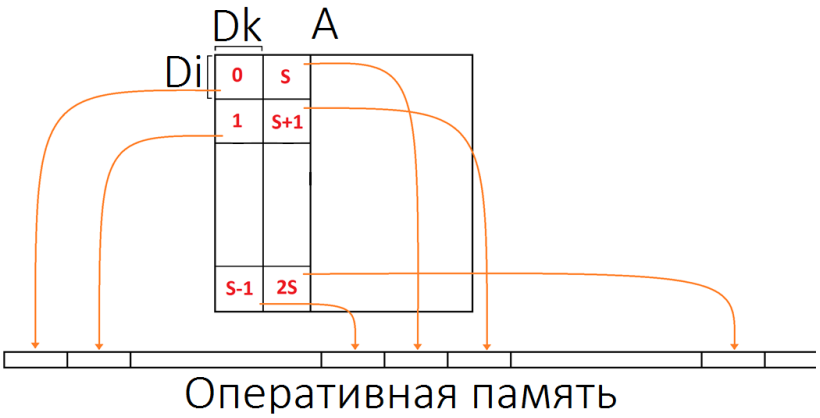

Рис. 8: Размещение блоков матрицы A в оперативной памяти. Стрелки показывают, где блоки матрицы хранятся в оперативной памяти.

Соответственно, формула вычисления адреса блока матрицы *A* имеет вид:

$$\text{BlockAAddr(di, dk)} = dk \cdot M + di \cdot D_k$$

Матрица **B** размещается в оперативной памяти блоками размера $D_k . \times D_j$, которые хранятся по строкам (Рисунок 9).

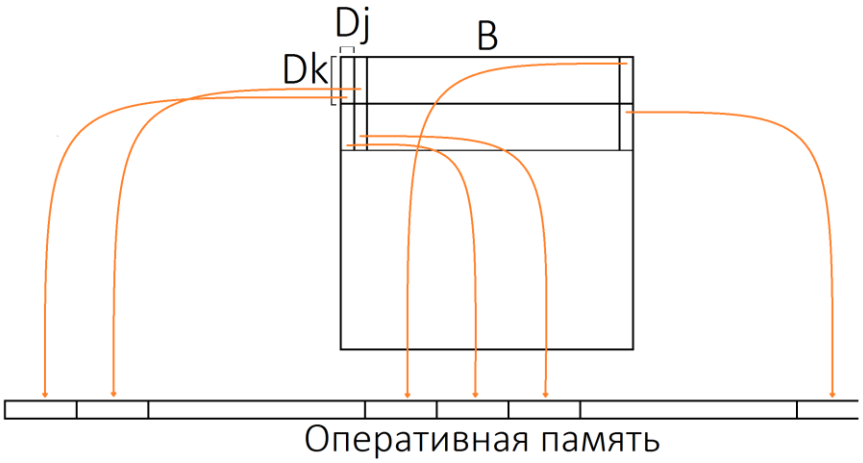

Рис. 9: Размещение блоков матрицы B в оперативной памяти. Стрелки показывают, где блоки матрицы хранятся в оперативной памяти.

Соответственно, формула вычисления адреса блока матрицы *B* имеет следующий вид:

$$\text{BlockBAddr(dk, dj)} = dk \cdot N + dj \cdot D_k$$

Матрица **C** разбивается на блоки размера $D_i . \times D_j$, которые хранятся по строкам (Рисунок 10).

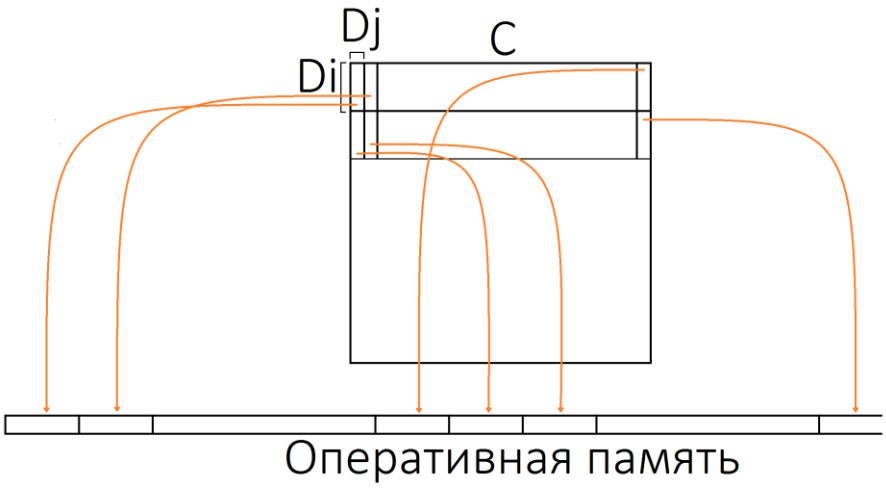

Рис. 10: Размещение блоков матрицы C в оперативной памяти. Стрелки показывают, где блоки матрицы хранятся в оперативной памяти.

Соответственно, формула вычисления адреса блока матрицы имеет вид:

$$\text{BlockCAddr(di, dj)} = di \cdot N + dj \cdot D_i$$

Для того, чтобы эффективно использовать кеш-память, размер блока матрицы **A** должен принимать максимальное значение и помещаться целиком в кеш-память, в то время как $D_j << Min(D_i.D_k)$. В реализованном алгоритме $D_j = 8$.

Алгоритм умножения блоков матрицы **A** и **B** является также блочным.

Листинг 14. Умножение блоков в рассматриваемом алгоритме умножения матриц

```
void BlockMult(double* BlockA,
     double* BlockB,
     double* BlockC)
{
  for (dk=0; dk<Dk; dk+=L)
  for (di=0; di<Di; di+=4)
    for (sk=0; sk<L; ++sk)
    for (si=0; si<4; ++si)
    for (sj=0; sj<8 ++sj)
      BlockC[di*8+sj*4+si] +=
```

```
        BlockA[dk*Di+di*L+sk*4+si]*
        BlockB[dk*8+sk*8+sj];
```

Умножаемые блоки при таком размещении разбиваются на блоки меньшего размера согласно схеме, представленной на Рисунке 11.

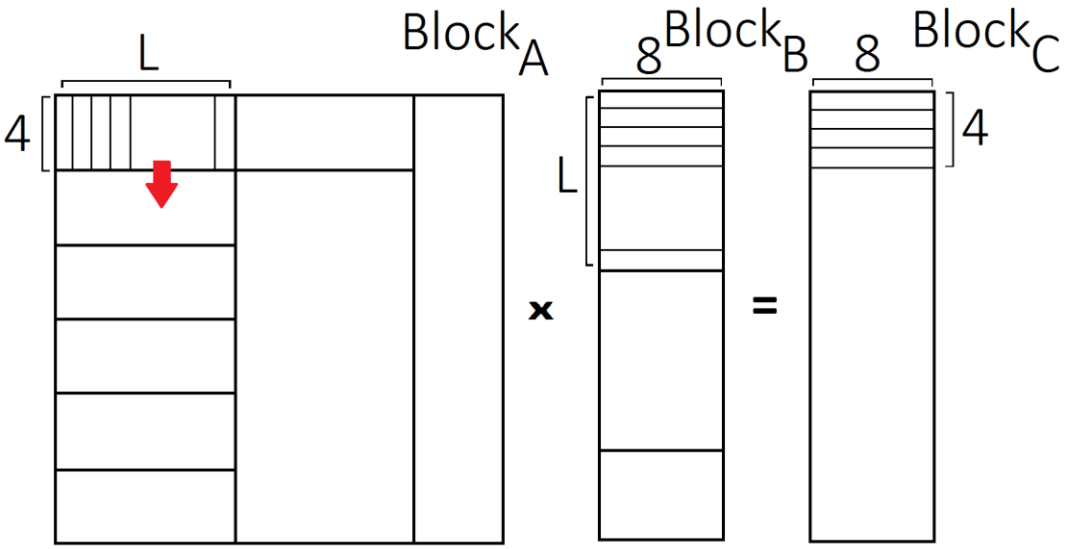

Рис. 11: Разбиение блока $Block_A$ матрицы **A** , блока $Block_B$ матрицы **B** и блока-результата $Block_C$ матрицы **C** для размещения их в оперативной памяти.

Данные внутри умножаемых блоков матрицы **A** хранятся вертикальными полосами ширины L. Внутри полосы матрицы **A** элементы хранятся блоками размера $4.\times L$. Будем называть такие блоки блоками второго уровня. Далее на Рисунке 12 продемонстрировано размещение элементов, лежащих внутри блоков матрицы **A**.

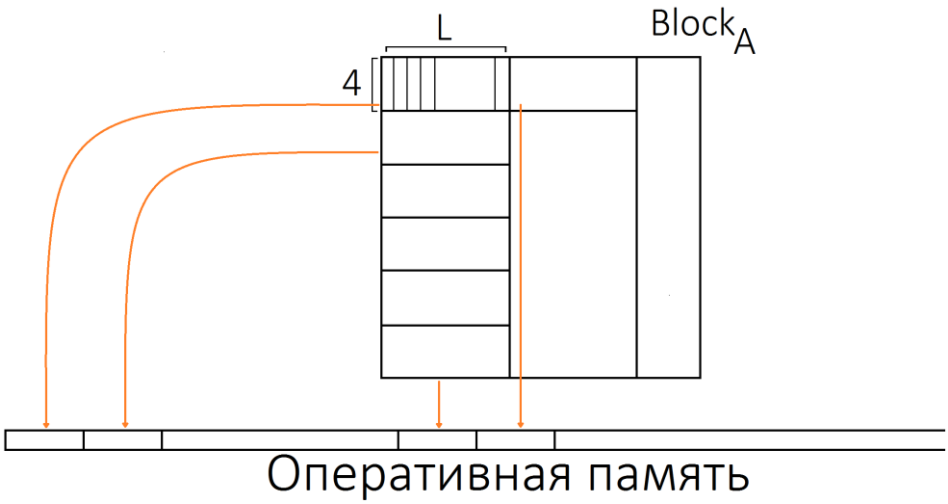

Рис. 12: Размещение блоков матрицы A в оперативной памяти

Элементы, лежащие внутри блоков матрицы **B** и матрицы **C**, хранятся по строкам. Высота блока второго уровня матрицы **A** ($Block_A$) и ширина блока матрицы **B** ($Block_C$) выбирается, исходя из количества и размеров векторных регистров процессора. Так, процессоры Intel Sandy Bridge/Ivy Bridge/Haswell включают 16 регистров размера 256 бит, каждый из которых может хранить 4 числа типа double. Для такой архитектуры эффективно разбивать матрицу **A** на блоки размера $4.\times L$, а матрицу **B** разбивать на блоки размера $L\times 8$. При умножении таких блоков восемь AVX - регистров резервируются для накопления результата, в то время как оставшиеся восемь регистров используются для загрузки данных из памяти. Подробнее использование данной методики описано в [26].

Параметр L выбирается, исходя из размера кеш-памяти L1, а именно - L должно быть таким, чтобы умножаемый блок второго уровня матрицы **A**, умножаемый блок второго уровня матрицы **B**, а также блок матрицы **C** с результатами вычислений помещались в кеш-памяти L1:

$$(4*8+4*L+8*L)*sizeof(double) \le sizeof(L1Cache)$$

В этом случае элементы блока матрицы **B** не будут вытесняться между итерациями.

Умножение блоков $4\times L$ и $L\times 8$ реализовано в отдельной процедуре, которая для достижения максимальной производительности запрограммирована на языке ассемблера. Поэтому при переносе на новую архитектуру, программную реализацию алгоритма умножения блоков потребуется каждый раз переписывать, что является непростой задачей. Так, для архитектуры Ivy Bridge размер программной реализации алгоритма умножения блоков у автора составил порядка 800 строк.

Для внутреннего цикла применяется преобразование раскрутки. Раскрутка цикла (loop unrolling) - преобразование циклов, заключающееся в многократном дублировании тела циклов с уменьшением числа итераций [99]. Также задействованы программная предвыборка данных (software prefetching), переупорядочивание инструкций, выравнивание данных.

Следует отметить, что для низкоуровневой оптимизации программной реализации умножения блоков не требуется напрямую писать код на языке ассемблера. В современных компиляторах языка C++ (например, ICC, GCC, Clang) существует возможность вставки в программную реализацию на языке C++ ассемблерных команд с использованием встроенных функций (intrinsics). Также с помощью специальных параметров компиляции (например, параметр “–march” компилятора GCC) имеется возможность указать компилятору генерировать векторные инструкции для конкретной архитектуры.

Автор провел эксперименты, в которых сравнивалась программная реализация алгоритма умножения блоков, реализованная полностью на языке ассемблера, а также программная реализация алгоритма умножения блоков, оптимизированная компилятором ICC с использованием встроенных функций. Результаты экспериментов показали, что в случае использования ручной оптимизации достигается дополнительное ускорение в 6%. Анализ сгенерированного компилятором кода показал, что компилятор неоптимально распределяет векторные регистры, что увеличивает количество обращений к

оперативной памяти. Так же компилятор не самым оптимальным образом расставляет инструкции в раскрученном цикле, что приводит к замедлению работы конвейера. Подробнее о порядке следования инструкций с целью подстройки под конвейер описано в работе [80].

3.3. Результаты численных экспериментов

В численых экспериментах сравнивалась производительность реализованного алгоритма, а также библиотек Intel MKL, PLASMA и OpenBLAS. Тестирование проводилось на вычислительной системе с процессором Intel Core i7-3820 (3600MHz, LGA2011, L3 10240Kb) с выключенной опцией Turbo Boost. Размер кеш-памяти L1 равен 64 КБ, размер кеш-памяти L2 равен 1 МБ, размер кеш-памяти L3 равен 10 МБ. В качестве компилятора использовался Intel C++ Composer XE.

Пиковая производительность вычислительной системы равна

$$P * m * H * T = 4 * 2 * 3.6 * 4 = 115{,}2 ГФЛОПС, \text{ где}$$

- $P = 4$- количество физических ядер процессора;
- m = 2 – количество команд, выполняемых параллельно;
- $H = 3.6$ Ггц – базовая тактовая частота;
- $T=4$ – количество чисел типа double, умещающихся в векторном регистре.

Исходя из размера кеш-памяти L3, $Di=512$, $Dk=256$, $Dj=8$, $L=256$. Ниже приводятся результаты проведения трех экспериментов. В каждом из экспериментов сравнивалась производительность программной реализации предложенного автором блочного алгоритма с алгоритмами умножения матриц высокоуровневых библиотек OpenBLAS, MKL, PLASMA.

Библиотека PLASMA включает возможность умножения матриц, размещенных блочно. В экспериментах присутствуют результаты измерений при использовании алгоритма, принимающего на вход стандартно размещенные

матрицы, а также - алгоритма, принимающего на вход матрицы, размещенные блочно.

Все используемые в экспериментах программные реализации для конкретных входных размеров матриц запускались по 5 раз, и считалось минимальное время работы. Производительность программной реализации считалась в гигафлопсах по следующей формуле:

$$Performance = \frac{2 * N_1 \cdot N_2 \cdot N_3}{10^9 \cdot T}, \text{ где}$$

- T – время работы программы в секундах.

На Рисунке 13 представлены результаты сравнения производительности программных реализаций алгоритмов для случая, когда матрица **A** имеет размер $512 * N$, а матрица **B** имеет размер $N * 512$.

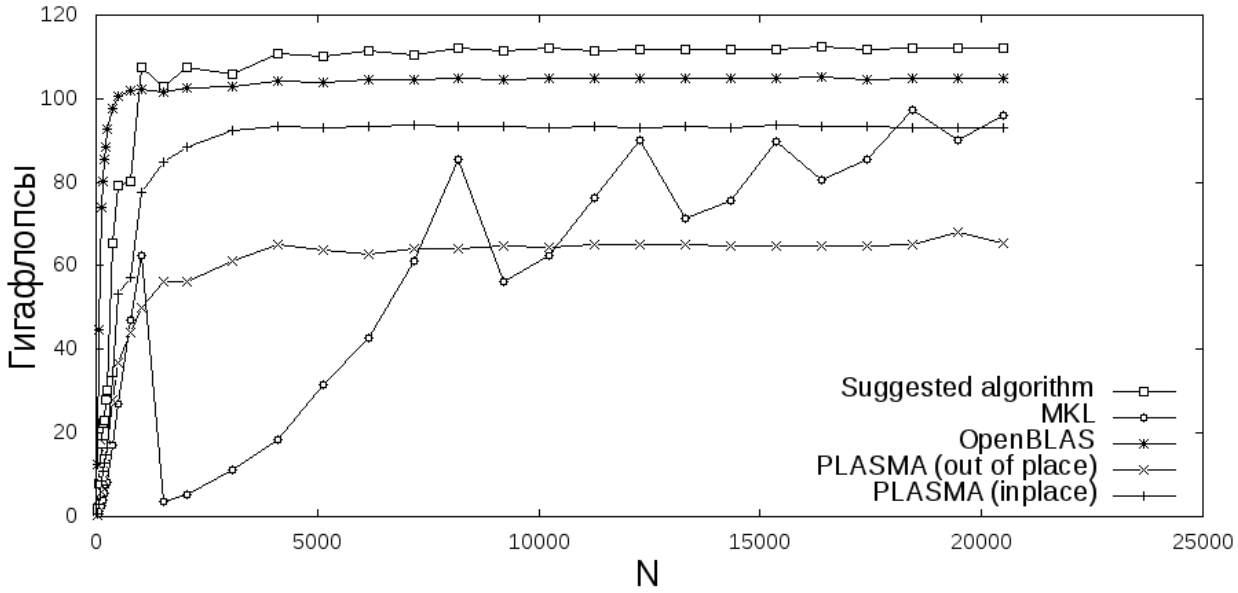

Рис. 13: Графики производительности программных реализаций блочного алгоритма, а также алгоритмов пакетов MKL, OpenBLAS и PLASMA. Случай, когда размер матрицы **A** равен *N*x*512* и размер матрицы **B** равен *512*x*N*.

В Таблице 7 приведены замеры времени работы программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA.

Таблица 7. Таблица с замерами времени работы программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA. Случай, когда размер матрицы A равен Nx512 и размер матрицы B равен 512xN

N	MKL (среднее время в сек.)	OpenBLAS (среднее время в сек.)	PLASMA, блочное размещение (среднее время в сек.)	PLASMA, стандартное размещение (среднее время в сек.)	Алгоритм с двойным блочным размещением данных (среднее время в сек.)
32	0.004794	0.000083	0.001920	0.003048	0.000571
64	0.005307	0.000094	0.001395	0.001881	0.000521
128	0.006097	0.000227	0.001974	0.002890	0.000971
160	0.006407	0.000327	0.002424	0.002915	0.001179
192	0.006905	0.000441	0.002914	0.002998	0.001630
224	0.006782	0.000580	0.003492	0.003028	0.001844
256	0.008037	0.000724	0.004123	0.004006	0.002215
384	0.008909	0.001546	0.004493	0.005426	0.002309
512	0.009914	0.002664	0.005028	0.007283	0.003386
768	0.012881	0.005934	0.010549	0.013671	0.007519
1024	0.017208	0.010512	0.013854	0.021534	0.009991
1536	0.696732	0.023805	0.028489	0.042926	0.023489
2048	0.823727	0.041884	0.048481	0.076344	0.039981
3072	0.866812	0.093776	0.104635	0.158429	0.091377
4096	0.929956	0.164710	0.184075	0.263273	0.155106
5120	0.847434	0.258017	0.288232	0.421272	0.244020
6144	0.906571	0.369676	0.414576	0.614086	0.347033

7168	0.861340	0.503857	0.561718	0.819973	0.475725
8192	0.805071	0.656025	0.735825	1.069395	0.613774
9216	1.545448	0.830966	0.930453	1.341699	0.779768
10240	1.718435	1.025341	1.154302	1.666372	0.959156
11264	1.705499	1.240318	1.389926	1.999716	1.164584
12288	1.719523	1.472992	1.659906	2.380248	1.382080
13312	2.538303	1.730635	1.943833	2.788779	1.624753
14336	2.788191	2.005603	2.258346	3.250929	1.880880
15360	2.693315	2.303709	2.582565	3.721568	2.159584
16384	3.415150	2.614615	2.941697	4.254285	2.446925
17408	3.628306	2.965379	3.323294	4.787849	2.773088
18432	3.576990	3.314098	3.734034	5.348168	3.105658
19456	4.305338	3.696274	4.162980	5.689152	3.462284
20480	4.477606	4.094069	4.612154	6.580666	3.826870

В Таблице 8 приведены значения производительности, рассчитанные относительно пиковой производительности, программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA.

Таблица 8. Таблица со значениями производительности, рассчитанными относительно пиковой производительности, программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA. Случай, когда размер матрицы A равен Nx512 и размер матрицы B равен 512xN

N	MKL	OpenBLAS	PLASMA, блочное размещение	PLASMA, стандартное размещение	Алгоритм с двойным блочным размещением
32	0.001899	0.110196	0.004740	0.002986	0.015941
64	0.006860	0.388984	0.026100	0.019354	0.069910

128	0.023886	0.641002	0.073777	0.050396	0.149954
160	0.035514	0.695889	0.093876	0.078064	0.193040
192	0.047454	0.742702	0.112450	0.109292	0.201080
224	0.065764	0.768981	0.127723	0.147295	0.241923
256	0.072484	0.804394	0.141291	0.145417	0.262999
384	0.147126	0.847814	0.291712	0.241581	0.567755
512	0.235029	0.874622	0.463439	0.319938	0.688177
768	0.407018	0.883562	0.497021	0.383493	0.697247
1024	0.541642	0.886636	0.672798	0.432839	0.932907
1536	0.030100	0.880956	0.736127	0.488548	0.892831
2048	0.045261	0.890146	0.769023	0.488349	0.932506
3072	0.096775	0.894535	0.801702	0.529488	0.918026
4096	0.160363	0.905414	0.810165	0.566450	0.961475
5120	0.274968	0.903107	0.808435	0.553126	0.954910
6144	0.370125	0.907672	0.809367	0.546412	0.966895
7168	0.530235	0.906435	0.813065	0.556986	0.960036
8192	0.740957	0.909299	0.810687	0.557814	0.971895
9216	0.488515	0.908550	0.811405	0.562700	0.968204
10240	0.542393	0.909032	0.807473	0.559339	0.971758
11264	0.661274	0.909284	0.811411	0.563981	0.968416
12288	0.780552	0.911191	0.808586	0.563881	0.971128
13312	0.620570	0.910183	0.810355	0.564833	0.969498
14336	0.655211	0.910874	0.808934	0.561948	0.971275
15360	0.778651	0.910337	0.812042	0.563513	0.971091
16384	0.698679	0.912598	0.811128	0.560868	0.975140
17408	0.742406	0.908375	0.810544	0.562607	0.971363
18432	0.844257	0.911228	0.808750	0.564660	0.972386
19456	0.781533	0.910312	0.808258	0.591435	0.971834
20480	0.832648	0.910652	0.808358	0.566549	0.974235

Полученные результаты свидетельствуют о том, что предложенный автором блочный алгоритм быстрее чем алгоритмы из других библиотек. В частности, данный алгоритм для *N=20480* работает на 14% быстрее MKL, на 6% быстрее OpenBLAS, на 17% быстрее алгоритма пакета PLASMA с блочным размещением входных матриц и на 41% быстрее алгоритма пакета PLASMA со стандартным размещением входных матриц.

На Рисунке 14 представлены результаты сравнения производительности программных реализаций алгоритмов для случая, когда матрица **A** и матрица **B** имеют размер $N * N$.

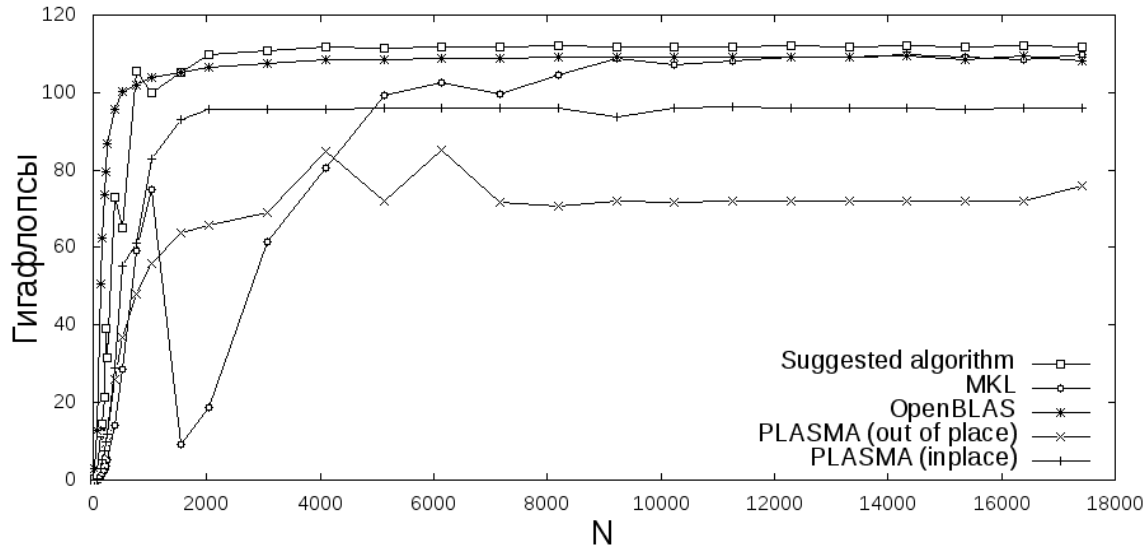

Рис. 14: Графики производительности программных реализаций блочного алгоритма, а также алгоритмов из пакетов MKL, OpenBLAS и PLASMA. Случай, когда матрица **A** и матрица **B** квадратные.

В Таблице 9 приведены замеры времени работы программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA.

Таблица 9. Таблица с замерами времени работы программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA. Случай, когда матрица A и матрица B квадратные.

N	MKL (среднее время в сек.)	OpenBLAS (среднее время в сек.)	PLASMA, блочное размещение (среднее время в сек.)	PLASMA, стандартное размещение (среднее время в сек.)	Алгоритм с двойным блочным размещением данных (среднее время в сек.)
32	0.002455	0.000022	0.001599	0.001884	0.000300
64	0.004438	0.000041	0.001225	0.001791	0.000253
128	0.005031	0.000083	0.001389	0.001875	0.000346
160	0.005244	0.000131	0.001582	0.002049	0.000569
192	0.005642	0.000192	0.001892	0.002138	0.000659
224	0.006060	0.000283	0.002244	0.002371	0.000575
256	0.006464	0.000386	0.002808	0.002826	0.001066
384	0.008016	0.001184	0.003912	0.004337	0.001554
512	0.009379	0.002673	0.004865	0.007306	0.004126
768	0.015323	0.008880	0.014835	0.018821	0.008579
1024	0.028683	0.020659	0.025912	0.038390	0.021456
1536	0.789802	0.068860	0.077888	0.113880	0.068918
2048	0.924078	0.161122	0.179821	0.261053	0.156541
3072	0.940943	0.539537	0.607007	0.840124	0.523852
4096	1.703943	1.268469	1.436395	1.620985	1.230058
5120	2.703669	2.476988	2.796308	3.736112	2.410771
6144	4.521604	4.261892	4.834312	5.438316	4.150973
7168	7.403821	6.778762	7.678455	10.256352	6.598716
8192	10.531584	10.071060	11.441679	15.569594	9.805591

9216	14.372511	14.362380	16.706144	21.729283	14.014868
10240	20.017718	19.661179	22.356848	30.029021	19.186708
11264	26.392723	26.173286	29.721026	39.715437	25.565923
12288	34.028953	33.959862	38.616189	51.650480	33.094541
13312	43.191824	43.174399	49.130200	65.659302	42.183483
14336	53.618242	53.863678	61.324462	82.002863	52.620787
15360	66.436454	66.806660	75.861025	100.741712	64.778354
16384	81.189183	80.280407	91.603202	121.939967	78.385985
17408	95.945400	97.411367	109.858208	138.889735	94.342319

В Таблице 10 приведены значения производительности, рассчитанные относительно пиковой производительности, программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA.

Таблица 10. Таблица со значениями производительности, рассчитанными относительно пиковой производительности, программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA. Случай, когда матрица A и матрица B квадратные.

N	MKL	OpenBLAS	PLASMA, блочное размещение	PLASMA, стандартное размещение	Алгоритм с двойным блочным размещением
32	0.000232	0.025397	0.000356	0.000302	0.001895
64	0.001026	0.110464	0.003714	0.002542	0.018003
128	0.007237	0.437607	0.026205	0.019422	0.105106
160	0.013561	0.542005	0.044962	0.034709	0.124888
192	0.021780	0.640667	0.064961	0.057464	0.186408
224	0.032202	0.689014	0.086964	0.082305	0.339355
256	0.045058	0.754980	0.103714	0.103083	0.273289

384	0.122642	0.830621	0.251314	0.226674	0.632505
512	0.248451	0.871808	0.479005	0.318930	0.564807
768	0.513226	0.885652	0.530119	0.417853	0.916694
1024	0.649905	0.902340	0.719404	0.485578	0.868826
1536	0.079659	0.913660	0.807753	0.552464	0.912890
2048	0.161383	0.925577	0.829330	0.571266	0.952665
3072	0.534907	0.932867	0.829177	0.599098	0.960799
4096	0.700168	0.940541	0.830584	0.736001	0.969911
5120	0.861854	0.940727	0.833302	0.623688	0.966566
6144	0.890510	0.944776	0.832907	0.740400	0.970021
7168	0.863606	0.943238	0.832717	0.623417	0.968974
8192	0.906262	0.947703	0.834176	0.613014	0.973360
9216	0.945523	0.946190	0.813446	0.625402	0.969652
10240	0.931243	0.948130	0.833809	0.620778	0.971576
11264	0.940094	0.947976	0.834818	0.624735	0.970496
12288	0.946613	0.948539	0.834165	0.623658	0.973340
13312	0.948213	0.948596	0.833602	0.623751	0.970879
14336	0.954001	0.949654	0.834118	0.623781	0.972085
15360	0.946989	0.941741	0.829340	0.624514	0.971228
16384	0.940457	0.951103	0.833540	0.626169	0.974090
17408	0.954553	0.940188	0.833665	0.659408	0.970773

Полученные результаты свидетельствуют о том, что предложенный автором блочный алгоритм работает быстрее чем алгоритмы из других библиотек. В частности, данный алгоритм для *N=17408* работает на 2% быстрее MKL, на 3% быстрее OpenBLAS, на 14% быстрее алгоритма пакета PLASMA с блочным размещением входных матриц и на 31% быстрее алгоритма пакета PLASMA со стандартным размещением входных.

На Рисунке 15 представлены результаты сравнения производительности программных реализаций рассматриваемых алгоритмов для случая, когда размер матрицы **A** равен $N \cdot 2048$ и размер матрицы **B** равен $2048 \cdot N$.

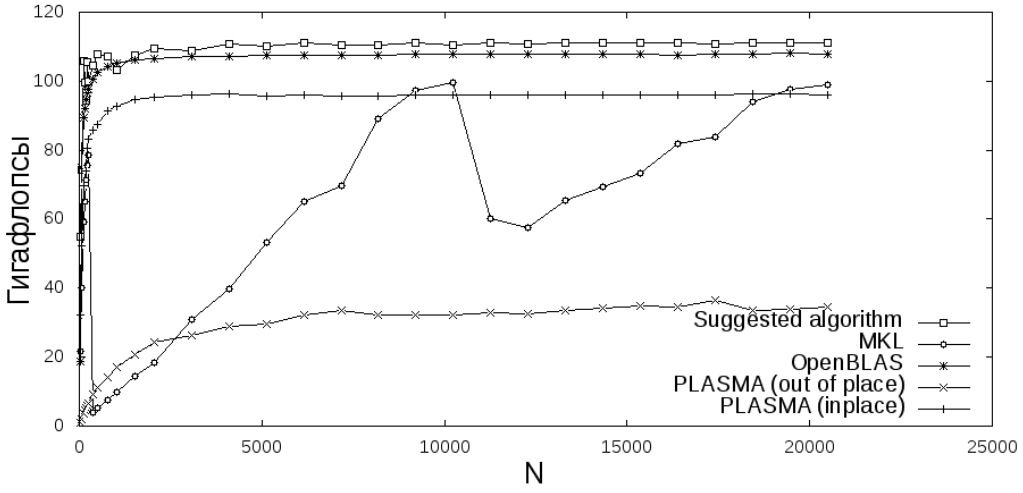

Рис. 15: График производительности реализованного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA. Случай, когда размер матрицы **A** равен $N \cdot 2048$ и размер матрицы **B** равен $2048 \cdot N$.

В Таблице 11 приведены замеры времени работы программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA.

Таблица 11. Таблица с замерами времени работы программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA. Случай, когда размер матрицы A равен $N * 2048$ и размер матрицы B равен $2048 * N$.

N	MKL (среднее время в сек.)	OpenBLAS (среднее время в сек.)	PLASMA, блочное размещение (среднее время в сек.)	PLASMA, стандартное размещение (среднее время в сек.)	Алгоритм с двойным блочным размещением данных (среднее время в сек.)
32	0.012393	0.014278	0.008360	0.271543	0.004892
64	0.013419	0.007158	0.010275	0.282903	0.007221

128	0.018123	0.012029	0.015395	0.297994	0.010157
160	0.020582	0.014602	0.017994	0.297511	0.013468
192	0.022572	0.016983	0.021777	0.310545	0.017082
224	0.024801	0.019459	0.023291	0.320953	0.017779
256	0.027343	0.021986	0.025770	0.330169	0.021498
384	0.795484	0.031977	0.037488	0.348231	0.030763
512	0.823887	0.041862	0.049057	0.385440	0.039823
768	0.840108	0.061885	0.070406	0.459303	0.060168
1024	0.856845	0.081632	0.092815	0.505227	0.083110
1536	0.890664	0.121329	0.136176	0.625341	0.119766
2048	0.925739	0.161234	0.180366	0.704137	0.156703
3072	0.832133	0.240290	0.267993	0.984827	0.237099
4096	0.860258	0.320193	0.356879	1.193293	0.310163
5120	0.806357	0.399078	0.449324	1.457211	0.389600
6144	0.793022	0.479088	0.536627	1.597571	0.463410
7168	0.861941	0.558966	0.629557	1.798361	0.543711
8192	0.771055	0.638530	0.717720	2.131150	0.621812
9216	0.794077	0.716800	0.804585	2.406689	0.696253
10240	0.862648	0.796616	0.895182	2.657666	0.776924
11264	1.568321	0.876807	0.983970	2.868626	0.850509
12288	1.796598	0.956505	1.073333	3.165721	0.929618
13312	1.703776	1.035891	1.163631	3.331086	1.005239
14336	1.730219	1.115237	1.252984	3.528876	1.083349
15360	1.755757	1.195639	1.340506	3.712208	1.160562
16384	1.676569	1.276761	1.429646	3.990392	1.236680
17408	1.739671	1.354827	1.520355	3.994614	1.316731
18432	1.646603	1.434749	1.607821	4.628184	1.390724
19456	1.674235	1.511131	1.697082	4.827923	1.469466
20480	1.733300	1.592229	1.788221	4.960932	1.545657

В Таблице 12 приведены значения производительности, рассчитанные относительно пиковой производительности, программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA.

Таблица 12. Таблица со значениями производительности, рассчитанными относительно пиковой производительности, программных реализаций блочного алгоритма, а также пакетов MKL, OpenBLAS и PLASMA. Случай, когда размер матрицы A равен $N * 2048$ и размер матрицы B равен $2048 * N$.

N	MKL	OpenBLAS	PLASMA, блочное размещение	PLASMA, стандартное размещение	Алгоритм с двойным блочным размещением
32	0.188026	0.163198	0.278742	0.008581	0.476303
64	0.347294	0.651085	0.453552	0.016473	0.645351
128	0.514307	0.774876	0.605451	0.031278	0.917624
160	0.566064	0.797905	0.647485	0.039161	0.865063
192	0.619385	0.823255	0.642002	0.045021	0.818465
224	0.657672	0.838233	0.700327	0.050821	0.917462
256	0.681770	0.847874	0.723385	0.056460	0.867104
384	0.035151	0.874442	0.745885	0.080297	0.908938
512	0.045252	0.890618	0.759991	0.096728	0.936215
768	0.066568	0.903677	0.794304	0.121759	0.929465
1024	0.087023	0.913434	0.803378	0.147588	0.897185
1536	0.125578	0.921859	0.821352	0.178859	0.933890
2048	0.161094	0.924932	0.826821	0.211792	0.951677
3072	0.268823	0.930942	0.834709	0.227143	0.943472
4096	0.346712	0.931506	0.835751	0.249948	0.961627
5120	0.462360	0.934221	0.829751	0.255850	0.956948
6144	0.564162	0.933842	0.833712	0.280045	0.965436

7168	0.605561	0.933792	0.829088	0.290241	0.959991
8192	0.773646	0.934213	0.831136	0.279907	0.959330
9216	0.845117	0.936228	0.834080	0.278843	0.963858
10240	0.864378	0.936027	0.832964	0.280567	0.959752
11264	0.522992	0.935462	0.833582	0.285928	0.964387
12288	0.498044	0.935473	0.833651	0.282648	0.962530
13312	0.568942	0.935765	0.833039	0.291001	0.964298
14336	0.603343	0.936048	0.833144	0.295821	0.963600
15360	0.637036	0.935468	0.834372	0.301298	0.963741
16384	0.711600	0.934432	0.834505	0.298980	0.964717
17408	0.728650	0.935627	0.833761	0.317330	0.962696
18432	0.815119	0.935479	0.834780	0.290001	0.965092
19456	0.846203	0.937538	0.834811	0.293448	0.964121
20480	0.860387	0.936617	0.833962	0.300610	0.964838

Полученные результаты свидетельствуют о том, что реализованный алгоритм работает быстрее чем алгоритмы из других библиотек. В частности, данный алгоритм для N=20480 работает на 10% быстрее MKL, на 3% быстрее OpenBLAS, на 13% быстрее алгоритма пакета PLASMA с блочным размещением входных матриц и на 66% быстрее алгоритма пакета PLASMA со стандартным размещением входных.

3.4. Выводы к третьей главе

В **третьей главе** описан и реализован высокопроизводительный алгоритм умножения матриц. Начиная с размерности матриц N > 2000 как для прямоугольных так и для квадратных матриц предлагаемый алгоритм опережает все остальные алгоритмы (MKL, PLASMA, OpenBLAS) по производительности. Ближе всех по производительности к предлагаемому алгоритму является пакет OpenBLAS, который имеет производительность на 3-6% меньше чем у предлагаемого алгоритма.

Описанный алгоритм является новым. Его новизна состоит в использовании нестандартного двойного блочного размещения матриц. Такое размещение матриц позволяет уменьшить количество промахов к данным кеш-памяти, к данным кеш-памяти TLB, увеличить скорость подкачки данных в векторные регистры и, тем самым, увеличить общую производительность алгоритма. Таким образом, в данной главе раскрывается решение задачи 2.

В **параграфе 3.1** описываются существующие высокопроизводительные алгоритмы умножения матриц. В частности, упоминаются пакеты Atlas, Intel MKL, PLASMA и OpenBLAS. В **параграфе 3.2** приводится описание алгоритма умножения матриц. Его особенностью является то, что данные хранятся в памяти небольшими блоками нестандартным образом, а не стандартным образом. В **параграфе 3.3** приводятся результаты численных экспериментов, в которых сравнивается производительность программной реализации разработанного алгоритма, а также существующих других высокопроизводительных реализаций (Intel MKL, PLASMA, OpenBLAS). Согласно результатам численных экспериментов представленный в диссертации алгоритм по производительности превосходит остальные алгоритмы, и его производительность составляет в среднем 98-99% от теоретической производительности процессора.

Глава 4

Автоматизация блочных размещений данных в системе OPC

В предыдущих главах показана важность метода размещения данных в оперативной памяти и его влияние на время выполнения программы. Современные языки программирования используют разные методы размещения данных в оперативной памяти. Язык ФОРТРАН использует размещение по столбцам [10] в то время как языки C/C++/Pascal используют размещение по строкам [10]. Поэтому программисту для написания быстрого алгоритма необходимо учитывать способ размещения данных, используемый в языке программирования. Также, современные компиляторы не имеют встроенной поддержки нестандартного размещения массивов.

Ручное изменение метода размещения данных является затруднительной операцией, т.к. предполагает высокую квалификацию программиста и дополнительные затраты времени. Непосредственное применение, например, блочного размещения увеличивает объем результирующей программы и порождает сложные индексные выражения во вхождениях блочно размещаемых массивов. Поэтому представляется интересной задача автоматизации поддержки блочного размещения массивов в языках программирования. Такая автоматизация реализована в системе OPC (Оптимизирующая распараллеливающая система) в виде нескольких директив компилятора. Она может использоваться для получения дополнительного ускорения блочных программ. Детали реализации такой автоматизации описаны в данной главе. В конце главы приводятся результаты ускорения программ за счет данных директив на задачах линейной алгебры (LU-разложение матрицы, умножение матриц), преобразования Фурье и т.д.

4.1. Оптимизирующая распараллеливающая система

Оптимизирующая распараллеливающая система представляет собой инструмент для построения оптимизирующих компиляторов. Она состоит из нескольких модулей:

1. Парсеры языка ФОРТРАН и Си. Парсер языка ФОРТРАН разработан автором диссертации. Особенности его реализации приводятся в параграфе 4.8.
2. Высокоуровневое внутреннее представление (Reprise). В его состав входит несколько десятков преобразований программ (гнездование цикла, канонизация циклов, разрезание цикла и т.д.). Также есть поддержка трансляции данного внутреннего представления в SSA форму.
3. Модули отображения внутреннего представления в языки Си, ФОРТРАН, VHDL.

4.2. Директивы блочного размещения данных в компиляторе языка Си

Автором диссертации в системе компиляторов ОРС поддержка блочного размещения массивов реализована в виде нескольких директив компиляции. Перед объявлением блочно размещаемого массива указывается директива.

```
#pragma ops array declare(name, array dimension size list, block
dimension size list)
```
(1)

Список параметров директивы:

- **name** – имя размещаемого массива.
- **array dimension size list** – массив размерностей размещаемого массива по каждому измерению.
- **block dimension size list** – массив размерностей блока по каждому измерению.

В данной директиве указывается информация о размере массива, а также размере блоков, на которые указанный массив стоит разбить.

Оператор, в котором производится выделение памяти для массива A, следует пометить директивой

#pragma ops array allocate(name) (2)

Директива (2) заменяет исходный оператор выделения памяти новым оператором освобождения памяти, в котором выделяется память под блочно размещенный массив. Блочно размещенный массив должен иметь размер кратный размеру блока, т.к. выполнение данного условия на практике сильно упрощает генерацию индексных выражений. Аналогично, оператор, в котором производится освобождение памяти массива A, следует пометить директивой

#pragma ops array release (name) (3)

Директива (3) заменяет исходный оператор освобождения памяти на новый оператор освобождения памяти, в котором освобождается память, выделенная под блочно размещенный массив. Ниже приведен пример использования директив блочного размещения матриц для блочного алгоритма умножения матриц:

Листинг 15. Блочный алгоритм умножения матриц с использованием директив блочного размещения матриц.

```
void matrix_mult(int N, int d) { ...
    #pragma ops array declare(A, N, N, d, d)
    #pragma ops array declare(B, N, N, d, d)
    #pragma ops array declare(C, N, N, d, d)
```

```
    double *A, *B, *C;

    #pragma ops array allocate(A)
    A = malloc(sizeof(double)*N*N);
    #pragma ops array allocate(B)
    B = malloc(sizeof(double)*N*N);
    #pragma ops array allocate(C)
    C = malloc(sizeof(double)*N*N);

    // инициализация матриц A, B

    // блочное умножение матриц
    for(di = 0; di < N; i+=d)
    for(dj = 0; dj < N; j+=d)
    for(dk = 0; dk < N; k+=d)
        for(i = di; i < MIN(N, di+d); i++)
        for(k = dk; k < MIN(N, dk+d); k++)
        for(j = dj; j < MIN(N, dj+d); j++)
            C[i*N+j] += A[i*N+k]*B[k*N+j];

    #pragma ops array release(A)
    free(A);
    #pragma ops array release(B)
    free(B);
    #pragma ops array release(C)
    free(C);
}
```

Для того, чтобы компилятор переразметил данные, к входной программе предъявляется следующее требование: в программе не должно быть операций взятия адреса блочно размещаемого массива, кроме как в операциях выделения и освобождения памяти массива. Операции выделения и освобождения памяти должны быть аннотированы директивами (2) и (3) соответственно.

4.3. Работа с блочным размещением в OPS Demo5

Блочное размещение данных легче всего применить к программе с помощью специальной программы OPSDemo5 [75], которая предназначена для демонстрации работы преобразований системы OPC. При запуске программа OPSDemo5 имеет следующий вид (Рисунок 16):

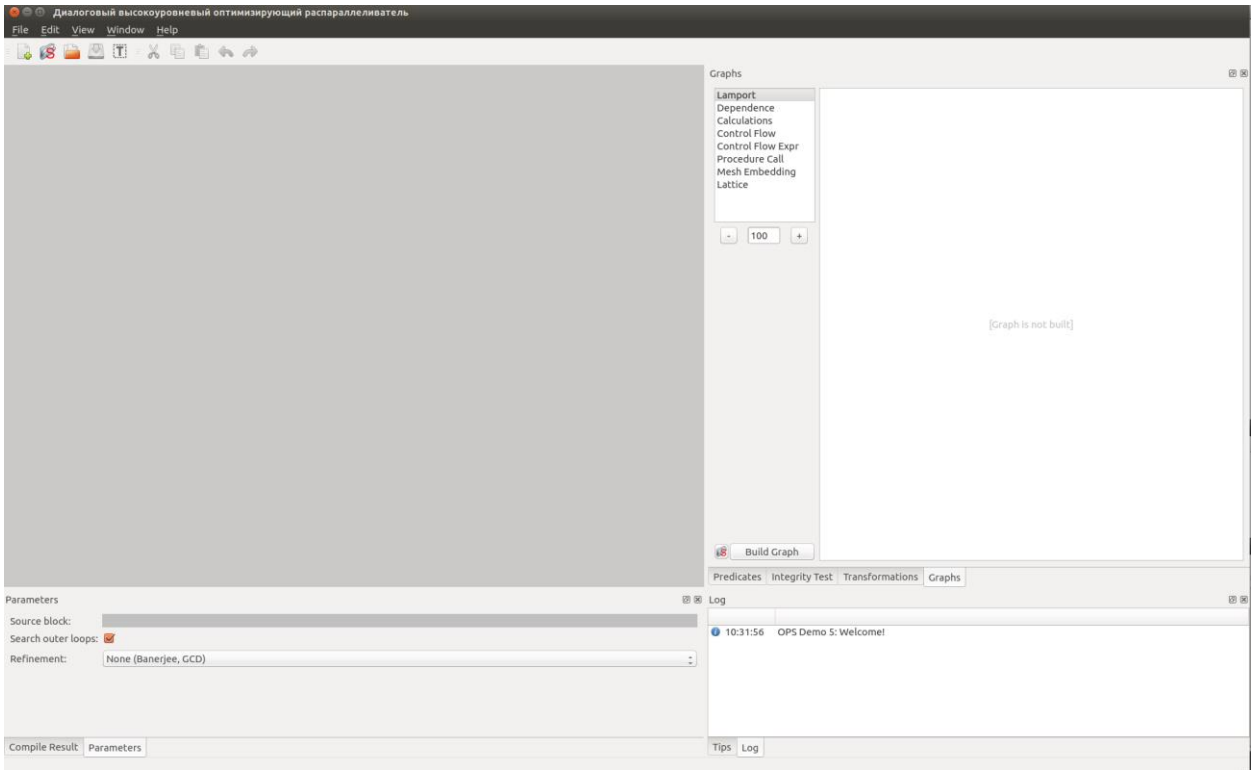

Рис. 16: Программа OPS5Demo

Для того чтобы применить к своей программе блочное размещение данных, пользователь должен с помощью кнопки **Open** загрузить ее исходный код в OPSDemo5 (Рисунок 17).

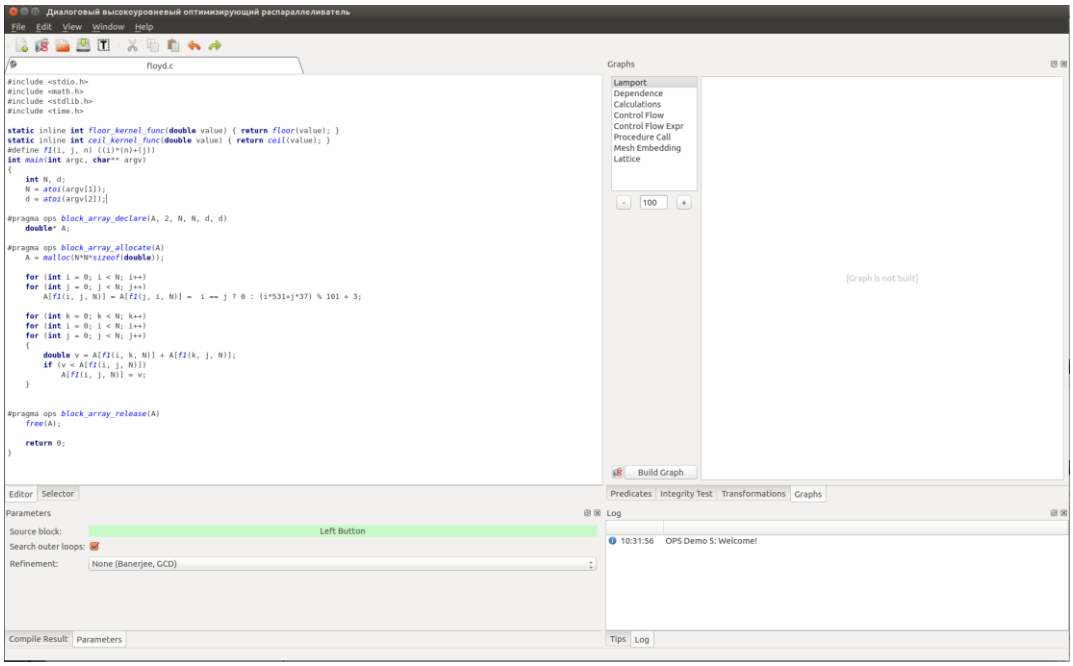

Рис. 17: Загруженная в OPSDemo5 программа пользователя

Далее, во вкладке **Transforms** пользователь должен выбрать преобразование “Data Distribution for shared memory” и нажать кнопку **Apply** (Рисунок 18).

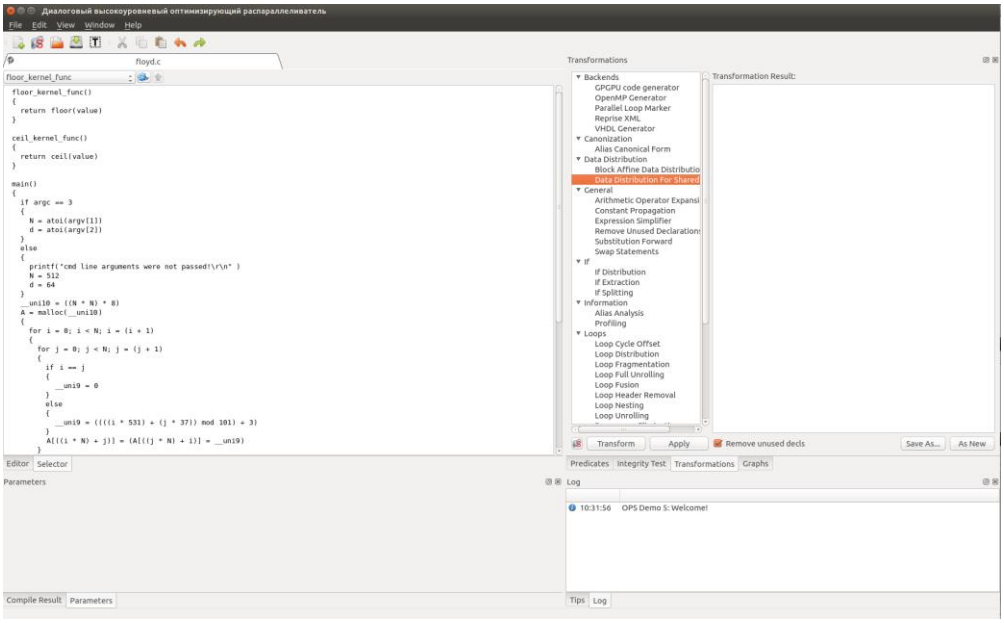

Рис. 18: Применение преобразования “Data distribution for shared memory”

После этого будет применено блочное размещение данных. Результат работы программы отобразится в правой вкладке “Transformation Result” (Рисунок 19).

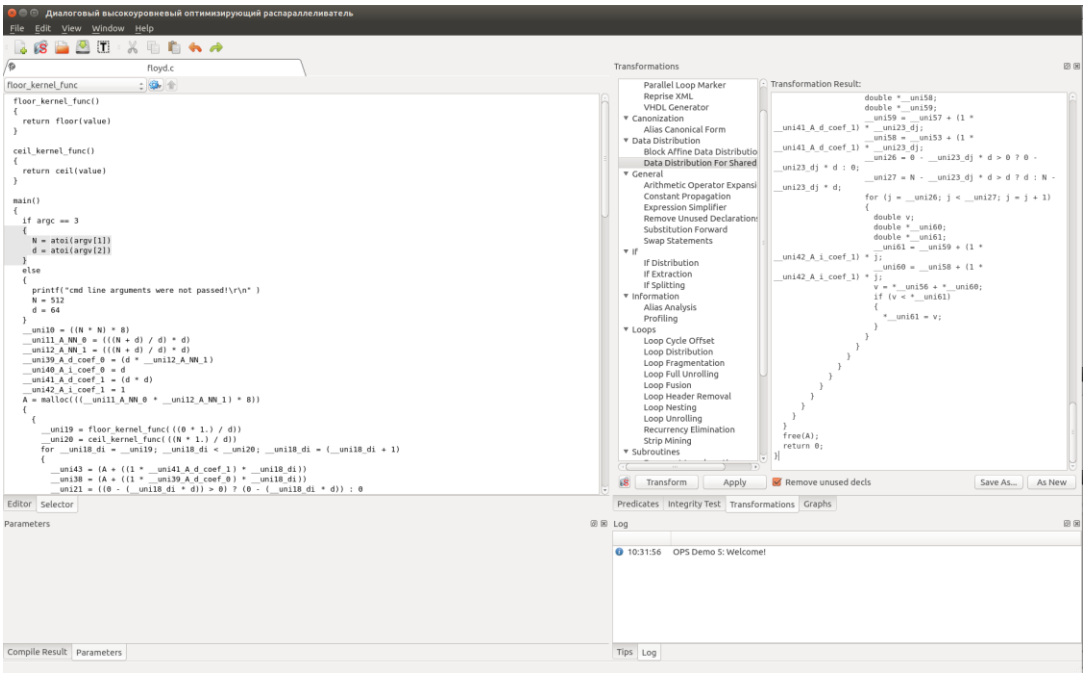

Рис. 19: Результат применения преобразования “Data distribution for shared memory”

Скачать программу OpsDemo5 можно по адресу [75].

4.4. Реализация блочного размещения данных в Web- распараллеливателе программ.

Описанные в параграфе 4.2 директивы блочного размещения данных вошли в состав Web-распараллеливатель [67], [77]. Web- распараллеливатель – это программа, предназначенная для ускорения блочных алгоритмов за счет эффективного использования кеш-памяти или распараллеливания на суперкомпьютер с распределенной памятью. Данный проект основан на базе ДВОР. На Рисунке 20 представлено главная страница Web- распараллеливателя.

Для того чтобы применить блочное размещение данных к программе

необходимо на Шаге 1 выбрать опцию “Автоматическое распределение данных под кеш-память”.

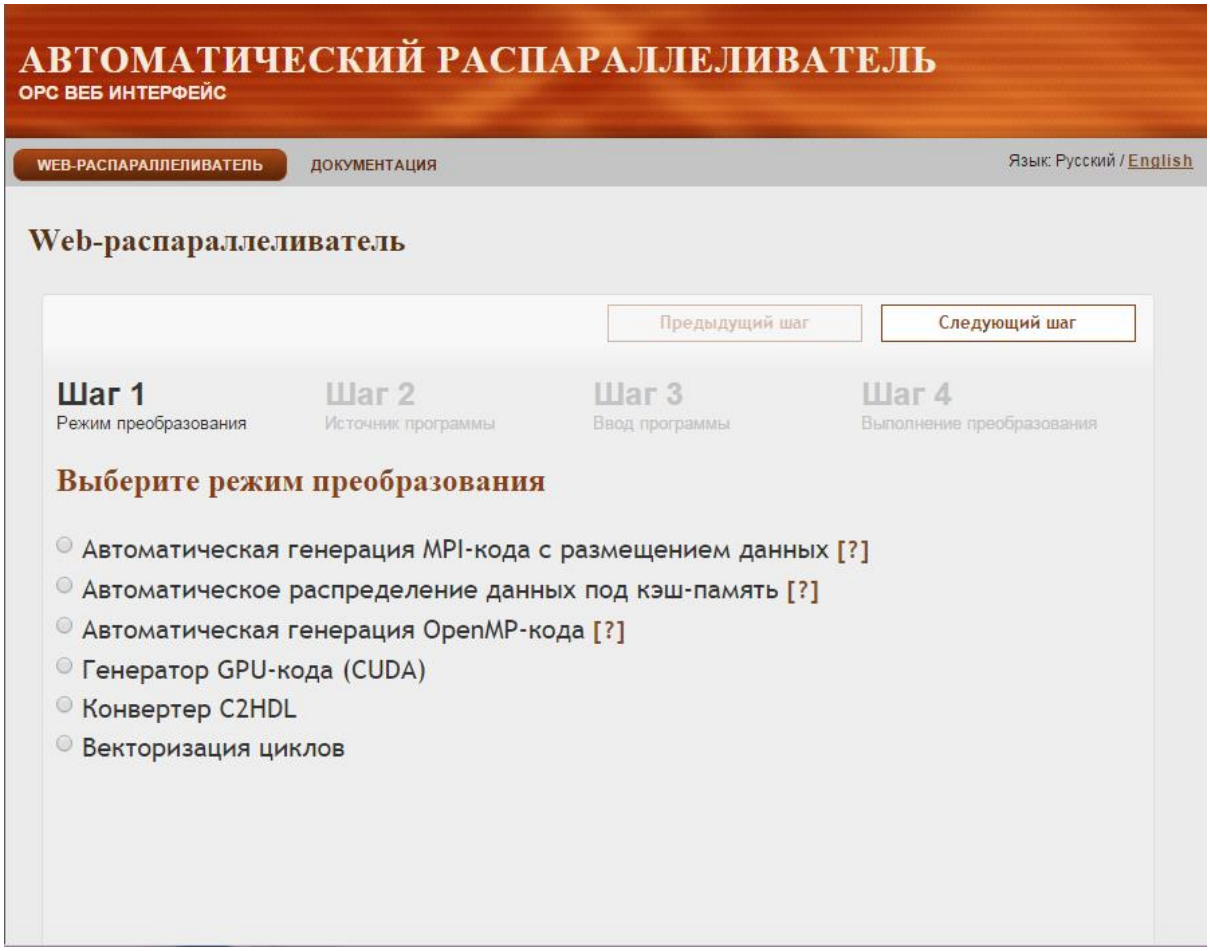

Рис. 20: Web-распараллеливатель программ. Выбор режима преобразования.

На Шаге 2 необходимо указать источник программы, которую необходимо преобразовать (Рисунок 21). На вход Web-распараллеливателю подаются программы на языке Си, соответствующие стандарту C99. Входная программа должна состоять из одного *.c файла. Если программа включает нестандартные заголовочные файлы, то перед загрузкой необходимо включить их исходный код в текст программы с помощью препроцессора.

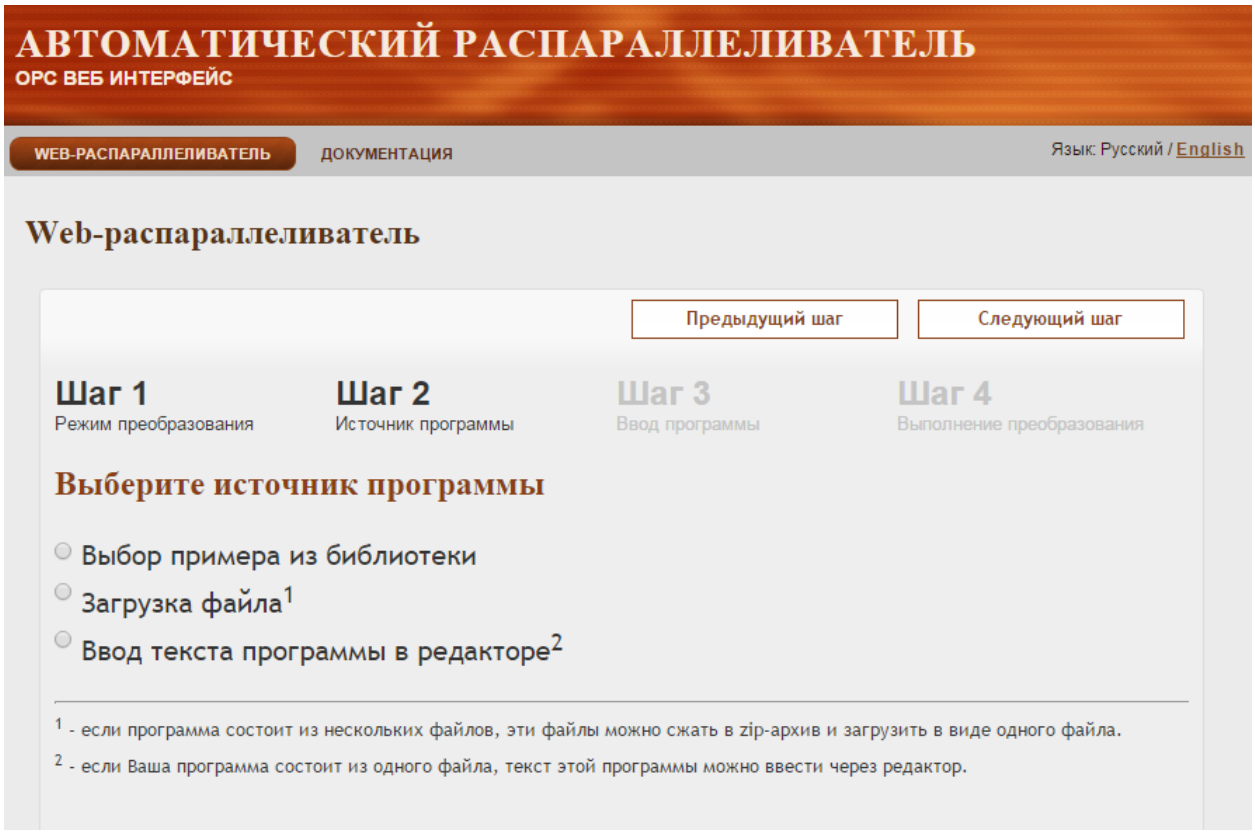

Рис. 21: Web-распараллеливатель программ. Выбор источника программы.

Результатом работы Web-распараллеливателя является преобразованная программа.

4.5. *Проблема вычисления адреса элемента при блочном размещении массива и анализ вариантов ее решения.*

Пусть дан многомерный массив $A[N_1, N_2, ..N_p]$. Будем считать, что изначально данные в массиве хранятся по строкам. Т.е. элемент $A[i_1, i_2, ..i_p]$ относительно начала массива хранится по адресу $Addr[A, i_1, i_2, ..i_p] = i_1 * N_2 * ... * N_p + i_2 * N_3 * ... * N_p + i_p$. Выражение i_k будем называть индексом по k-му измерению при вхождении массива A.

Предположим, что массив A нужно разместить блочным образом с размером блока равным $(BlockSize_1, BlockSize_2, ..., BlockSize_p)$. В этом случае соответствующий ему блочный массив имеет размер $A^{block}[N_1^{block}, N_2^{block}, ..N_p^{block}]$, где

$N_k^{block} = \lceil N_k / BlockSize_k \rceil$. При этом элемент $A[i_1, i_2, ..i_p]$ относительно начала массива уже будет храниться по адресу

$$
\begin{aligned}
&Addr(A^{block}, i_1, i_2, ..i_p] = di_1 * BlockSize_1 * N_2 * ... * N_p \\
&+ di_2 * BlockSize_1 * BlockSize_2 * N_3 * ..N_p + ... \\
&+ di_p * BlockSize_1 * BlockSize_2 * ... * BlockSize_p \\
&+ wi_1 * BlockSize_2 * BlockSize_3 * .. * BlockSize_p \\
&+ wi_2 * BlockSize_3 * BlockSize_4 * .. * BlockSize_p + ... + wi_p = \\
&= \begin{pmatrix} di_1 & di_2 & ... & di_p & wi_1 & wi_2 & ... & wi_p \end{pmatrix} * \begin{pmatrix} BlockSize_1 * N_2 * ... * N_p \\ BlockSize_1 * BlockSize_2 * N_3 * ..N_p \\ ... \\ BlockSize_1 * BlockSize_2 * ... * BlockSize_p \\ BlockSize_2 * BlockSize_3 * .. * BlockSize_p \\ BlockSize_3 * .. * BlockSize_p \\ ... \\ 1 \end{pmatrix}
\end{aligned} \quad (3)
$$

Где $di_k = \lfloor i_k / BlockSize_k \rfloor$ - номер блока по измерению *k*, а $wi_k = (i_k \% BlockSize_k)$ - номер элемента относительно начала блока по измерению *k*.

В формуле (3) второй умножаемый вектор можно вычислить заранее. Тем не менее, вычисление адреса элемента блочно размещенного массива является затратной операцией, т.к. для получения переменных di_k и wi_k требуются дополнительные операции целочисленного деления и взятия остатка.

На практике в большинстве случаев дополнительные вычисления можно избежать. Эффективное вычисление индексных выражений блочно размещенных массивов зависит от формы записи циклов, счетчики которых в них используются.

Введем несколько определений.

Определение 3. Цикл с заголовком “for (di=LBound; di<RBound;di+=step)” называется потенциальным циклом по блокам i-го измерения массива *A*, если $LBound \geq 0$ и $RBound <= \lfloor N / BlockSizek \rfloor$

Определение 4. Цикл с заголовком “for (di=LBound; di<RBound;di+=step)” называется потенциальным циклом по адресам блоков k-го измерения массива *A*, если

1) $LBound \geq 0$
2) $LBound\%BlockSize_k$ равно 0
3) $RBound < N$
4) $step\%BlockSize_k$ равно 0

Определение 5. Цикл с заголовком “for (wi=LBound; wi<RBound; wi+=step)” называется потенциальным циклом по элементам блока по k-го измерения массива *A*, если

1) $LBound \geq 0$
2) $RBound < BlockSize_k$

Определение 6. Цикл с заголовком “for (i=LBound; i<RBound;i+=step)” называется потенциальным циклом по адресам элементов блока по k-му измерению массива *A*, если

1) $LBound \geq 0$
2) $RBound < N$
3) $\exists\ s \geq 0$, при котором $i \geq BlockSize_k * s$ и $i < BlockSize_k * (s+1)$

Рассмотрим варианты записи циклов, счетчики которых входят в индексные выражения блочно размещаемых массивов: для каждого случая выпишем условия, накладываемые на индексные выражения блочно размещаемых массивов, а также формулы быстрого вычисления di_k и wi_k.

Вариант 1. Внешний цикл, является потенциальным циклом по блокам k-го измерения массива *A*; внутренний цикл является потенциальным циклом по элементам блоков k-го измерения массива *A*.

Если некоторое обращение к массиву *A* имеет вид $A[BlockSize * f(di) + g(i)]$ и выполняются следующие условия:

1. *di* – счетчик внешнего цикла, *i* – счетчик внутреннего цикла.
2. *BlockSize* - размер блока по k-му измерению;
3. $0 \le f(di) \le \lfloor N/BlockSize \rfloor$, $0 \le g(i) < BlockSize$,

то $di_k = f(di)$, $wi_k = g(i)$.

Пример 9:

Листинг 16. Гнездо циклов, удовлетворяющее условиям варианта 1

```
// локальный индекс блока
for (di = 0; di <= N/BlockSize; ++di)
 // локальный счетчик элемента внутри блока
 for (i = 0; i <BlockSize; ++i)
   A[di*BlockSize+i] (1) = di*BlockSize+i;
```

Для вхождения (1) одномерного блочно размещаемого массива A $di_1 = di$, $wi_1 = i$.

Вариант 2. Внешний цикл, является потенциальным циклом по блокам k-го измерения массива *A*; внутренний цикл является потенциальным циклом по адресам элементов блоков *i*-го измерения массива *A*.

Если некоторое обращение к массиву *A* имеет вид $A[i + free\_coef]$ и выполняются следующие условия:

1. *di* – счетчик внешнего цикла, *i* – счетчик внутреннего цикла.
2. *BlockSize* - размер блока по k-му измерению.

3. $\text{free\_coef} < \text{BlockSize}$ - некоторая константа.

4. $\exists$ k: $k * \text{BlockSize} \le i + \text{free\_coef} < (k+1) * \text{BlockSize}$.

то $\text{di}_k = k$, $\text{wi}_k = i + free\_coef - k * BlockSize$.

Пример 10:

Листинг 17. Гнездо циклов, удовлетворяющее условиям варианта 2

```
// локальный индекс блока
for (di = 0; di <= N/BlockSize; ++di)
 // глобальный счетчик элемента внутри блока
 for (i = MAX(0, di*BlockSize); i <MIN(N, (di+1)*BlockSize-2); ++i)
  A[i+2] (1) = i;
```

Для вхождения (1) одномерного блочно размещаемого массива A верны равенства $\text{di}_1 = di$ и $\text{wi}_1 = i + 2 - di * BlockSize$.

Вариант 3. Внешний цикл, является потенциальным циклом по адресам блоков k-го измерения массива *A*; внутренний цикл является потенциальным циклом по элементам блоков k-го измерения массива *A*.

Если некоторое обращение к массиву *A* имеет вид $A[di + g(i)]$ и выполняются следующие условия:

1. *di* – счетчик внешнего цикла, *i* – счетчик внутреннего цикла.

2. *BlockSize* - размер блока по k-му измерению;

3. $0 \le g(i) < BlockSize$,

то $\text{di}_k = di / BlockSize$, $\text{wi}_k = g(i)$.

Пример 11:

Листинг 18. Гнездо циклов, удовлетворяющее условиям варианта 3

```
// глобальный индекс блока
for (di = 0; di < N; di+=BlockSize)
 // локальный счетчик элемента внутри блока
 for (i = 0; i < MIN(N-di,BlockSize); ++i)
   A[di+i] = di+i;
```

Для вхождения (1) одномерного блочно размещаемого массива A верны равенства $\mathrm{di}_1 = di / BlockSize$ и $\mathrm{wi}_1 = i$.

Вариант 4. Внешний цикл, является потенциальным циклом по адресам блоков k-го измерения массива *A*; внутренний цикл является потенциальным циклом по адресам элементов блоков *i*-го измерения массива *A*.

Если некоторое обращение к массиву *A* имеет вид $\mathrm{A[i + free\_coef]}$ и выполняются следующие условия:

1. *di* – счетчик внешнего цикла, *i* – счетчик внутреннего цикла.
2. $BlockSize$ - размер блока по k-му измерению.
3. $\mathrm{free\_coef} < \mathrm{BlockSize}$ - некоторая константа.
4. $\exists$ k: $\mathrm{di} + \mathrm{k} * \mathrm{BlockSize} \le \mathrm{i} + \mathrm{free\_coef} < \mathrm{di} + (\mathrm{k}+1) * \mathrm{BlockSize}$.

то $\mathrm{di}_k = di / BlockSize + k$, $\mathrm{wi}_k = i + free\_coef - di_k * BlockSize$.

Пример 12:

Листинг 19. Гнездо циклов, удовлетворяющее условиям варианта 4

```
// глобальный индекс блока
for (di = 0; di < N; ++di)
 // глобальный счетчик элемента внутри блока
 for (i = MAX(di,0); i < MIN(di+BlockSize,N); ++i)
```

```
A[i] (1) = i;
```

Для вхождения (1) одномерного блочно размещаемого массива A $\mathrm{di}_1 = di / BlockSize$, $\mathrm{wi}_1 = i - di_1 * BlockSize$.

Вариант 5 (Общий случай).

Если некоторое обращение к массиву *A* имеет вид A[f(i)] и выполняется условие $0 \le \mathrm{f(i)} < \mathrm{N}$, то $\mathrm{di}_1 = f(i) / BlockSize$, $\mathbf{wi}_1 = f(i)\%BlockSize$.

Выше для каждого из вариантов 1-5 записи циклов описан алгоритм получения wi_k и di_k. Для варианта 1 генерация индексов является самой быстрой, т.к. не требует дополнительных вычислений. Для вариантов 2-4 вычислений производится несколько больше. Вариант 5 является самым медленным, т.к. в нем используются операции деления и деления по модулю, которые являются ресурсоемкими.

4.6. Алгоритм блочного размещения данных.

Алгоритм блочного размещения данных реализован в ОРС в виде отдельного преобразования, которое применяется к каждой функции, определенной в исходной программе. При анализе конкретной функции производится поиск всех директив блочного размещения данных, которыми аннотируются:

- Объявления блочных массивов. Перед ними ставится директива *#pragma ops block_array_declare*.
- Операторы выделения памяти под блочные массивы. Перед ними ставится директива *block_array_allocate*.
- Операторы выделения памяти под блочные массивы. Перед ними ставится директива *block_array_release*.

Для каждого аннотированного массива производится проверка на то, что его можно разместить блочно. Единственное требование, которое накладывается на блочно размещаемый массив, заключается в том, чтобы к обращениям массива не применялись операции адресной арифметики, за исключением быть может операторов, аннотированных директивами *block_array_allocate* и *block_array_release*.

Далее производится двухшаговый алгоритм генерации индексных выражений блочно размещаемых массивов. На каждом шаге строится специальная таблица. С каждой строкой таблицы сопоставлено вхождение p-мерного массива A_l, а также измерение k ($0 \le k < p$). С каждым столбцом сопоставляется цикл, который встречается в тексте программы. Если индекс A_l по измерению k линейно зависит от счетчика i некоторого цикла, то в соответствующей ячейке таблицы находится коэффициент при i. Рассмотрим пример построения такой таблицы.

Пример 13. Пусть на вход преобразованию подается следующая программа:

Листинг 20. Программа, содержащая директивы ops block array declare и ops block array allocate

```
int main()
{
	int i,j, i1, j1;
	int N1, N2;
	int d1, d2;

#pragma ops block_array_declare(A, 2, N1, N2, d1, d2)
	double* A;

   d1 = 5; N1 = 7;
   d2 = 8; N2 = 9;
```

```
#pragma ops block_array_allocate(A)
      A = malloc(N1*N2*sizeof(double));

      for (i1=0; i1<N1; i++)
   for (j1=0; j1<N2; j1++)
      A[i1*N2+j1] (1) = i1+j1;
}
```

В данной функции выполняется блочное переразмещение матрицы A. В процессе выполнения преобразования будет построена следующая таблица (Таблица 13):

Таблица 13. Таблица с коэффициентами при индексах, линейно зависящих от счетчиков циклов, получаемая на первом шаге алгоритма.

<Вхождение>:<номер измерения>	i1	j1	i	j
A1: 0	1	0	0	0
A2: 0	0	0	1	0
A1: 1	0	1	0	0
A2 1	0	0	0	1

В данном примере каждая из строк таблицы попадает под вариант 5 (см. параграф 4.5) обращения к элементам блочно размещаемого массива. Генерация индексных выражений для варианта 5 приведет к замедлению результирующей программы.

Поэтому на первом шаге производится проверка на то, что каждая строка таблицы (в данном примере, это A1:0, A1:1, A2:0, A2:1) попадает под варианты 1-4 обращения к элементам блочно размещаемого массива (см. параграф 4.5). Если

данное условие выполняется, то дополнительных преобразований программы не требуется и производится переход на второй шаг алгоритма.

В противном случае преобразование пытается преобразовать некоторые циклы входной программы так, чтобы количество строк таблицы, попадающих под варианты 1-4, было максимально. В качестве вспомогательного преобразования используется гнездование циклов.

В данном примере к каждому из циклов применяется гнездование к каждому из циклов программы. Исходная таблица примет следующий вид (Таблица 14):

Таблица 14. После применения гнездования к циклам программ

Пара <Вхождение>:<номер измерения>	di_	i	dj_	j	di_2	i	dj_2	j
A1: 0	d1	1	0	0	0	0	0	0
A2: 0	0	0	0	0	d1	1	0	0
A1: 1	0	0	d2	1	0	0	0	0
A2 1	0	0	0	0	0	0	d2	1

Теперь каждая из строк таблицы попадает под вариант 1 и, следовательно, генерируемые индексные выражения будут содержать минимальное количество арифметических операций.

На втором шаге алгоритма производится генерация индексных выражений согласно формуле (3). В качестве дополнительных оптимизаций производятся предвычисление констант, а также вынос общих подвыражений. Для нашего примера результирующая программа будет иметь следующий вид:

Листинг 21. Результат работы преобразования блочного размещения массивов.

```
int main()
{
  …
  d1 = 5;
  N1 = 7;
  d2 = 8;
  N2 = 9;
  __uni9_A_NN_0 = ((N1 + d1) / d1) * d1;
  __uni10_A_NN_1 = ((N2 + d2) / d2) * d2;
  __uni32_A_d_coef_0 = d1 * __uni10_A_NN_1;
  __uni33_A_i_coef_0 = d2;
  __uni34_A_d_coef_1 = d1 * d2;
  __uni35_A_i_coef_1 = 1;
  A = malloc(((__uni9_A_NN_0 * __uni10_A_NN_1) * 8));
  {
    int __uni16_di;
    int __uni17;
    int __uni18;
    int __uni19;
    int __uni20;
    __uni17 = floor_kernel_func(((0 * 1.) / d1));
    __uni18 = ceil_kernel_func(((N1 * 1.) / d1));
    for (__uni16_di = __uni17; __uni16_di < __uni18; __uni16_di = __uni16_di + 1)
    {
      double *__uni31;
      __uni31 = A + (1 * __uni32_A_d_coef_0) * __uni16_di;
      __uni19 = 0 - __uni16_di * d1 > 0 ? 0 - __uni16_di * d1 : 0;
      __uni20 = N1 - __uni16_di * d1 > d1 ? d1 : N1 - __uni16_di * d1;
```

```
    for (i = __uni19; i < __uni20; i = i + 1)
    {
      double *__uni36;
      __uni36 = __uni31 + (1 * __uni33_A_i_coef_0) * i;
      {
        int __uni11_dj;
        int __uni12;
        int __uni13;
        int __uni14;
        int __uni15;
        __uni12 = floor_kernel_func(((0 * 1.) / d2));
        __uni13 = ceil_kernel_func(((N2 * 1.) / d2));
        for (__uni11_dj = __uni12; __uni11_dj < __uni13; __uni11_dj = __uni11_dj + 1)
        {
          double *__uni37;
          __uni37 = __uni36 + (1 * __uni34_A_d_coef_1) * __uni11_dj;
          __uni14 = 0 - __uni11_dj * d2 > 0 ? 0 - __uni11_dj * d2 : 0;
          __uni15 = N2 - __uni11_dj * d2 > d2 ? d2 : N2 - __uni11_dj * d2;
          for (j = __uni14; j < __uni15; j = j + 1)
          {
            double *__uni38;
            __uni38 = __uni37 + (1 * __uni35_A_i_coef_1) * j;
            *__uni38 = (i + __uni16_di * d1) + (j + __uni11_dj * d2);
          }
        }
      }
    }
  }
}
{
```

```
int __uni26_di;
int __uni27;
int __uni28;
int __uni29;
int __uni30;
__uni27 = floor_kernel_func(((0 * 1.) / d1));
__uni28 = ceil_kernel_func(((N1 * 1.) / d1));
for (__uni26_di = __uni27; __uni26_di < __uni28; __uni26_di = __uni26_di + 1)
{
 double *__uni39;
 __uni39 = A + (1 * __uni32_A_d_coef_0) * __uni26_di;
 __uni29 = 0 - __uni26_di * d1 > 0 ? 0 - __uni26_di * d1 : 0;
 __uni30 = N1 - __uni26_di * d1 > d1 ? d1 : N1 - __uni26_di * d1;
 for (i = __uni29; i < __uni30; i = i + 1)
 {
  double *__uni40;
  __uni40 = __uni39 + (1 * __uni33_A_i_coef_0) * i;
  {
   int __uni21_dj;
   int __uni22;
   int __uni23;
   int __uni24;
   int __uni25;
   __uni22 = floor_kernel_func(((0 * 1.) / d2));
   __uni23 = ceil_kernel_func(((N2 * 1.) / d2));
   for (__uni21_dj = __uni22; __uni21_dj < __uni23; __uni21_dj = __uni21_dj + 1)
   {
    double *__uni41;
    __uni41 = __uni40 + (1 * __uni34_A_d_coef_1) * __uni21_dj;
    __uni24 = 0 - __uni21_dj * d2 > 0 ? 0 - __uni21_dj * d2 : 0;
```

```
          __uni25 = N2 - __uni21_dj * d2 > d2 ? d2 : N2 - __uni21_dj * d2;
          for (j = __uni24; j < __uni25; j = j + 1)
          {
            double *__uni42;
            __uni42 = __uni41 + (1 * __uni35_A_i_coef_1) * j;
            printf(" %6.2f", (*__uni42));
          }
        }
       }
       printf("\r\n");
     }
    }
  }
  free(A);
  return 0;
}
```

4.7. Численные эксперименты

Для проверки эффективности и корректности реализованных директив написан пакет прикладных блочных программ, аннотированных директивами блочного размещения данных. В Таблице 15 приведены результаты сравнения производительности программ, скомпилированных с включенной опцией блочного размещения данных, а также без нее. Тестирование производилось на компьютере с процессором Intel Core i5-2410M Processor (3M Cache, 2.90 GHz). В качестве компилятора использовался Intel C++ Composer XE 2013 SP1 for Linux, Update 3 [38].

Таблица 15. Результаты тестирования директив блочного размещения в памяти компилятором Intel C++ Composer XE 2013 SP1 for Linux, Update 3

Название алгоритма	**Размер матриц**	**Размер блока**	**Время работы алгоритма без директив (сек.)**	**Время работы алгоритма с директивами (сек.)**	**Ускорение**
Двумерное быстрое преобразование Фурье	4096x4096	256x256	17.9	10.54	41%
Блочный алгоритм Флойда	2048x2048	256x256	47.49	41.17	13.3%
Блочное QR-разложение матрицы	2048x1024	256x256	19.6	17.11	12.7%
Блочное LU-разложение матрицы	2048x2048	256x256	27.4	14.44	47%
Блочное умножение квадратных матриц	2048x2048	256x256	14.93	11.2	25%
Блочное возведение матрицы в квадрат	2048x2048	256x256	81.37	17.36	78.6%

Как видно из результатов, представленных в Таблице 15, ускорение, получаемое за счет использования реализованного преобразования блочного размещения данных, сильно зависит от входной программы. Так, для программы, реализующей блочное QR-разложение матрицы, ускорение составляет 1.15 раза, в то время как для программы, реализующей быстрое блочное возведение матрицы в квадрат, ускорение составляет 4.69 раз.

Тест 1. Быстрое двумерное преобразование Фурье

В данном эксперименте в качестве входной программы подается программа, реализующая быстрое преобразование Фурье. Исходный текст программы, реализующей двумерное быстрое преобразование Фурье взят из [61]. и ручному переписыванию не подвергался. В текст программы были добавлены директивы блочного размещения данных.

На Рисунке 22 представлен график времени работы алгоритма со стандартным размещением входной матрицы и график времени работы того же самого алгоритма с блочным размещением входной матрицы. Переход к блочному размещению был произведен при помощи реализованного автором преобразования блочного размещения данных.

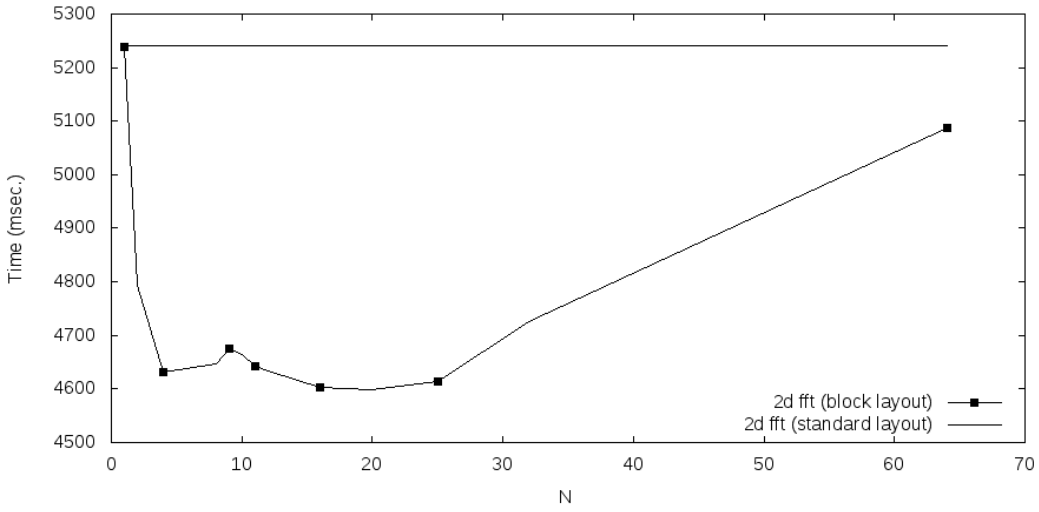

Рис. 22: График сравнения производительности исходной программы и преобразованной программы. По вертикальной оси отложено время работы программы, по горизонтальной оси – размер блоков, на которые разбивается входная матрица

По горизонтальной оси отложен размер блоков, на которые разбивается входная матрица при блочном ее размещении. Как видно из результатов эксперимента блочное размещение входной матрицы в данном эксперименте

позволило ускорить входную программу. Оптимальный размер блока находится в диапазоне – 15-20.

Тест 2. Блочный алгоритм Флойда

В данном эксперименте в качестве входной программы подается программа, реализующая блочный алгоритм Флойда:

Листинг 22. Блочный алгоритм Флойда, аннотированный директивами блочного размещения массивов.

```
// Цикл по блокам. Идем по диагонали
for (int L = 0; L < n; L+= d)
{
	int k1 = L;
	int k2 = MIN(n, L+d);

	// Фаза 1. Рассчитываем self-depended block (не используя другие блоки)
	for (int k = k1; k < k2; k++)
	for (int i = k1; i < k2; i++)
	for (int j = k1; j < k2; j++)
	{
		double v = A[f1(i, k, n)] + A[f1(k, j, n)];
		if (v < A[f1(i, j, n)])
			A[f1(i, j, n)] = v;
	}

	// Фаза 2. Рассчитываем блоки на той же строке и на том же столбце.
	for (int K = 0; K < n; K+=d)
	{
		if (K == L)
			continue;
```

```
        int j1 = K;
        int j2 = MIN(n, K+d);

        int i1 = L;
        int i2 = MIN(n, L+d);

        for (int k = k1; k < k2; k++)
        for (int i = i1; i < i2; i++)
        for (int j = j1; j < j2; j++)
        {
                double v = A[f1(i, k, n)] + A[f1(k, j, n)];
                if (v < A[f1(i, j, n)])
                        A[f1(i, j, n)] = v;
        }

        for (int k = k1; k < k2; k++)
        for (int j = j1; j < j2; j++)
        for (int i = 0; i < d; i++)
        {
                double v = A[f1(j, k, n)]  +A[f1(k, i, n)];
                if (v < A[f1(j, i, n)])
                        A[f1(j, i, n)] = v;
        }

}

// Фаза 3. Рассчитываем остальные блоки
for (int K = 0; K < n; K+=d)
{
```

```
            int i1 = K;
            int i2 = MIN(n, K + d);
            for (int S = 0; S < n; S+=d)
            {
                if (K == L || S == L)
                    continue;
                int j1 = S;
                int j2 = MIN(n, S + d);

                for (int k = k1; k < k2; k++)
                for (int i = i1; i < i2; i++)
                for (int j = j1; j < j2; j++)
                {
                    double v = A[f1(i, k, n)] + A[f1(k, j, n)];
                    if (v < A[f1(i, j, n)])
                        A[f1(i, j, n)] = v;
                }
            }
        }
}
```

В Таблице 16 представлено время работы программной реализации данного алгоритма, использующего стандартное размещение матрицы, а также время работы программной реализации того же самого алгоритма, использующего блочное размещение входной матрицы. Размер блока выступает в качестве параметра. Размер матрицы фиксирован и равен 2048. Переход к блочному размещению был произведен при помощи реализованного автором преобразования блочного размещения данных.

Таблица 16. Время работы алгоритма Флойда до и после применения блочного размещения матрицы.

Размер матрицы	**Размер блока**	**Время работы программы со стандартным размещением матрицы (мсек)**	**Время работы программы с блочным размещением матрицы (мсек)**
8	16	13210.39	12190.31
48	32	12137.88	7495.38
2048	64	10860.37	6071.04
2048	96	10614.73	5304.56
2048	128	10249.03	4939.49
2048	192	14885.37	5339.54
2048	256	20949.18	5418.81

На Рисунке 23 представлен график времени работы данного алгоритма со стандартным размещением входной матрицы и график времени работы того же самого алгоритма с блочным размещением входной матрицы.

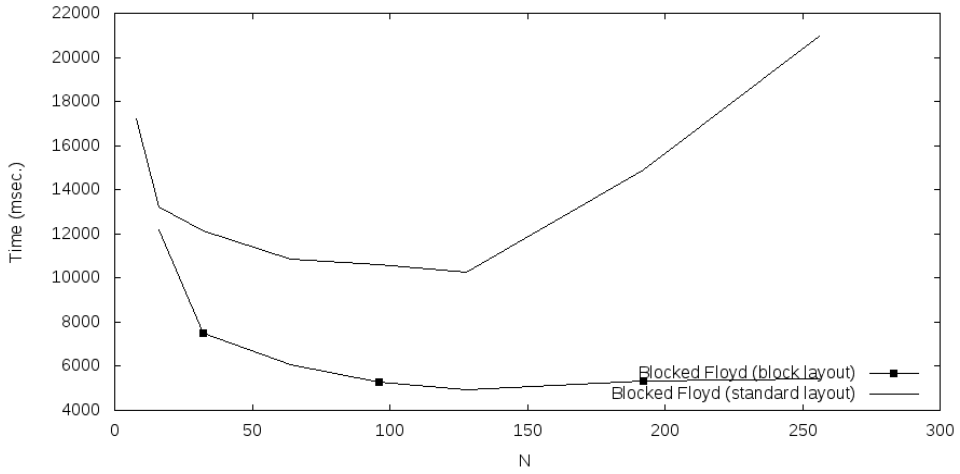

Рис. 23: График сравнения производительности исходной программы и преобразованной программы. По вертикальной оси отложено время работы программы, по горизонтальной оси – размер блока

По горизонтальной оси отложен размер блоков, на которые разбивается входная матрица при блочном ее размещении. Как видно из результатов эксперимента блочное размещение входной матрицы в данном эксперименте позволило ускорить входную программу. Оптимальный размер блока находится в диапазоне – 120-130.

Результаты численных экспериментов подтверждают, что реализованное автором преобразование блочного размещения данных позволяет существенно увеличить производительность результирующей программы. Оно может использоваться в качестве вспомогательного преобразования для оптимизации блочных программ.

4.8. Реализация парсера языка ФОРТРАН для системы ОРС

Для системы ОРС [75] автором был разработан парсер языка ФОРТРАН. Парсер поддерживает стандарт ФОРТРАН 77, а также включает некоторые конструкции работы с динамической памятью из языка ФОРТРАН 90/95. Парсер полностью разбирает пакет численных методов ACELAN [1] и POM (Princeton Ocean Model) [103]. Корректность работы парсера проверялась также на тестах из GCC [85], Parawise [37], ParcBench [90]. Разработанный парсер может быть использован в ОС Windows и Linux.

При разработке парсера языка ФОРТРАН использовался генератор парсеров ANTLR. В качестве исходной грамматики была взята грамматика языка ФОРТРАН из проекта Open Fortran Parser [91].

На реализацию парсера большое влияние оказало внутреннее представление ОРС. Оно имеет структуру близкую к языку Си, а не ФОРТРАН, и в связи с этим возникли проблемы, связанные с отображением программ на языке ФОРТРАН во внутреннее представление. Рассмотрим примеры решения некоторых таких проблем.

Пример 14. Отображение COMMON-блоков во внутреннее представление. Фрагмент кода.

Листинг 23. Объявление common-блока SampleCommonBlock внутри функций A и B.

```
SUBROUTINE A
COMMON / SampleCommonBlock / iVar1_1, iVar1_2
END SUBROUTINE A
SUBROUTINE B
COMMON / SampleCommonBlock / iVar2_1, iVar2_2
END SUBROUTINE B
```

отображается во внутреннее представление следующим образом:

Листинг 24. Результат преобразования common-блока в язык Си

```
struct SampleCommonBlock {
union A { int ivar1_1, ivar1_2 }
union B { int ivar2_1, ivar2_2 }
}

SampleCommonBlock block_var;
void A()
{
...
}

void B()
{
...
}
```

Пример 15. Проблема отображения срезов массивов. В языке ФОРТРАН есть поддержка обращений к срезам массивов, в то время как в языке Си такой возможности нет. В парсере реализована ограниченная поддержка таких операций. Он разбирает операторы присваивания вида

Листинг 25. Оператор присваивания нескольким элементам массива X некоторого значения

```
X[x1:y1:k1,x2..y2] = expr;
```

в гнезда циклов вида

Листинг 26. Результат преобразования в язык Си

```
for (i = x1; i<=y1; i+=k1)
 for (j = x2; j<=y2, j+=1)
   X[i][j] = expr;
```

4.9. Выводы к четвертой главе

В **четвертой главе** приводится алгоритм автоматизации блочных размещений данных, реализованный в системе ОРС. Такая автоматизация реализована на уровне директив компилятора языка Си. Автором реализован также парсер языка ФОРТРАН 77/90, который был протестирован на пакетах ACELAN и POM. Реализованный парсер языка ФОРТРАН позволяет использовать систему ОРС для оптимизации программ, написанных на языке ФОРТРАН. Таким образом, в данной главе раскрывается решение **задачи 3** и **задачи 4**.

Описание директив блочного размещения данных приведено в **параграфе 4.2.** Описанные директивы позволяют блочно размещать также массивы произвольной размерности. В **параграфах 4.3-4.4** приводятся программные

проекты, в которых можно протестировать работу данных директив. В **параграфах 4.5-4.5** строится алгоритм обработки компилятором директив блочного размещения данных. **Параграф 4.7** содержит результаты численных экспериментов, в которых сравнивается время работы программ, скомпилированных в двух режимах – без директив блочного размещения данных и без них. Результаты численных экспериментов подтверждают, что реализованная автоматизация блочных размещений дает дополнительное ускорение для некоторых классов задач.

Заключение

В диссертации проведены исследования, направленные на оптимизацию работы с памятью существующих вычислительных систем.

Построенная модель времени выполнения программ позволяет оценивать время работы программ. В ней учитываются частота процессора, а также различные характеристики иерархии кеш-памяти, такие как латентность, физический размер, ассоциативность. Учет этих параметров позволяет прогнозировать более точно время выполнения программ, что было подтверждено численными экспериментами в **параграфах 2.1-2.2**. Таким образом, описанная в **главе 2** модель времени выполнения программ является решением выносимого на защиту **положения 3**.

В **третьей главе** описан и реализован высокопроизводительный алгоритм умножения матриц. Этот алгоритм может использоваться для ускорения алгоритмов линейной алгебры, которые сводятся к задаче умножения матриц (например, QR разложение матрицы).

Описанный алгоритм является новым. Его новизна состоит в использовании нестандартного иерархического блочного размещения матриц. Такое размещение матриц позволяет уменьшить количество кеш-промахов и тем самым увеличить общую производительность алгоритма. Таким образом, в данной главе раскрывается решение **положения 1**, выносимого на защиту.

В **четвертой главе** приводится алгоритм автоматизации блочных размещений данных, реализованный в с системе ОРС. Такая автоматизация реализована на уровне директив компилятора языка Си. Реализована также поддержка блочных размещений для программ, написанных на языке Фортран. С целью внедрения автоматического блочного размещения данных в программах, написанных на языке ФОРТРАН, автором был реализован парсер языка ФОРТРАН 77/90 в системе ОРС, который был апробирован на пакете ACELAN.

Этим самым, в данной главе раскрывается решение **положения 2**, выносимого на защиту.

И так, все поставленные в диссертации задачи решены, и цель диссертации достигнута.

Результаты диссертации внедрены в научных исследованиях (при разработке Оптимизирующей распараллеливающей системы мехмата Южного федерального университета) и в образовании (в магистерской программе «Высокопроизводительные вычисления и технологии параллельного программирования»).

Можно выделить следующие **рекомендации** по применению полученных результатов:

1. Предлагаемый алгоритм умножения матриц может использоваться для ускорения работы существующих пакетов прикладных программ, написанных для процессоров с поддержкой AVX, и использующих умножение матриц в качестве подзадачи. При этом программист должен учитывать, что подаваемые на вход матрицы должны храниться в памяти блочно.
2. Директивы, реализованные в компиляторе языка Си системы ОРС, позволяющие автоматически блочно размещать матрицы, могут использоваться для ускорения программ.
3. Полученная модель времени выполнения может использоваться для прогнозирования времени работы некоторых работающих с матрицами программ и может использоваться для построения подобных моделей для других случаев.

Перспективы дальнейшей работы таковы:

- Адаптация полученных в диссертации методов оптимизации к вычислительным архитектурам близкого будущего и исследование влияния этих методов на производительность оптимизируемых программ.

- Применение изложенных методов оптимизации программ к другим задачам, требующим большого объема вычислений.

Список сокращений и условных обозначений

OPS – Optimizing parallelization system

ОРС – оптимизирующая распараллеливающая система

ДВОР – диалоговый высокоуровневый оптимизирующий распараллеливатель

AVX – Advanced Vector Extensions – набор векторных команд, оперирующих с 256-битными регистрами

LLVM – Low Level Virtual Machine

DVM – Distributed Virtual Memory

ICC – Intel C Compiler

GCC – GNU C Compiler,

MSVC – Microsoft Visual C++

SUIF – Stanford University Intermediate Format

TLB – Translation Lookaside Buffer

LRU – Least Recently Used

MRU – Most Recently Used

Литература

1. ACELAN. [electronic resource]. — URL: http://www.math.rsu.ru/mexmat/mathmodel/ac/index.php. (online;accessed:2015-12-10).

2. Charles, V.L. A Block QR Factorization Scheme for Loosely Coupled Systems of Array Processors [Text] / V.L. Charles, M. Schultz // Numerical Algorithms for Modern Parallel Computer Architectures. — NY: Springer US, 1986. — Vol. 13, — P. 217-232.

3. Charles, V.L. The WY representation for products of householder matrices [Text] / Ch. Bischof, V.L. Charles. // Computer science technical report, Cornell University, — 1985.

4. Denning, P.J. The Locality Principle [Text] / P.J. Denning // Communications of the ACM - Designing for the mobile device. — 2005. — Vol. 48, no 7, — P. 19–24.

5. Gustavson, F.G. Cache Blocking for Linear Algebra Algorithms [Text] / F. G. Gustavson // PPAM 2011: Parallel Processing and Applied Mathematics: 9th International Conference, September 11-14, 2011. Revised Selected Papers, Part I — Torun, Poland: Springer Berlin Heidelberg. — 2012. — P. 122–132.

6. Lam, M.S. The cache performance and optimizations of blocked algorithms [Text] / Monica S. Lam, Edward E. Rothberg, Michael E. Wolf // Proceeding ASPLOS IV Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. —1991. — P. 63-74.

7. Prokop, H. Cache-oblivious algorithms [Text] / M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran // Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS’99). — 1999. — P. 285–297.

8. Gustavson, F.G. Is Cache-Oblivious DGEMM Viable? [Text] / A. G. Joh, F. G. Gustavson, K. Pingali, K. Yotov // Proceedings of the 8th international conference on Applied parallel computing. Springer-Verlag — Berlin, Heidelberg 2007. ISBN:3-540-75754-6 978-3-540-75754-2. — 2007. — P. 919-928.

9. Larus, J. Improving Pointer-Based Codes Through Cache-Conscious Data Placement [Text] / T. Chilimbi, J. Larus, M. D. Hill // Wisconsin University Technical Report CS-TR-98-1365. —1998.

10. Chatterjee, S. Nonlinear Array Layouts for Hierarchical Memory Systems [Text] / S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra, and M. Thottethodi // Proc. 13th ACM Int'l Conf. Supercomputing. — 1999. — P. 444-453.

11. Prasanna, V. K. Cache Conscious Walsh-Hadamard Transform [Text] / N. Park, V. K. Prasanna // The 26th International Conference on Acoustics, Speech, and Signal Processing. — 2001. — Vol. 2. — P. 1205-1208.

12. Jeyarajan, T. Alternative Array Storage Layouts for Regular Scientific Programs [Text] / T. Jeyarajan // University of London Imperial College London Department of Computing. — 2005.

13. Manjikian, N. Array Data Layout for the Reduction of Cache Conflicts [Text] / N. Manjikian, T. Abdelrahman // department of Electrical and Computer Engineering. In Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems — 1995.

14. Karlsson, L. Blocked In-Place Transposition with Application to Storage Format Conversion [Text] / Lars Karlsson // Technical Report UMINF 09.01, Dept. of Computing Science, Umee University, Sweden. — 2009.

15. Prasanna, V. K. Efficient Matrix Multiplication Using Cache Conscious Data Layouts [Text] / N. Park, W. Liu, V. K. Prasanna, C. Raghavendra // The DoD HPCMO Users Group Conference — 2000.

16. Prasanna, V. K. Tiling, Block Data Layout, and Memory Hierarchy Performance [Text] / N. Park, B. Hong, V. K. Prasanna // IEEE Transactions on Parallel and Distributed Systems — Vol. 14, no. 7 — 2003. — P. 640-654.

17. Zuckerman, S. Fine Tuning Matrix Multiplications on Multicore [Text] / S. Zuckerman, M. Perache, W. Jalby //High Performance Computing - HiPC 2008, Lecture Notes in Computer Science — Vol. 5374 — 2008. — P. 30-41.

18. Herrero, J.R. Using Non-canonical Array Layouts in Dense Matrix Operations [Text] / J. R. Herrero, J. J. Navarro // Applied Parallel Computing. State of the Art in

Scientific Computing Lecture Notes in Computer Science. — Vol. 4699. — 2007. — P. 580-588.

19. Prasanna, V.K. Analysis of Memory Hierarchy Performance of Block Data Layout [Text] / N. Park, B. Hong, V. K. Prasanna // International Conference on Parallel Processing. — 2002. — P. 35 - 44.

20. Frens, J. D. Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code [Text] / J. D. Frens and D. S. Wise. // Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — Las Vegas, NV, June 1997. — P. 206–216.

21. Goodchild, M. F. Optimizing raster storage: an examination of four alternatives [Text] / M. F. Goodchild, A. W. Grandfield // Proceedings Auto Carto 6 Ottawa, Oct. 1983. — P. 400–407.

22. Jagadish, H. V. Linear clustering of objects with multiple attributes [Text] / H. V. Jagadish. // Proceedings of the 1990 ACM SIGMOD international conference on Management of data / ACM. — New York, NY, USA. — 1990. — P. 332-342.

23. Akin, B. FFTs with near-optimal memory access through block data layouts [Text] / B. Akin, F. Franchetti, J. C. Hoe // Proceedings of IEEE International Conference Acoustics Speech and Signal Processing. — 2014. — P. 3898-3902.

24. Gustavson, F. G. Rectangular full packed format for LAPACK algorithms timings on several computers [Text] / F. G. Gustavson, J. W. Sniewski // Applied Parallel Computing, State of the Art in Scientific Computing, PARA 2006. — Vol. LNCS 4699. —Springer-Verlag, Berlin Heidelberg, 2007. — P. 570–579.

25. Geijn, R. Parallel out-of-core computation and updating of the QR factorization [Text] / Robert A. Van De Geijn, B. C. Gunter, Robert A. Van De Geijn // Journal ACM Transactions on Mathematical Software (TOMS). — Vol. 31, no. 1. —2005. — ACM New York, NY, USA. — P. 60-78.

26. Goto, K. Anatomy of High-Performance Matrix Multiplication [Text] / K. Goto, R. A. van de Geijn // ACM Transactions on Mathematical Software. — Vol. 34, no. 3. — 2008. P. 1-25.

27. Dongarra, J. Programming the LU Factorization for a Multicore System with Accelerators [Text] / J. Kurzak, P. Luszczek, M. Faverge, J. Dongarra // High Performance Computing for Computational Science (VECPAR 2012), Lecture Notes in Computer Science. — Vol. 7851. — 2013. — P. 28-35.

28. PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, University of Tennessee [Electronic resource]. — URL: http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf (online;accessed:2015-12-10).

29. ATLAS (Automatically Tuned Linear Algebra Software). [Electronic resource]. — URL: http://math-atlas.sourceforge.net/ (online;accessed:2015-12-10).

30. OPENBLAS . [Electronic resource]. — URL: http://www.openblas.net/ (online;accessed:2015-12-10).

31. Kennedy, K. Optimizing compilers for modern architectures: a dependence-based approach / K. Kennedy, J. R. Allen // San Francisco, CA: Morgan Kaufmann Publishers Inc. — 2001.

32. Muchnik, S. Advanced compiler design and implementation / Muchnik S. // San-Francisco, CA, USA: Morgan-Kaufmann. — 1997.

33. Sharma, K. User-Specified and Automatic Data Layout Selection for Portable Performance / K. Sharma, I. Karlin, J. Keasler, J. R McGraw, V. Sarkar. // Rice University Technical Report (TR13-03), LLNL Technical Report TR-637873. — 2013.

34. Keasler, J. TALC: A Simple C Language Extension For Improved Performance and Code Maintainability [Text] / J. Keasler, T. Jones, D. Quinlan // 9th LCI International Conference on High-Performance Clustered Computing - Urbana, IL, May — 2008.

35. Liu, J. Data layout optimization for GPGPU architectures [Text] / J. Liu, W. Ding, O. Jang, M. Kandemir // PPoPP '13 Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming. — ACM New York, NY, USA. — 2013. — P. 283-284. — ISBN: 978-1-4503-1922-5.

36. Liu, J. Compiler Framework for Extracting Superword Level Parallelism [Text] /J. Liu, Y. Zhang, O. Jang, W. Ding, M. Kandemir // A Proceeding PLDI '12

Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation / ACM. — New York, NY, USA. — 2012. — P. 347-358.

37. ParaWise Automatic Parallelization Environment. [Electronic resource]. — URL: http://www.parallelsp.com/parawise.htm (online;accessed:2015-12-10).

38. Intel C++ Composer XE 2013 SP1 for Linux, Update 3. [Electronic resource]. — URL: https://software.intel.com/en-us/articles/intel-c-composer-xe-2013-sp1-for-linux-update-3. (online;accessed:2015-12-10).

39. Fortran DVM system. [Electronic resource]. — URL: http://www.keldysh.ru/dvm/ (online;accessed:2015-12-10).

40. Компилятор SUIF. [Electronic resource]. — URL: http://suif.stanford.edu/suif/suif2/ (online;accessed:2015-12-10).

41. Lam, S. M. A data locality optimizing algorithm [Text] / M. E. Wolf, M. S. Lam // Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. — ACM New York, NY, USA. — 1991. — P. 30–44. — ISBN:0-89791-428-7.

42. Lam, S. M. SUIF: An infrastructure for research on parallelizing and optimizing compilers [Text] / R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. S. Lam, and J. Hennessy // ACM SIGPLAN Notices. — ACM New York, NY, USA. — 1994. — Vol. 29, no. 12. — P. 31 - 37.

43. Bondhugula, U. Practical Automatic Polyhedral Parallelizer and Locality Optimizer [Text] / U. Bondhugula, A. Hartono, J. Ramanujan, P. Sadayappan // Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation. — ACM New York, NY, USA. — 2008. — P. 101-113. — ISBN: 978-1-59593-860-2.

44. Rose Compiler. [Electronic resource]. — URL: http://rosecompiler.org/ (online;accessed:2015-12-10).

45. PGI Optimizing Compilers. [Electronic resource]. — URL: http://www.pgroup.com/. (online;accessed:2015-12-10).

46. Datta, K. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures [Text] / K. Datta, M. Murphy, V. Volkov, S. Williams, J.

Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick // Proceeding SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing. — Article no. 4. — IEEE Press Piscataway, NJ, USA. — 2008. — ISBN: 978-1-4244-2835-9.

47. Dursun, H. In-Core Optimization of High-order Stencil Computations [Text] / H. Dursun, K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, P. Vashishta. — 2009. — P. 533-538.

48. Henretty, T. Data layout transformation for stencil computations on short-vector SIMD architectures [Text] / T. Henretty, K. Stock, L. Pouchet, F. Franchetti, J. Ramanujam, P. Sadayappan // Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software. — P. 225-245.

49. Drepper, U. What Every Programmer Should Know About Memory [Text] / Ulrich Drepper // Red Hat, Inc. — 2007.

50. Касперский, К. Техника оптимизации программ [Текст]. Эффективное использование памяти. — СПб.: БХВ-Петербург. — 2003.

51. Intel-64 and IA-32 Architectures Optimization Reference Manual // Intel corp., 2011, . [Electronic resource]. — URL: http://www.intel.com/products/processor/manuals/. (online;accessed:2015-12-10).

52. Intel Itanium Architecture Software Developer's Manual // Intel corp., 2005, . [electronic resource]. — URL: http://www.intel.com/design/itanium/manuals/iiasdmanual.htm (online;accessed:2015-12-10).

53. Intel VTune Amplifier XE 2013 . [electronic resource]. — URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe (online;accessed:2015-12-10).

54. AMD Catalyst Software home page [electronic resource]. — URL: http://www.amd.com/en-gb/innovations/software-technologies/catalyst (online;accessed:2015-12-10).

55. Valgrind home page [electronic resource]. — URL: http://valgrind.org/ (online;accessed:2015-12-10).

56. John L. Hennessy, David A. Patterson. Computer Architecture, Fifth Edition: A Quantitative Approach. The Morgan Kaufmann Series in Computer Architecture and Design. 2011

57. Markov, I. Limits on Fundamental Limits to Computation [Text] / I. Markov // Nature 512. — 2014. — P. 147-154.

58. Intel Sandy Bridge [electronic resource]. — URL: http://ru.wikipedia.org/wiki/Sandy_Bridge (online;accessed:2015-12-10).

59. Intel Haswel [electronic resource]. — URL: https://ru.wikipedia.org/wiki/Haswell (online;accessed:2015-12-10).

60. Cortex M0 Devices Generic User Guide home page [electronic resource]. — URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf (online;accessed:2015-12-10).

61. Реализация быстрого преобразования Фурье [электронный ресурс]. — URL: http://paulbourke.net/miscellaneous/dft/ (дата обращения: 2015-12-10).

62. Юрушкин, М.В. Новым процессорам — новые компиляторы [Текст] / М.В. Юрушкин, Б.Я. Штейнберг // Открытые системы. СУБД. — 2013. — Т. 1. — С. 55-58. — ISSN 1028-7493.

63. Юрушкин, М.В. Автоматизация распараллеливания программ с блочным размещением данных [Текст] / Л.Р. Гервич, Е.Н. Кравченко, Б.Я. Штейнберг, М.В. Юрушкин // Сибирский журнал вычислительной математики. — 2015. — Т. 18, №1. — С. 41-53. — **DOI:** 10.1134/S1995423915010012.

64. Юрушкин, М.В. Автоматизация блочного размещения данных в памяти компилятором языка Си [Текст] / М. В. Юрушкин // Программная инженерия. — 2013. — С. 355-358. — ISSN 2220-3397.

65. Юрушкин, М. В. Предметно-ориентированные технологии создания виртуальных рабочих пространств в среде облачных вычислений Clavire [Текст] / А. В. Духанов, Е. В. Болгова, Л. Р. Гервич, В. Г. Колпаков, Е. Н. Кравченко, И. И. Курочкин, Е. Д. Масленников, И. В. Офёркин, А. О. Рубцов, С. А. Смирнов, О. Б.

Штейнберг, М. В. Юрушкин // Известия ВУЗов. Приборостроение. — Т. 5. — 2013.

66. Юрушкин, М.В. Диалоговый высокоуровневый автоматический распараллеливатель (ДВОР) [Текст] / Б.Я. Штейнберг, А.А. Абрамов, Е.В. Алымова, А.П. Баглий, С.А. Гуда, Д.В. Дубров, Е.Н. Кравченко, Р.И. Морылев, З.Я. Нис, В.В. Петренко, С.В. Полуян, И.С. Скиба, В.Н. Шаповалов, О.Б. Штейнберг, Р.Б. Штейнберг // Научный сервис в сети Интернет: Труды Всероссийской суперкомпьютерной конференции (20-26 сентября 2010 г., г. Новороссийск) . — М.: Изд-во МГУ. — 2010. — С. 71-75.

67. Юрушкин, М.В. Распараллеливание и оптимизация программ с помощью Web-ускорителя [Текст] / Е.В. Алымова, Е.Н. Кравченко, Р.И. Морылев, Б.Я. Штейнберг, М.В. Юрушкин // Труды Международной суперкомпьютерной конференции «Научный сервис в сети Интернет»: (17-22 сентября 2012 г., г. Новороссийск). — М.: Изд-во МГУ. — 2012. — С. 612-618.

68. Юрушкин, М.В. Реализация алгоритма распределения данных под общую и распределенную память в системе ДВОР [Текст] / М.В. Юрушкин // Сборник трудов XIV молодежной конференции-школы с международным участием. Ростов-на-Дону, издательство Южного федерального университета. — 2011. — С. 396-400.

69. Юрушкин, М.В. Модель времени вычислений [Текст] / Б.Я. Штейнберг, М.В. Юрушкин // Тезисы выступлений МСКФ’14. — 2014, http://www.ospcon.ru/node/107941.

70. Юрушкин, М.В. Программирование экзафлопсных систем [Текст] / Л.Р. Гервич, Б.Я. Штейнберг, М.В. Юрушкин // Открытые системы. СУБД. — Т. 8. — 2013. — С. 26-29. — ISSN 1028-7493.

71. Юрушкин, М.В. Разработка параллельных программ с оптимизацией использования структуры памяти [Текст] / Л.Р. Гервич, Б.Я. Штейнберг, М.В. Юрушкин // Ростов-на-Дону, изд-во Южного федерального университета. — 2014. — 120 с.

72. Юрушкин, М.В. Web-ориентированный автоматический распараллеливатель программ [Текст] / Б.Я. Штейнберг, А.Н. Аллазов, Е.В. Алымова, А.П. Баглий, С.А. Гуда, Д.В. Дубров, Е.Н. Кравченко, Р.И. Морылев, А.С. Рошаль, М.В. Юрушкин, Р.Б. Штейнберг // Труды международной научной конференции ПаВТ'14. — Издательский центр ЮУрГУ. — 2014. — С. 380-380.

73. Юрушкин, М.В. Автоматизация блочного размещения данных в оперативной памяти компилятором языка Си [Текст] / М.В. Юрушкин // Труды международной научной конференции ПаВТ'14. — Издательский центр ЮУрГУ. — 2014. — С. 355-358.

74. Юрушкин, М.В. Двойное блочное размещение данных в оперативной памяти при решении задачи умножения матриц [Текст] / М.В. Юрушкин // Программная инженерия. — 2016. — С. 132-139.

75. Оптимизирующая распараллеливающая система [Электронный ресурс]. — URL: www.ops.rsu.ru (Дата обращения:2015-12-10).

76. Штейнберг, Б.Я. Оптимизация размещения данных в параллельной памяти [Текст] / Б.Я. Штейнберг // Ростов-на-Дону. — изд-во Южного федерального университета. — 2010. — С. 255.

77. Автоматический распараллеливатель программ с web-интерфейсом [Электронный ресурс]. http://ops.opsgroup.ru/opsweb-datadistr.php (Дата обращения:2015-12-10).

78. Лиходед, Н.А. Обобщенный тайлинг [Текст] / Н.А. Лиходед // Доклады НАН Беларуси — 2011. — Т.55. №1. — С. 16-21.

79. Программа перемножения матриц рекордной производительности. [Электронный ресурс]. — URL: http://ops.opsgroup.ru/downloads/dgemm.zip (дата обращения:2015-12-10).

80. Agner Fog, Optimizing subroutines in assembly language: An optimization guide for x86 platforms. — URL: http://www.agner.org/optimize/optimizing_assembly.pdf . (online;accessed:2015-12-10).

81. Grosser, T. Polly-Polyhedral optimization in LLVM [Text] / T Grosser, H Zheng, R Aloor, A Simbürger, A Größlinger, LN Pouchet // Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT).

82. Polly LLVM [electronic resource]. — URL: http://polly.llvm.org/ (online;accessed:2015-12-10).

83. The LLVM Compiler Infrastructure [electronic resource]. — URL: http://llvm.org/ (online;accessed:2015-12-10).

84. Intel MKL [electronic resource]. — URL: http://software.intel.com/en-us/intel-mkl (online;accessed:2015-12-10).

85. GCC home page [electronic resource] — URL: http://gcc.gnu.org/ (online;accessed:2015-12-10).

86. GotaBLAS home page [electronic resource]. — URL: http://c2.com/cgi/wiki?GotoBlas (online;accessed:2015-12-10).

87. PLUTO. An automatic loop nest parallelizer for multicores. http://pluto-compiler.sourceforge.net/ (online;accessed:2015-12-10).

88. Quinlan, D. The rose source-to-source compiler infrastructure [Text] / D. Quinlan, C. Liao // Cetus users and compiler infrastructure workshop. — 2011.

89. Quinlan, D. A Tool for Building Source-to-Source Translators [Text] / D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, Q. Yi // ROSE User Manual:, July 8. — 2013.

90. Gunther, N. J. PARCbench: A Benchmark for Shared Memory Architectures [Text] / N. J. Gunther, M. T. Noga // Computer Architecture News / ACM. — Т. 17. — 1989. — С. 54-61.

91. Open Fortran Parser (OFP) home page [electronic resource]. — URL: http://fortran-parser.sourceforge.net/ (online;accessed:2015-12-10).

92. Фаддеев, Д.К. Параллельные вычисления в линейной алгебре [Текст] / В.Н. Фаддеева, Д.К. Фаддеев // Кибернетика. — №. 6. — 1977. — С 28-40.

93. Фаддеев, Д.К. Параллельные вычисления в линейной алгебре 2 [Текст] / В.Н. Фаддеева, Д.К. Фаддеев // Кибернетика. — №. 3. — 1982. — С 19-44.

94. Ахо, А. Построение и анализ вычислительных алгоритмов [Текст] / А. Ахо, Дж. Хопкрофт, Дж. Ульман // М.: Мир. — 1979. — С. 536.

95. Strassen, V. Gaussian Elimination is not Optimal [Text] / V. Strassen // Numerische Mathemetik, Bd. 13. — 1969. — P. 354–356.

96. Winograd, Sh. Matrix multiplication via arithmetic progressions [Text] / D. Coppersmith and Sh. Winograd // Journal of Symbolic Computation. — 1990. — P. 251-280.

97. Кристофидес, Н. Теория графов [Текст] / Н. Кристофидес // Алгоритмический подход. — М.: Мир. — 1978. — С. 432.

98. Волконский, В.Ю. Метод использования мелкоформатных векторных операций в оптимизирующем компиляторе / В.Ю. Волконский, А.Ю. Дроздов, Е.В. Ровинский //. Информационные технологии и вычислительные системы. — № 3/ — 2004. — С. 63-77.

99. Касьянов, В.Н. Оптимизирующие преобразования программ [Текст] / В.Н. Касьянов // М., Наука. — 1988. — С. 336.

100. Штейнберг, Б. Я. Блочно-рекурсивное параллельное перемножение матриц [Текст] / Б. Я. Штейнберг //Известия ВУЗов. Приборостроение. — Т. 52, №10. — 2009. — С. 33-41.

101. Robinson, S., Toward an Optimal Algorithm for Matrix Multiplication [Text] / S. Robinson // SIAM News, Vol. 38, Num. 9. — 2005.

102. Cohn, H. Group-theoretic algorithms for matrix multiplication [Text] / H. Cohn, R. Kleinberg, B. Szegedy, C. Umans // Foundations of Computer Science. — 2005. — FOCS 2005. 46th Annual IEEE Symposium. — P. 379 – 388.

103. Princeton Ocean Model [electronic resource]. — URL: http://www.ccpo.odu.edu/POMWEB/ (online;accessed:2015-12-10).

104. Visual C++ Compiler [electronic resource]. — URL: http://www.microsoft.com/en-us/download/details.aspx?id=41151 (online;accessed:2015-12-10).