Федеральное государственное автономное образовательное учреждение
высшего образования «Уральский федеральный университет имени
первого Президента России Б.Н. Ельцина»

На правах рукописи

ГАРЕЕВ Роман Альбертович

МЕТОДЫ ОПТИМИЗАЦИИ ВЫПОЛНЕНИЯ ТЕНЗОРНЫХ ОПЕРАЦИЙ НА МНОГОЯДЕРНЫХ ПРОЦЕССОРАХ

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

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
АКИМОВА Елена Николаевна
доктор физ.-мат. наук, доцент

Екатеринбург – 2020

Оглавление

Введение 5

Глава 1. Алгоритмы сокращения времени выполнения тензорных операций 21

1.1. Обзор методов сокращения времени выполнения матричных и матрично-векторных произведений 22

1.2. Модель гипотетического процессора 27

1.3. Обзор методов сокращения времени выполнения тензорных сверток 33

1.4. Выводы 37

Глава 2. Алгоритм автоматического сокращения времени выполнения тензорных операций 38

2.1. Модификация модели гипотетического процессора. Матрично-векторное произведение. Общая задача о путях 39

2.2. Алгоритм вычисления обобщенного матричного произведения 50

2.3. Алгоритмы вычисления обобщенного матрично-векторного произведения 52

2.4. Алгоритм сокращения времени выполнения свертки тензоров 65

2.5. Выводы 71

Глава 3. Программная система автоматической оптимизации тензорных операций 73

3.1. Архитектура программной системы автоматической оптимизации тензорных операций 74

3.2. Реализация программной системы автоматической оптимизации тензорных операций 76

3.3. Автоматическое распараллеливание программ на многоядерных процессорах общего назначения с общей памятью . . . 78
3.4. Выводы 82

Глава 4. Оценка эффективности разработанных алгоритмов 83
4.1. Обратная задача гравиметрии о восстановлении раздела между средами 84
4.1.1. Постановка и решение задачи гравиметрии 84
4.1.2. Результаты экспериментов 86
4.2. Оценка алгоритма вычисления обобщенного матричного произведения 88
4.2.1. Оценка погрешности векторизации для матричного произведения 90
4.2.2. Результаты экспериментов для матричного произведения. Случай однопоточной производительности . . 96
4.2.3. Результаты экспериментов для матричного произведения. Случай многопоточной производительности . 104
4.2.4. Матричное произведение и целочисленные типы данных. Случай однопоточной производительности . . . 105
4.2.5. Матричное произведение и целочисленные типы данных. Случай многопоточной производительности . . 106
4.2.6. Частные случаи общей задачи о путях. Однопоточная производительность 108
4.2.7. Частные случаи общей задачи о путях. Многопоточная производительность 113
4.3. Оценка алгоритма вычисления обобщенного матрично-векторного произведения 115
4.3.1. Однопоточная производительность 116
4.3.2. Многопоточная производительность 123

4.3.3. Оценка значений параметров N_b и N_r 130
4.3.4. Оценка значений парамета D 131
4.3.5. Оценка значений параметра M_c 134
4.3.6. Оценка значений параметра N_c 138
4.4. Оценка алгоритма сокращения времени выполнения сверток тензоров 142
4.4.1. Однопоточная производительность 143
4.4.2. Многопоточная производительность 146
4.5. Выводы 149

Заключение 152

Литература 155

Приложение 1. Основные обозначения 178

Введение

Актуальность темы исследования. Тензорные операции имеют широкое применение в различных научных дисциплинах. В частности, операция свертывания или свертки тензоров применяется в машинном обучении, спектральных методах и квантовой химии. Тензорные операции используются при решении дифференциальных и интегральных уравнений, применяемых для описания многих прикладных задач. В большинстве случаев трудно получить точное аналитическое решение таких уравнений. После дискретизации дифференциальные и интегральные уравнения сводятся к системам линейных и нелинейных уравнений. Для решения таких систем используются численные методы, применяющие матричные и матрично-векторные произведения, которые могут рассматриваться как частные случаи свертки тензоров.

Для уменьшения времени решения задач на больших сетках используется распараллеливание алгоритмов и многопроцессорные системы. Проблемам исследования и распараллеливания алгоритмов при решении прикладных задач посвящены работы В.В. Воеводина [1], Дж. Ортеги [2], Д.К. Фаддеева, В.П. Ильина, Е.Н. Акимовой [3, 4], Б.Н. Четверушкина, В.П. Гергеля [5], М.В. Якобовского, В.Н. Алеевой, Ю.Я. Болдырева, Л.Б. Соколинского [6], С.М. Абрамова [7], Б.Я. Штейнберга [8], Б.М. Глинского, М.Л. Цымблера [9], В.Э. Малышкина и др.

Существенный вклад в развитие тензорного исчисления внесли Дж. Риччи, Т. Леви-Чивита, А. Эйнштейн, М. Гроссман, В. Фойгт, Я. Схоутен, Л. Витернабен, И.С. Сокольников, П.К. Рашевский, А.П. Широков, В.Ф. Каган, Н.Е. Кочин, Н.Е. Ефимов, И.Н. Векуа, Б.Е. Победря, В.В. Лохин., Ф.И. Федоров, Н.Г. Ченцов, С.Г. Лехницкий, М.П. Шаскольская, Е.Е. Тыртышников, И.В. Оселедец.

Матричные и матрично-векторные произведения могут быть обобщены на замкнутые полукольца [10] для сокращения времени выполнения решений общей задачи о путях (нахождение кратчайших путей, построение минимального остовного дерева, задача о самом широком пути и др.). Обобщенные матричные и матрично-векторные произведения являются частью интерфейса GraphBLAS [11], используемого для создания высокопроизводительных алгоритмов на графах. В частности, обобщенное матричное произведение может применяться совместно с блочной модификацией алгоритма Флойда—Уоршелла и другими алгоритмами для сокращения времени выполнения решений частных случаев общей задачи о путях [12].

Большинство подходов, используемых для сокращения времени выполнения операций над тензорами, основаны на *ручной настройке* (*manual-tuning*) и *автонастройке* (*auto-tuning*), требующих доступа к целевой аппаратной платформе и потенциально больших временных затрат. Для выполнения ручной настройки для отдельно рассматриваемой архитектуры компьютера специалист, обладающий знаниями особенностей используемых аппаратных средств, создает новую реализацию необходимой операции. В случае автонастройки для каждой отдельно рассматриваемой компьютерной архитектуры выполняется перебор значений параметров заранее созданной реализации для измерения времени выполнения и нахождения их наилучших значений. Проблема использования ручной настройки и автонастройки осложняется в случае недоступности целевой аппаратной платформы и ограниченности времени выполнения оптимизации. Примером такой ситуации служат оптимизации, выполняемые промышленными компиляторами по умолчанию во время кросс-компиляции.

Для того чтобы избежать автонастройки и ручной настройки применяются готовые реализации сверток тензоров, в частности, умножения матриц и умножения матриц на векторы. Интерфейс BLAS (Basic Linear

Algebra Subprograms) [13–15] является общепринятым стандартом интерфейса библиотек, содержащих высокопроизводительные реализации операций линейной алгебры. В большинстве случаев интерфейс BLAS своевременно реализуется для новых целевых аппаратных платформ в виде проприетарных библиотек. Для ряда целевых архитектур имеются реализации интерфейса BLAS с открытым исходным кодом.

Автонастройка и ручная настройка необходимы, если интерфейс BLAS неэффективен или неприменим. Примером такой ситуации являются свертки тензоров, не описываемые интерфейсом BLAS. Устаревший метод прямого сведения сверток тензоров к матричным и матрично-векторным произведениям с целью вызова их оптимизированных реализаций требует дополнительного времени выполнения и дополнительной памяти [16, 17]. Интерфейс BLAS неприменим для реализации новых методов сокращения времени выполнения свертки тензоров, в данный момент поддерживающих малое количество целевых аппаратных платформ [16, 17].

Обобщение матричного и матрично-векторных произведений на замкнутые полукольца не могут быть реализованы с помощью интерфейса BLAS [10]. Высокопроизводительные реализации интерфейса GraphBLAS, в частности реализации обобщений матричных и матрично-векторных произведений недоступны для многих платформ [11].

Для увеличения производительности сверток тензоров, в частности матричных произведений, для хранения элементов тензоров можно использовать типы данных меньшего размера, не поддерживаемые интерфейсом BLAS. Такой подход увеличивает количество элементов тензоров, обрабатываемых векторными инструкциями, и уменьшает издержки передачи данных за счет уменьшения точности вычислений. В частности, матричное произведение для 16-битных типов данных используется для обучения нейронных сетей. Процессоры общего назначения могут использоваться вместо графических ускорителей в случае преобладания издержек на передачу

данных [18]. Матричное произведение для 16-битных типов данных реализовано, например, в Intel MKL и ArmPL, но отсутствует во многих других реализациях интерфейса BLAS.

Во всех описанных случаях могут применяться алгоритмы сокращения времени выполнения, использующие модели гипотетических процессоров. Такие методы могут использоваться в процессе компиляции, ограниченной по времени, позволяя создать высокопроизводительные многопоточные реализации программ для решения задач большой размерности без ручной настройки и автонастройки и без доступа к целевой аппаратной платформе.

Оптимизации, использующие модели гипотетических процессоров, могут применяться в процессе компиляции, ограниченной по времени, для автоматического получения высокопроизводительных реализаций тензорных операций для многоядерных процессоров общего назначения (архитекутры x86-64, x86, ppc64le, aarch64 и др.) без доступа к целевой аппаратной платформе.

Степень разработанности темы. В своих работах 1886-1901 гг. Дж. Риччи обобщил векторное исчисление на системы с произвольным числом индексов, создав основу для тензорного исчисления. К середине XX в., благодаря работам Т. Леви-Чивита, А. Эйнштейна, М. Гроссмана, В. Фойгта, Я. Схоутена, Г. Вейля, О. Веблена, Е. Вильсона, Ф. Мурнагана, Э. Картана, Р. Вайценбека, Д. Витали, Дж.Л. Синджа, Т. Томаса, З. Аппеля, Л. Эйзенхарта, Л. Витернабена, А.Дж. Мак.-Коннела, Л. Брилюэна, А. Ляву, тензорное исчисление развилось в эффективный математический аппарат, широко используемый в различных областях науки.

Существенный вклад в развитие тензорного исчисления внесли такие Российские ученые как, например, И.С. Сокольников, П.К. Рашевский, А.П. Широков, В.Ф. Каган, Н.Е. Кочин, Н.Е. Ефимов, И.Н. Векуа, Б.Е. Победря, В.В. Лохин., А.В. Шубников, Ю.И. Сиротин, Н.В. Белов, И.С. Желу-

дов, Ф.И. Федоров, П.В. Бехтерев, Н.Г. Ченцов, С.Г. Лехницкий, М.П. Шаскольская, Е.Е. Тыртышников, И.В. Оселедец. В частности, И.Н. Векуа создал теорию ковариантного дифференцирования в комплексных криволенейных координатах [19]. Б.Е. Победря разработал спектральное представление тензоров, позволившее значительно развить теорию нелинейных тензорных функций [20]. Е.Е. Тыртышников, в частности, рассматривал тензорные аппроксимации матриц, порождённых асимптотически гладкими функциями [21]. И.В. Оселедец предложил новое представление тензоров — TT-формат, использующийся для представления всех операций с тензорами [22].

За последние двадцать лет прогресс во многих научных дисциплинах, таких как квантовая химия и общая теория относительности, основывался на моделировании физических систем, применяющем свертку тензоров. Несмотря на это, количество подходов к сокращению времени выполнения свертки тензоров намного меньше, чем количество подходов к сокращению времени выполнения матричного и матрично-векторного произведений. Большинство методов сокращения времени выполнения сверток тензоров используют оптимизированные матричные и матрично-векторные произведения.

Специалисты в области приложений линейной алгебры одними из первых предложили использовать единый интерфейс к высокопроизводительным фундаментальным операциям, реализованным как стороннее программное обеспечение. Более сорока лет таким интерфейсом является BLAS [13–15]. Описываемые BLAS подпрограммы разделяются на три уровня. Первый уровень описывает операции над векторами. На втором уровне интерфейса BLAS содержится описание операций над матрицами и векторами. Третий уровень описывает операции над матрицами. Интерфейс BLAS продолжает развиваться. BLAS++ [23], обновленная версия интерфейса BLAS,

используется в проектах, направленных на достижение экзафлопсной производительности.

Применение BLAS требует новой реализации фундаментальных операций для каждой новой архитектуры процессора. Реализации матричного и матрично-векторного произведений используются для реализации всех остальных операций второго и третьего уровней интерфейса BLAS, соответственно [24, 25]. В большинстве случаев они создаются экспертами в области высокопроизводительных вычислений, что требует значительных временных затрат. Реализации фундаментальных операций распространяются как открытое программное обеспечение (GotoBLAS [26,27], OpenBLAS [28], BLIS [29]) и как проприетарные библиотеки, выпускаемые производителями аппаратного обеспечения (Intel MKL [30], IBM's ESSL [31], ArmPL [32]).

Созданию высокопроизводительных реализаций матричного произведения с использованием ручной настройки посвящены работы K. Goto [26, 27], J. Gunnels [33], R. Agarwal [34], R. Whaley [35]. В работах S.A. Hassan [36] и J. Liang [37] были предложены эффективные реализации матрично-векторного произведения. Для ускорения выполнения матричного произведения разрабатываются способы хранения матриц в памяти [38,39]. В частности, в работе [38] предложено двойное блочное размещение матриц, позволяющее уменьшить количество промахов к данным кэш-памяти и добиться производительности 97% от пиковой.

Для сокращения времени разработки высокопроизводительных реализаций BLAS R.C. Whaley [40] и J. Bilmes [41] предложили использование автонастройки. D.G. Spampinato [42] и G. Belter [43] исследовал применение автонастройки совместно с аналитическими техниками с целью определения наилучшей оптимизации. Q. Wang [44] рассмотрел использование автонастройки совместно с эвристическими алгоритмами векторизации, распределения регистров и планирования команд. P.M. Knijnenburg [45] предложил выполнять дополнительные измерения на целевой аппаратной

платформе для уменьшения размера множества значений исследуемых параметров и сокращения времени автонастройки. K. Yotov [46] предложил способ сокращения пространства поиска автонастройки, используя модели гипотетических процессоров.

В работах T.M. Low [47] и K. Yotov [46] было предложено моделировать потребление ресурсов гипотетического процессора вместо перебора значений параметров программы и тестирования полученных реализаций на целевой аппаратной платформе. J. Demmel [48] и G. Ballard [49] использовали модели гипотетических процессоров для минимизации перемещений данных и, как следствие, уменьшения времени работы приложения и его энергопотребления.

В трудах M. Frigo [50] и K. Yotov [51] рассматриваются *кэш-независимые алгоритмы* (*cache-oblivious algorithms*), позволяющие асимптотически оптимально использовать кэш-память, игнорируя ее отдельные параметры.

A. Heinecke [52] создал алгоритмы сокращения времени выполнения матричного произведения для матриц, размерность которых меньше 80×80 элементов. X. Su [53] предложил эвристический алгоритм оптимизации частей реализации матричного произведения, содержащих векторные инструкции.

S. Hirata [54] предложил выполнять перестановку индексов тензоров с целью их представления в виде матриц и использования оптимизированных реализаций матричного произведения. Модификации метода S. Hirata для распределенных вычислений применяются в библиотеках Tensor Contraction Engine [54], Cyclops Tensor Framework [55], libtensor [56], TiledArray [57,58]. Алгоритм, описанный в работе E. D. Napoli [59], представляет сворачиваемые тензоры как наборы матриц, для которых выполняется матричное произведение. P. Springer [17] и D. Matthews [16] для сокращения издержек, связанных с перестановкой индексов тензоров, предложили использовать

только части реализаций матричного и матрично-векторных произведений, содержащие векторные инструкции.

Для сокращения времени выполнения сверток тензоров E. Apra [60] применил последовательности стандартных оптимизаций циклов. K. Stock [61] разработал алгоритмы векторизации для сверток тензоров малой размерности. Для сокращения времени выполнения сверток тензоров большой размерности W. Ma [62] использовал генератор кода циклов для графических ускорителей.

Цель и задачи исследования. *Целью* данной работы является разработка методов оптимизации времени выполнения тензорных операций без ручной настройки и автонастройки, а также создание программной системы автоматического выполнения оптимизаций и их автоматического распараллеливания во время компиляции на многоядерных процессорах общего назначения. Для достижения этой цели требовалось выполнить следующие *задачи*.

1. Разработать модель гипотетического процессора, которая позволяет сократить время выполнения матрично-векторных операций и их обобщений на замкнутые полукольца с элементами из множества вещественных чисел.

2. Разработать новые алгоритмы выполнения тензорных операций константной сложности относительно размерности тензоров, уменьшающие время выполнения таких операций.

3. Разработать программную система AOTO для автоматической оптимизации времени выполнения тензорных операций и их автоматического распараллеливания при компиляции программ для многоядерных процессоров общего назначения.

4. Провести вычислительные эксперименты, подтверждающие эффективность разработанной программной системы по сравнению с аналогами, использующими ручную настройку и автонастройку.

Научная новизна. Результаты, представленные в диссертации, являются новыми, имеют теоретическую и практическую ценность.

1. Разработана новая модель гипотетического процессора, которая позволяет сократить время выполнения матрично-векторных операций и их обобщений на замкнутые полукольца с элементами из множества вещественных чисел.

2. Разработаны новые алгоритмы выполнения тензорных операций константной сложности относительно размерности тензоров, уменьшающие время выполнения таких операций. С помощью моделирования выполнения представленных алгоритмов на гипотетическом процессоре выведены формулы, позволяющие получить значения параметров алгоритмов выполнения тензорных операций в зависимости от характеристик многоядерных процессоров общего назначения для архитектур x86-64, x86, ppc64le, aarch64. Доказаны утверждения о существовании значений параметров, при которых отсутствует простой конвейера векторных инструкций гипотетического процессора для представленных алгоритмов при возможности мгновенной загрузки данных из памяти на векторные регистры.

3. Разработана программная система AOTO для автоматической оптимизации времени выполнения тензорных операций и их автоматического распараллеливания при компиляции программ для многоядерных процессоров общего назначения. Получена оценка производительности многопоточной программы.

Теоретическая и практическая значимость работы Результаты, изложенные в диссертации, могут быть эффективно использованы при численном решении на многоядерных процессорах общего назначения задач математической физики, математической геофизики, механики, квантовой химии.

Методология и методы исследования. Методологической основой выполненного исследования служат тензорное исчисление и теория

лингвистических основ информатики. Для сокращения времени выполнения тензорных операций, используя модель гипотетического процессора, применялись оптимизации полиэдрального представления программы. Для разработки программной системы автоматической оптимизации времени выполнения тензорных операций применялся метод объектно-ориентированного программирования и технология параллельного программирования OpenMP.

Положения, выносимые на защиту. На защиту выносятся следующие новые научные результаты.

1. Разработана новая модель гипотетического процессора, которая позволяет сократить время выполнения матрично-векторных операций и их обобщений на замкнутые полукольца с элементами из множества вещественных чисел.

2. Разработаны новые алгоритмы выполнения тензорных операций константной сложности относительно размерности тензоров, уменьшающие время выполнения таких операций. Выведены формулы, позволяющие получить значения параметров алгоритмов выполнения тензорных операций в зависимости от характеристик многоядерных процессоров общего назначения для архитектур x86-64, x86, ppc64le, aarch64.

3. Разработана программная система AOTO для автоматической оптимизации времени выполнения тензорных операций и их автоматического распараллеливания при компиляции программ для многоядерных процессоров общего назначения. Получена оценка производительности многопоточной программы, представленной группой полностью вложенных циклов. Автоматическая оптимизация времени выполнения обобщения матричного произведения внедрена в основной код Polly проекта LLVM.

4. С помощью экспериментов при решении обратной задачи гравиметрии, общей задачи о путях, оптимизации матрично-векторных операций и тензорных сверток подтверждена применимость программной системы

АОТО для оптимизации времени выполнения тензорных операций. Показано, что программная система АОТО сопоставима по производительности скомпилированного кода с кодом библиотек Intel MKL, OpenBLAS, BLIS, реализующих матричные и матрично-векторные произведения; с фреймворками TCCG и TBLIS, реализующими свертки тензоров; со специализированным компилятором ICC и существенно превосходит компиляторы общего назначения Clang и GCC.

Степень достоверности результатов. Результаты исследования подтверждаются данными экспериментов, выполненных в соответствии с общепризнанными стандартами.

Апробация результатов исследования. Основные положения диссертационной работы, разработанные модели, методы, алгоритмы и результаты вычислительных экспериментов докладывались автором на следующих международных и всероссийских научных конференциях и семинарах.

1. 46-ая международная молодежная школа-конференция «Современные проблемы математики и ее приложений» (СоПроМат-2015). Екатеринбург, 25—31 января 2015 г. URL: https://sopromat.imm.uran.ru/.

2. 2nd International Workshop on Radio Electronics & Information Technologies (REIT'2017). Yekaterinburg, November 15, 2017. URL: https://reit-rtf.ru/2017b.html.

3. 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). Yekaterinburg, October 25—27, 2019. URL: https://sibircon.ieeesiberia.org.

4. Национальный Суперкомпьютерный Форум (НСКФ-2019). Переславль-Залесский, 26—29 ноября 2019 г. URL: http://2019.nscf.ru.

5. XIV международная конференция «Параллельные вычислительные технологии»(ПаВТ'2020). Пермь, 31 марта—2 апреля 2020 г. URL: http://agora.guru.ru/pavt2020.

Публикации соискателя по теме диссертации. Основные результаты диссертации опубликованы в следующих научных работах:

Статьи в изданиях из перечня ВАК

1. Акимова Е.Н., Гареев Р.А. Аналитическое моделирование матрично-векторного произведения на многоядерных процессорах // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2020. Т. 9, № 1. С. 69–82. DOI: 10.14529/cmse200105.
2. Гареев Р.А. Методы оптимизации обобщенных тензорных сверток // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2020. Т. 9, № 2. С. 19–39. DOI: 10.14529/cmse200202.

Статьи в изданиях, индексируемых в SCOPUS, Web of Science

3. Gareev R., Grosser T., Kruse M. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach // ACM Transactions on Architecture and Code Optimization (TACO). 2018. Vol. 15, no. 3. P. 34:1–34:27.
4. Akimova E.N., Gareev R.A. Algorithm of Automatic Parallelization of Generalized Matrix Multiplication // Proceedings of the 2nd International Workshop on Radio Electronics & Information Technologies (Ekaterinburg, November 15, 2017). CEUR Workshop Proceedings. 2017. Vol. 2005. P. 1–10.
5. Akimova E.N., Gareev R.A., Misilov V.E. Analytical Modeling of Matrix-Vector Multiplication on Multicore Processors: Solving Inverse Gravimetry Problem // (SIBIRCON): 2019 International Multi-Conference on Engineering, Computer and Information Sciences (Novosibirsk, Russia, October 25-27, 2019). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2019. P. 0823–0827. DOI: 10.1109/SIBIRCON48586.2019.8958103.

Статья в издании, индексируемом в РИНЦ

6. Гареев Р.А. Сравнение средств генерации абстрактного синтаксического дерева из полиэдральной модели в библиотеках CLooG и ISL // Труды 46-й Международной молодежной школы-конференции “Современные проблемы математики и ее приложений - 2015”. Институт математики и механики УрО РАН им. Н.Н. Красовского, 2015. С. 200–202.

Все результаты, представленные в данной работе, получены автором лично. Содержание диссертации и основные положения, выносимые на защиту, отражают персональный вклад автора в опубликованные работы. В работах [1, 4, 5] научному руководителю Е.Н. Акимовой принадлежит постановка задачи, Р.А. Гарееву принадлежат все разделы статьи и все остальные результаты. В работе [3] Т. Гроссару (Tobias Grosser) принадлежит постановка задачи, М. Крузу (Michael Kruse) принадлежит часть введения, Р.А. Гарееву принадлежат все остальные результаты и разделы статьи. В работе [5] В.Е. Мисилову принадлежит часть введения, Р.А. Гарееву принадлежат все остальные разделы статьи и все остальные результаты.

Структура и объем работы. Диссертация состоит из введения, четырех глав, заключения и библиографии. В приложении 1 приведены основные обозначения, используемые в диссертации. Объем диссертации составляет 178 страниц, объем библиографии — 176 наименований.

Краткое содержание работы.

Во **введении** обосновывается актуальность темы проведенных исследований и дан обзор публикаций, близких к теме диссертации. Во введении сформулирована цель работы, научная новизна и практическая значимость результатов, кратко изложено содержание работы.

Первая глава посвящена обзору известных подходов сокращения времени выполнения TC (Tensor Contraction). Рассматривается понятие модели гипотетического процессора (ГП). Описываются методы сокращения времени выполнения TC, использующие ручную настройку, автонастройку и моделирование вычислений на ГП. Рассматриваются способы сведения сокращения времени выполнения TC к сокращению времени выполнения MMM (matrix-matrix multiplication) и MVM (matrix-vector multiplication).

Во второй главе представлена новая модель ГП, описывающая инструкции предвыборки и операции из замкнутых полуколец с элементами из множества вещественных чисел. Представлены новые алгоритмы для вычисления обобщений MMM и MVM на замкнутые полукольца. С помощью моделирования выполнения описанных алгоритмов на ГП, выведены формулы, позволяющие получить значения параметров описанных алгоритмов в зависимости от характеристик многоядерных процессоров общего назначения. Разработан оригинальный алгоритм автоматического сокращения времени выполнения тензорных операций.

Третья глава посвящена разработке программной системы автоматической оптимизации тензорных операций (ПС АОТО) на основе моделей, алгоритмов и формул, предложенных во второй главе. ПС АОТО автоматически сокращает времени выполнения тензорных операций без доступа к целевой аппаратной платформе и без выполнения автонастройки и ручной настройки. Приводится описание архитектуры ПС АОТО. Рассматриваются возможные реализации ПС АОТО. Предложен метод автоматического распараллеливания программ ПС АОТО на многоядерных процессорах общего назначения с общей памятью.

Четвертая глава посвящена оценки эффективности ПС АОТО при решении обратной задачи гравиметрии, общей задачи о путях, оптимизации TC, MMM и MVM. Выполнено сравнение с оптимизированными библиотеками, компиляторами и фреймворками.

В **разделе 4.1** в качестве примера приложения алгоритмов рассмотрена оптимизация времени выполнения решения трехмерной структурной обратной задачи гравиметрии о восстановлении раздела между средами по известному скачку плотности и гравитационному полю, измеренному на некоторой площади земной поверхности.

Рассмотрено ускорение реализации решения задачи гравиметрии на основе предложенного автором алгоритма вычисления матрично-векторно-

го произведения с реализациями на основе матрично-векторного произведения оптимизированных библиотек. Эксперименты показали, что в среднем результаты предложенного алгоритма отличаются не более чем на 1% и превосходят библиотеки OpenBLAS и BLIS.

В **разделе 4.2** выполнено сравнение однопоточной и многопоточной производительностей ПС АОТО с библиотеками, содержащими оптимизированную реализацию матричного произведения, и современными компиляторами. Выполнена оценка погрешности автоматической векторизации, выполняемой библиотекой LLVM Core в реализации ПС АОТО для случая МММ.

В результате экспериментов по сравнению однопоточной производительности установлено, что ПС АОТО достигает 1.63-кратного ускорения по сравнению с компилятором ICC [63]; 20-кратного ускорения по сравнению с фронтендом для компиляции Clang [64] и компиляторами GCC [65], IBM XLC [66]; 83.33% производительности рассмотренных библиотек Intel MKL, ARMPL, OpenBLAS и BLIS. Установлено, что ПС АОТО достигает 86.18% многопоточной производительности рассмотренных библиотек Intel MKL, OpenBLAS и BLIS.

В **разделах 4.2.4** и **4.2.5** выполнено сравнение однопоточной и многопоточной производительностей ПС АОТО с компиляторами в случае оптимизации времени выполнения решений общей задачи о путях, применяющих обобщение матричного произведения на замкнутые полукольца с элементами из множества вещественных чисел. Эксперименты показали, что ПС АОТО достигает 85% верхней теоретической границы производительности. Для всех рассматриваемых задач ПС АОТО достигла 1.4-кратного ускорения по сравнению с компилятором ICC и 151-кратного ускорение по сравнению с другими компиляторами.

В **разделе 4.3** выполнено сравнение однопоточной и многопоточной производительностей реализации предложенных автором алгоритмов с биб-

лиотеками, содержащими реализации MVM. В результате экспериментов установлено, что достигается однопоточная и многопоточная производительности библиотек Intel MKL, OpenBLAS и BLIS. Выполнена оценка значений параметров, полученных с использованием формул, выведенных в разделе 2.

В **разделе 4.4** выполнено сравнение однопоточной и многопоточной производительностей ПС АОТО с фреймворками, позволяющими получить оптимизированную реализацию свертки тензоров, и современными компиляторами. В результате экспериментов по сравнению однопоточной производительности установлено, что ПС АОТО достигает 80-кратного ускорения по сравнению с рассмотренным фронтендом для компиляции Clang и компиляторами GCC, ICC; 86.12% однопоточной производительности рассмотренных фреймворков TCCG и TBLIS. Установлено, что ПС АОТО достигает 88.18% многопоточной производительности фреймворка TBLIS.

В **заключении** приводятся основные выводы по теме диссертации, представляются отличия диссертационной работы от ранее выполненных родственных работ других авторов, обсуждаются возможности применения полученных результатов и перспективы дальнейшего развития данного направления.

В **приложении 1** приводятся основные обозначения, используемые в диссертационной работе.

Глава 1

Алгоритмы сокращения времени выполнения тензорных операций

Первая глава посвящена обзору известных подходов сокращения времени выполнения ТС, имеющих широкое практическое применение. Рассматривается понятие модели ГП. Описываются методы сокращения времени выполнения ТС, использующие ручную настройку, автонастройку и моделирование вычислений на ГП. Рассматриваются способы сведения сокращения времени выполнения ТС к сокращению времени выполнения MMM и MVM.

1.1. Обзор методов сокращения времени выполнения матричных и матрично-векторных произведений

MMM и MVM широко применяются при решении прикладных задач. Примером использования MMM служит «обрезка весов», использующаяся в машинном обучении [67–69]. Энергия корреляции молекул, рассматриваемая в квантовой химии, рассчитывается с использованием MMM [70–72]. В качестве примеров применения MVM можно привести обрезку весов, применяемую в машинном обучении [73–75]; анализ графов [76–78]; расчет коэффициентов скорости реакции [79–81].

В силу широкой распространенности и практической значимости сокращение времени выполнения MMM и MVM является отдельным предметом изучения высокопроизводительных вычислений. Операции третьего уровня BLAS могут быть вычислены, используя реализацию MMM и операции второго уровня BLAS [24]. Для вычисления всех операций второго уровня BLAS могут применяться операции первого уровня BLAS и реализация MVM [25]. Оптимизированные MMM и MVM могут применяться для сокращения времени выполнения TC для тензоров большей размерности (см. раздел 1.3). Рассмотрим основные подходы к сокращению времени выполнения MMM и MVM.

Ручная настройка является самым старым и наиболее часто используемым способом сокращения времени выполнения MMM и MVM. В процессе ее выполнения эксперт в области высокопроизводительных вычислений создает новую реализацию программы для каждой требуемой целевой аппаратной платформы при доступе к ней.

Предложенный в работах K. Goto [26, 27] параметризованный алгоритм 1 является примером современного алгоритма выполнения MMM, применяемого в качестве основы для создания новых реализаций BLAS в библиотеках OpenBLAS и BLIS методом ручной настройки. Его основой

служит эффективное использование кэш-памяти и векторных инструкций процессора, являющихся одними из основных ресурсов для достижения высокой производительности программ для многопроцессорных систем [8,82].

Алгоритм 1: Вычисление MMM

```
1  begin
2    for j = 0...N step N_c do
3      for p = 0...K step K_c do
4        Копирование B(p:p + K_c − 1, j:j + N_c − 1) в массив B_c
5        for i = 0...M step M_c do
6          Копирование A(i:i + M_c − 1, p:p + K_c − 1) в массив A_c
7          for j_c = 0...N_c step N_r do
8            for i_c = 0...M_c step M_r do
9              for p_c = 0...K_c do
10               𝕀 = i_c : i_c + M_r − 1 , 𝕁 = j_c : j_c + N_r − 1
11               C_c(𝕀, 𝕁) += A_c(𝕀, p_c) · B_c(p_c, 𝕁)
12             end
13           end
14         end
15       end
16     end
17   end
18 end
```

На вход алгоритма 1 подаются матрицы A, B и C, имеющие размерность $M \times K$, $K \times N$ и $M \times N$, соответственно. Матрица C содержит результат выполнения MMM. В процессе выполнения алгоритма матрицы A и B разбиваются на блоки, как показано на рис. 1.1. Это позволяет многократно использовать элементы блоков, хранящихся в кэш-памяти. Размеры блоков N_c, K_c, M_c, N_r и M_r являются параметрами алгоритма. Их значения могут определяться с помощью автонастройки и ручной настройки.

С целью обеспечения последовательного доступа к операндам MMM в процессе выполнения алгоритма 1 создаются временные выравненные в памяти массивы A_c и B_c, хранящие элементы перемножаемых подматриц размерности $M_c \times K_c$ и $K_c \times N_c$, соответственно. Значения параметров M_c, K_c и N_c выбираются так, чтобы хранимые элементы оставались в кэш-памяти

между двумя последовательными итерациями цикла с индуктивной переменной p.

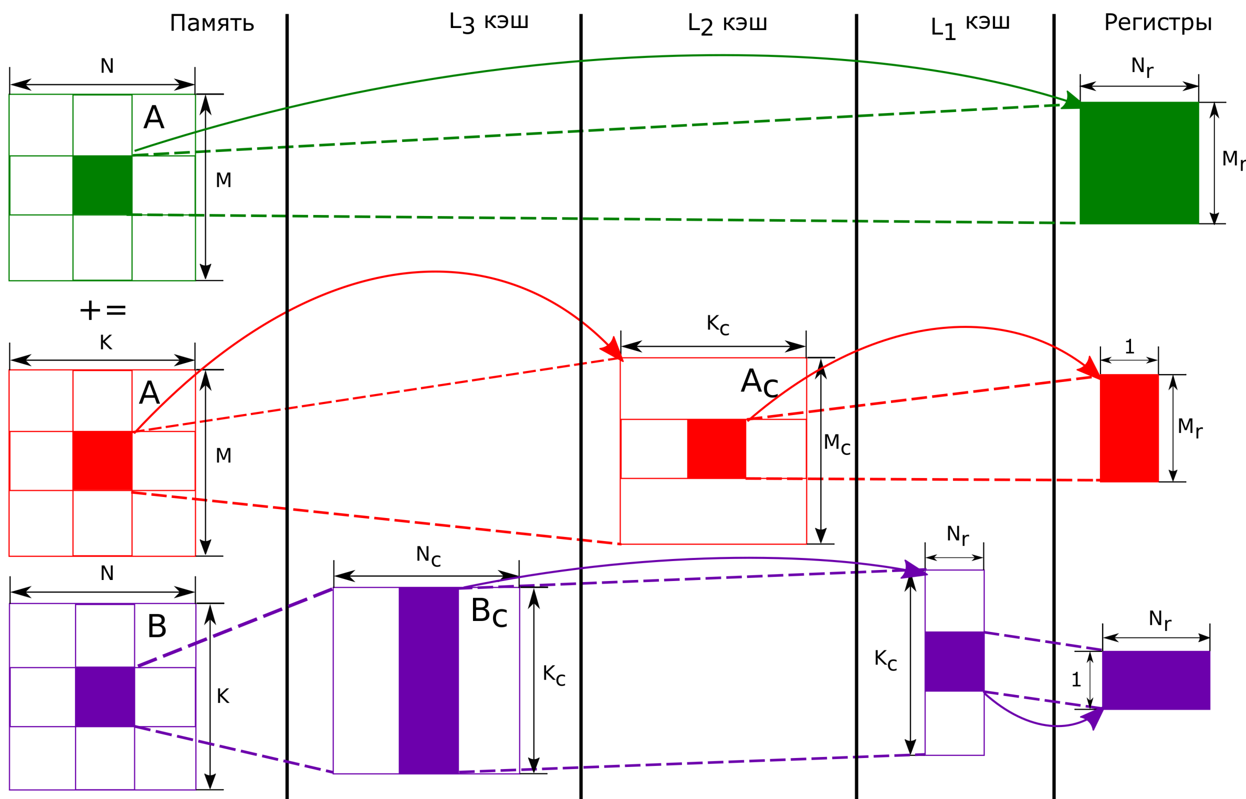


Рис. 1.1. Разбиение матриц A, B и C на блоки.

Внутренний цикл алгоритма 1 вместе с содержащимися в нем векторными инструкциями может генерироваться автоматически в процессе компиляции или реализовываться вручную в виде отдельной процедуры, написанной на языке ассемблера для целевой аппаратной платформы [3, 53].

Предложенные в работе S. A. Hassan [36] параметризованные алгоритмы 2 и 3 являются примерами современных алгоритмов выполнения MVM, используемых в качестве основы для создания новых реализаций BLAS методом ручной настройки для случаев транспонированной и нетранспонированной матрицы, соответственно. На вход алгоритма 2 подаются матрица A и векторы x и y, имеющие размерность $M \times N$, M и N, соответственно. Вектор y содержит результат выполнения MVM для случая $y = A^{\mathsf{T}}x + y$. На вход алгоритма 3 подаются матрица A и векторы x, y, имеющие раз-

мерность $M \times N$, N и M, соответственно. Вектор y содержит результат выполнения MVM для случая $y = Ax + y$.

Алгоритм 2: Вычисление MVM. Случай транспонированной матрицы

```
1  begin
2    for i_b = 0...M step BS do
3      for j_b = 0...N step BS do
4        if (j_b < N − D·BS) {
5          Предвыборка A(i_b : i_b + BS − 1, j_b + D·BS : j_b + (D + 1)BS − 1)
           в L_2 и y(j_b + D·BS : j_b + (D + 1)BS − 1) в L_1
         } else if (i_b < M − BS) {
6          Предвыборка A(i_b+BS : i_b+2BS−1, j_b : j_b−(N−D·BS)+BS−1)
           в L_2 и y(j_b : j_b − (N − D·BS) + BS − 1) в L_1
         }
7        for i = 0...BS step 1 do
8          I = i_b + i , BS_r = VectorLength,acc(0:BS_r − 1) = x(I)
9          for j = 0...BS step BS_r do
10           J = j_b + j
11           y(J:J + BS_r − 1) += A(I, J : J + BS_r − 1) · acc(0 : BS_r − 1)
12         end
13       end
14     end
15   end
16 end
```

В процессе выполнения алгоритмов 2 и 3 матрицы и векторы разбиваются на блоки. Это влияет на предвыборку данных, загружающую данные в L_1 и L_2 кэш, а также позволяет многократно использовать элементы блоков, хранящиеся в кэш-памяти. Размеры блоков BS, BS_r и шаг предвыборки D являются параметрами алгоритмов. Их наилучшие значения могут определяться с помощью перебора во время ручной настройки.

Для сокращения времени разработки высокопроизводительных реализаций BLAS авторы библиотек ATLAS [40] и PHiPAC [41] предложили использование автонастройки. В ходе ее работы выполняется перебор всех возможных значений параметров рассматриваемой программы с целью получения реализации, имеющей наименьшее время выполнения. Как и в слу-

чае ручной настройки, для выполнения автонастройки необходим доступ к целевой аппаратной платформе, а также потенциально большие временные затраты. Было показано, что в случае ATLAS и MMM автонастройка может приводить к неоптимальным результатам [27].

Алгоритм 3: Вычисление MVM. Случай нетранспонированной матрицы

```
1  begin
2    for i_b = 0...M step BS do
3      for j_b = 0...N step BS do
4        if (j_b < N − D·BS) {
5          Предвыборка A(i_b : i_b + BS − 1, j_b + D·BS : j_b + (D+1)BS − 1)
           в L_2 и y(j_b + D·BS : j_b + (D+1)BS − 1) в L_1
         } else if (i_b < M − BS) {
6          Предвыборка A(i_b+BS : i_b+2BS−1, j_b : j_b−(N−D·BS)+BS−1)
           в L_2 и y(j_b : j_b − (N − D·BS) + BS − 1) в L_1
         }
7        for i = 0...BS step 1 do
8          I = i_b + i , BS_r = VectorLength, acc(0:BS_r − 1) = 0
9          for j = 0...BS step BS_r do
10           J = j_b + j
11           acc(0 : BS_r − 1) += A(I, J : J + BS_r − 1) · x(J:J + BS_r − 1)
12         end
13         y(I) += Σ_{i=0}^{BS_r−1} acc(0, i)
14       end
15     end
16   end
17 end
```

Автонастройка и аналитические техники могут применяться совместно с целью определения наилучшей оптимизации. Такой подход используется в компиляторах LGen [42] и BTO (Build to Order BLAS) [43].

LGen — компилятор для базовых операций линейной алгебры с операндами малой размерности, известной на этапе компиляции. Для выполнения разбиений циклов, векторизации и слияния циклов LGen использует специализированный язык, позволяющий построить математическое пред-

ставление программы для ее дальнейшей оптимизации. Результатом работы LGen являются функции на языке C, содержащие SIMD инструкции.

BTO компилирует последовательность выражений, содержащих матрицы и векторы, написанную на языке MATLAB, в оптимизированный код на языке C++. В процессе оптимизации BTO выполняет слияние циклов, основываясь на модели ГП и эмпирических методах.

Фреймворк AUGEM [44] использует автонастройку совместно с эвристическими алгоритмами векторизации, распределения регистров и планирования команд. С помощью библиотеки POET [83] фреймворк AUGEM распознает части абстрактного синтаксического дерева программы на языке C, содержащие заранее определенный набор операций с матрицами, для их дальнейшей оптимизации.

Время выполнения реализации алгоритма для рассматриваемого набора значений его параметров может быть оценено с использованием машинного обучения [84–86]. Такой подход сокращает время выполнения автонастройки за счет выполнения дополнительных измерений на аппаратных платформах, имеющих схожие характеристики с целевой. В общем случае существующие подходы к оптимизации времени выполнения программ, основанные на машинном обучении, недостаточны для автоматического получения высокопроизводительных реализаций программ в процессе компиляции [87, 88].

1.2. Модель гипотетического процессора

Для достижения высокой производительности реализация алгоритма должна учитывать особенности целевой аппаратной платформы. В большинстве случаев для этого достаточно упрощенного описания компьютера, опускающего некоторые детали работы аппаратных средств. Модель ГП является абстрактным описанием многоядерного процессора общего назначе-

ния, используемым для отображения реализуемого алгоритма на реальный компьютер. В результате отображения могут быть выведены формулы, позволяющие получить значения параметров отображаемого алгоритма в зависимости от характеристик многоядерных процессоров общего назначения.

Рассмотрим модель ГП, позволяющую найти значения параметров реализации алгоритма 1 в библиотеке BLIS [47]. Полученные значения параметров позволяют достигнуть производительности библиотек, содержащих реализацию MMM [47]. ГП описывается следующим образом.

1. **Архитектура загрузки/сохранения и векторные регистры:** данные должны быть загружены в регистры процессора перед тем, как с ними могут быть выполнены вычисления. N_{REG} — количество векторных регистров. N_{VEC} — количество значений размерности S_{DATA}, которые может содержать векторный регистр. Если N_{REG} равно 0, N_{VEC} присваивается значение 1. Предполагается, что команды для работы с памятью могут выполняться одновременно с командами арифметики с плавающей точкой.

2. **Векторные инструкции:** N_{VFMA} — количество векторных инструкций VFMA (Vector Fused Multiply-Add), вычисляемых процессором за один такт. N_{VFMA} определяет пропускную способность процессора. VFMA выполняет N_{VEC} умножений и сложений, составляющих FMA. L_{VFMA} — минимальное количество тактов, которое должно быть совершено перед началом выполнения новой зависимой по данным VFMA-инструкции. L_{VFMA} определяет задержку VFMA-инструкции.

3. **Кэш-память:** весь кэш данных — множественно-ассоциативный кэш с политикой вытеснения последнего по времени использования (англ. *Least Recently Used*, сокр. *LRU*). Каждый уровень кэша L_i характеризуется следующими параметрами: S_{L_i} — размер L_i; W_{L_i} — степень ассоциативности; N_{L_i} — количество множеств; C_{L_i} — размер линии кэша, где $S_{\mathrm{L}_i} = N_{\mathrm{L}_i} C_{\mathrm{L}_i} W_{\mathrm{L}_i}$. В случае полностью ассоциативного кэша $N_{\mathrm{L}_i} = 1$.

В реальных случаях значения параметров ГП определяются техническими характеристиками целевой аппаратной платформы. Так, например, информация о пропускных способностях и задержках инструкций процессора может быть найдена в справочной документации по оптимизациям программного обеспечения для данных процессоров [89, 90]. Данные о кэш-памяти и векторных инструкциях доступны в сопроводительной документации к производимым процессорам.

Если у процессора отсутствуют VFMA инструкции, L_{VFMA} может быть вычислена как сумма задержек векторного сложения и умножения. N_{VFMA} может быть вычислено как наименьшее из пропускных способностей векторных инструкций сложения и умножения, если обе эти инструкции могут быть выполнены процессором одновременно [47]. В противном случае, N_{VFMA} может быть вычислена как наименьшее из пропускных способностей векторных инструкций сложения и умножения, разделенное на два.

Определение значений параметров алгоритма 1. Для вывода формул, позволяющих найти значения параметров алгоритма 1, в работе T. M. Low [47] предлагается смоделировать выполнение алгоритма 1 на ГП.

Внутренний цикл алгоритма 1 выполняет $N_r \times M_r$ инструкций VFMA, вычисляющих новое значение для каждого из элементов подматрицы размерности $N_r \times M_r$ матрицы C. В силу того, что VFMA-инструкции имеют задержку L_{VFMA} тактов процессора, L_{VFMA} тактов должно пройти перед вычислением нового значения каждого отдельно рассматриваемого элемента подматрицы. Так как пропускная способность процессора составляет N_{VFMA} VFMA-инструкций, то ГП должен начать выполнение не менее $N_{\mathrm{VFMA}} L_{\mathrm{VFMA}}$ VFMA-инструкций, чтобы избежать простоя конвейера процессора. Следовательно, $N_{\mathrm{VFMA}} L_{\mathrm{VFMA}} N_{\mathrm{VEC}}$ элементов подматрицы C должно быть вычислено. Вследствие этого

$$M_r N_r \geq N_{\mathrm{VEC}} L_{\mathrm{VMMA}} N_{\mathrm{VMMA}}. \tag{1.1}$$

На каждой итерации внутреннего цикла алгоритма 1 вычисляются новые значения подматрицы размерности $N_r \times M_r$ матрицы C и используются M_r и N_r новых элементов матрицы A и B. Для ускорения времени доступа к элементам матриц, используемых внутренним циклом, данные загружаются и хранятся на векторных регистрах процессора. Так как количество регистров ограничено, то используются наименьшие из возможных значений параметров M_r и N_r. Такой подход позволяет выполнить размотку внутреннего цикла алгоритма 1 для более эффективного использования векторных регистров, учитывая ограничение на количество векторных регистров. Следовательно,

$$M_r N_r = N_{\text{VEC}} L_{\text{VMMA}} N_{\text{VMMA}}. \tag{1.2}$$

Внутренний цикл алгоритма 1 вычисляет $2M_r N_r K_c$ операций с плавающей точкой и выполняет $2M_r N_r + M_r K_c + N_r K_c$ операций с памятью. Учитывая уравнение (1.2), для того чтобы минимизировать соотношение количества операций с памятью к количеству операций с плавающей точкой, параметры M_r и N_r должны иметь равные значения. Для того чтобы гарантировать полное заполнение векторных регистров, используемых в процессе выполнения микро-ядра, M_r или N_r должны быть кратны N_{VEC}. Так как M_r и N_r принимают целочисленные значения, они могут быть вычислены следующим образом:

$$N_r = \left\lceil \sqrt{N_{\text{VEC}} L_{\text{VFMA}} N_{\text{VFMA}}} / N_{\text{VEC}} \right\rceil N_{\text{VEC}},$$

$$M_r = \left\lceil N_{\text{VEC}} L_{\text{VFMA}} N_{\text{VFMA}} / N_r \right\rceil.$$

Выведем формулы для определения значений параметров K_c, M_c и N_c. Для этого рассмотрим, как изменение их значений влияет на использование кэш-памяти ГП.

В большинстве случаев размер уровней кэш-памяти уменьшается при их возрастании. При этом скорость доступа к данным кэш-памяти уменьшатся с увеличением уровня. Вследствие этого размеры подматриц долж-

ны быть подобраны так, чтобы элементы матриц, используемые чаще остальных, оставались в кэш-памяти наименьшего из доступных уровней как можно дольше.

Рассмотрим цикл алгортима 1 с индуктивной переменной i_r. На каждой итерации данного цикла используются одни и те же $K_c \times N_r$ элементов матрицы B и $M_r \times K_c$ новых элементов матрицы A. Обозначим данные подматрицы с помощью B_r и A_r, соответственно. Чтобы элементы B_r оставались в L_1 между двумя последовательными итерациями рассматриваемого цикла, они не должны вытесняться загружаемыми элементами A_r и C. При этом параметр K_c должен иметь наибольшее значение, чтобы увеличить размер данных, используемых повторно.

Принимая во внимание политику вытеснения LRU гипотетического процессора, можно сформулировать следующие условия, позволяющие повысить вероятность сохранения элементов B_r в L_1.

1. На каждом шаге цикла с индуктивной переменной i_r каждый элемент B_r и A_r загружается только один раз. Вследствие этого, если загрузить элементы A_r в L_1 раньше чем элементы B_r, элементы A_r будут находится раньше в очереди на вытеснение по сравнению с элементами B_r.

2. Так как элементы A_r не используются повторно на следующей итерации описываемого цикла, они могут быть полностью вытеснены. Следовательно, полное замещение элементов A_r элементами A_r, используемых на следующий итерации, позволило бы хранить в L_1 подматрицу B_r больших размеров.

Для того чтобы выполнялось первое условие, предвыборка элементов B_r может быть выполнена после предвыборки элементов A_r. Для выполнения второго условия подберем наибольшее значение параметра K_c, позволяющее на каждой итерации цикла с индуктивной переменной i_r использовать одни и те же множества L_1 для хранения элементов A_r.

Так как L_1 множественно-ассоциативного кэша ГП имеет N_{L_1} множеств, то элементы A_r с адресами в памяти, отличающимися на $N_{L_1}C_{L_1}$ байтов, будут загружаться в одно и то же множество L_1. Следовательно, разница адресов памяти соответствующих элементов A_r, используемых на двух последовательных итерациях цикла с индуктивной переменной i_r, должна быть кратна $N_{L_1}C_{L_1}$.

Вследствие того, что элементы A_r хранятся последовательно во временном массиве A_c, и подматрица A_r содержит $M_r \times K_c$ элементов, разница между адресами двух соответствующих элементов A_r, используемых двумя последовательными итерациями цикла с индуктивной переменной i_r, составляет $M_r K_c S_{\mathrm{DATA}}$ байтов. Следовательно,

$$M_r K_c S_{\mathrm{DATA}} = C_{A_r} N_{L_1} C_{L_1}, \tag{1.3}$$

где $C_{A_r} \in \mathbb{N}$ — количество линий каждого множества L_1, занимаемых элементами A_r, согласно уравнению 1.3. Выведем формулу для вычисления значений параметра K_c:

$$K_c = (C_{A_r} N_{L_1} C_{L_1})/(M_r S_{\mathrm{DATA}}). \tag{1.4}$$

Для того чтобы избежать вытеснения данных из L_1, количество доступных линий кэша каждого используемого множества не должно превышать имеющиеся. Следовательно, $C_{A_r} \leq W_{L_1}$. Так как во время выполнения внутреннего цикла алгоритма 1 используются элементы подматрицы B_r, то

$$C_{A_r} + C_{B_r} \leq W_{L_1}, \tag{1.5}$$

где $C_{B_r} \in \mathbb{N}$ — количество линий каждого множества L_1, занимаемых элементами B_r.

По крайней мере одна линия каждого множества L_1 может использоваться для хранения элементов матрицы C. Вследствие этого

$$C_{A_r} + C_{B_r} \leq W_{L_1} - 1. \tag{1.6}$$

Так как элементы B_r хранятся последовательно во временном массиве B_c, и подматрица B_r содержит $K_c \times N_r$ элементов, используя уравнение 1.3

получаем

$$C_{B_r} = (N_r K_c S_{\text{DATA}})/(N_{\text{L}_1} C_{\text{L}_1}) = (N_r C_{A_r})/M_r. \tag{1.7}$$

Вследствие того, что $C_{A_r} \in \mathbb{N}$, $C_{B_r} \in \mathbb{N}$, и требуется выделить как можно больше места для элементов подматрицы B_r, $C_{B_r} = \lceil (N_r C_{A_r})/M_r \rceil$ и $C_{A_r} = \lfloor (W_{\text{L}_1} - 1)/(1 + N_r/M_r) \rfloor$.

Аналогичные рассуждения можно применить для вывода формул, определяющих значения параметров N_c и M_c. Таким образом, имеем следующие формулы [47]:

$$N_r = \lceil \sqrt{\gamma}/N_{\text{VEC}} \rceil N_{\text{VEC}}, M_r = \lceil \gamma/N_r \rceil, K_c = \left\lfloor \frac{W_{\text{L}_1} - 1}{1+N_r/M_r} \right\rfloor N_{\text{L}_1} C_{\text{L}_1}/M_r S_{\text{DATA}},$$

$$M_c = (W_{\text{L}_2} - 2) S_{\text{L}_2}/K_c S_{\text{DATA}} W_{\text{L}_2}, N_c = \lfloor C_{B_c}/K_c S_{\text{DATA}} N_r \rfloor N_r,$$

где $\gamma = N_{\text{VEC}} L_{\text{VFMA}} N_{\text{VFMA}}$, C_{B_c}— количество байтов, доступных для B_c.

Выведенные формулы позволяют определить значения параметров M_r, N_r, K_c, M_c и N_c однопоточной реализации алгоритма 1 за константное время без доступа к целевой аппаратной платформе. Это упрощает создание новых реализаций МММ для новых целевых аппаратных платформ. В случае многоядерных процессоров и нескольких потоков вывод формул, позволяющих определить значения параметров алгоритма 1, остается открыт.

1.3. Обзор методов сокращения времени выполнения тензорных сверток

За последние двадцать лет прогресс во многих научных дисциплинах, таких как квантовая химия и общая теория относительности, основывался на моделировании физических систем, использующем ТС. ТС применяется в алгоритмах машинного обучения, таких как обучение и логический вывод в глубинных нейронных сетях [91–93]. Квантовая химия широко использует ТС в методах Хартри—Фока, а также при решении других задач, имеющих

существенную практическую значимость [94–96]. TC применяется при моделировании потоков несжимаемой жидкости на основе спектральных методов [97–100]. TC может быть использована для вычисления многомерного преобразования Фурье [101], применяемого, в частности, при обработке сигналов [102] и изображений [103, 104]. TC используется при моделировании климата [105–107]. Несмотря на широкое применение TC, количество подходов к сокращению времени выполнения этой операции намного меньше чем количество подходов к сокращению времени выполнения MMM и MVM [59]. В данном разделе описываются основные методы сокращения времени выполнения TC.

Рассмотрим определения d-мерного тензора и TC, позволяющие свести сокращение времени выполнения этой операции к сокращению времени выполнения MMM и MVM:

Определение 1.1. d-мерный тензор $\mathcal{T} \in \mathbb{R}^{n_{u_0} \times \ldots \times n_{u_{d-1}}}$ может быть определен следующим образом [16]:

$$\mathcal{T} \equiv \{A_{u_0 \ldots u_{d-1}} \in \mathbb{R} | (u_0, \ldots, u_{d-1}) \in n_{u_0} \times \ldots \times n_{u_{d-1}}\}.$$

Определение 1.2. Пусть $\mathcal{A}$, $\mathcal{B}$ и $\mathcal{C}$ — d_A-, d_B- и d_C- мерные тензоры, соответственно. Сворачиваемые индексы $\mathcal{A}$ и $\mathcal{B}$ описываются кортежем $P = p_0 \ldots p_{t-1}$. Индексы $\mathcal{C}$, а также свободные индексы $\mathcal{A}$ и $\mathcal{B}$ описываются кортежами $I = i_0 \ldots i_{r-1}$ и $J = j_0 \ldots j_{s-1}$, соответственно. Операция свертывания или свертки $\mathcal{A}$ и $\mathcal{B}$ определяется как $\mathcal{C}_{\pi_C(IJ)} = \sum\limits_P \alpha \cdot \mathcal{A}_{\pi_A(IP)} \cdot \mathcal{B}_{\pi_B(PJ)} + \beta \cdot \mathcal{C}_{\pi_C(IJ)}$, где $\sum\limits_P = \sum\limits_{p_0=0}^{n_{p_0}-1} \ldots \sum\limits_{p_{t-1}=0}^{n_{p_{t-1}}-1}$; $\pi_C(IJ)$, $\pi_A(IP)$ и $\pi_B(PJ)$ — перестановки индексов; α, $\beta \in \mathbb{R}$ [17].

С целью сокращения времени выполнения TC может быть использован программный код оптимизированных MMM и MVM, полученный в результате выполнения алгоритмов, описанных в разделе 1.1. Так, например, метод Transpose-Transpose-GEMM-Transpose (TTGT) [54], применяемый в различных прикладных библиотеках (NumPy [108], Eigen [109], MATLAB

Tensor Toolbox [110] и др.), выполняет перестановку индексов тензоров с целью их представления в виде матриц и использования МММ. Для выполнения таких перестановок требуются дополнительное время и память.

Метод Loops-over-GEMMS (LoG) [59] представляет сворачиваемые тензоры как наборы матриц, для которых выполняется МММ. Производительность LoG зависит от размера таких матриц, а также расположения тензоров в памяти [59]. Так, например, в случае свертки 3-мерного тензора $\mathcal{A} \in \mathbb{R}^{m \times n \times k}$ и 2-мерного тензора $\mathcal{B} \in \mathbb{R}^{k \times l}$ в 3-мерный тензор $\mathcal{C} \in \mathbb{R}^{m \times n \times l}$, имеющей следующий вид $\mathcal{C}_{\alpha\beta\rho} = \sum_{\delta=0}^{k-1} \mathcal{A}_{\alpha\beta\delta} \cdot \mathcal{B}_{\delta\rho} + \mathcal{C}_{\alpha\beta\rho}$, LoG вычислит МММ для каждого из n значений индекса β.

Для решения задач в таких научных областях, как, например, квантовая химия, созданы модификации метода TTGT для распределенных вычислений. Такие подходы применяют разбиение тензоров на блоки и другие техники для уменьшения передачи данных в иерархии памяти, а также между процессорами. Примерами библиотек, содержащих реализации описанных методов, служат Tensor Contraction Engine [54], Cyclops Tensor Framework [55], libtensor [56], TiledArray [57, 58] и др.

Для сокращения издержек, связанных с перестановкой индексов тензоров, можно использовать только части реализации оптимизированного МММ, содержащие векторные инструкции на языке ассемблера [16, 17]. Такие части не описываются интерфейсом BLAS и в данный момент доступны только в библиотеке BLIS, поддерживающей ограниченное количество целевых аппаратных платформ.

Описываемый подход применяется в методах GETT (GEMM-like Tensor-Tensor multiplication) [17] и TBLIS (Tensor-Based Library Instantiation Software) [16]. Как и в случае алгоритма 1, для вычисления МММ подходы GETT и TBLIS копирует элементы операндов TC во временные одномерные массивы с целью дальнейшего использования кода реализации МММ, содержащего векторные инструкции. Интерфейс BLAS неприменим для ре-

ализации GETT и TBLIS, которые в данный момент поддерживают малое количество целевых аппаратных платформ [16, 17].

В отличии от GETT, TBLIS представляет тензоры в виде матриц, что позволяет явно выполнять разбиение матриц на блоки и другие трансформации, используемые в алгоритме 1. В качестве примера применения такого представления рассмотрим d-мерный тензор $\mathcal{T} \in \mathbb{R}^{n_{u_0} \times \ldots \times n_{u_{d-1}}}$. $A_{u_0 \ldots u_{d-1}}$, являющийся элеменом тензора $\mathcal{T}$, расположен в памяти со смещением $\sum_{k=0}^{d-1} u_k \prod_{l=k+1}^{d-1} n_{u_l}$. Пусть свободные и сворачиваемые индексы тензора $\mathcal{T}$ описываются кортежами $I = i_0 \ldots i_{r-1}$ и $P = p_0 \ldots p_{s-1}$, соответственно. Тогда смещение может быть представлено в виде $\sum_{k=0}^{d_I-1} i_k \left(\prod_{l=k+1}^{d_I-1} n_{i_l}\right) \left(\prod_{l=0}^{d_P-1} n_{p_l}\right) + \sum_{k=0}^{d_P-1} p_k \prod_{l=k+1}^{d_P-1} n_{p_l} = \left(\sum_{k=0}^{d_I-1} i_k \prod_{l=k+1}^{d_I-1} n_{i_l}\right) n_P + \sum_{k=0}^{d_P-1} p_k \prod_{l=k+1}^{d_P-1} n_{p_l} = \overline{I} n_P + \overline{P}$. Можно вывести формулы, позволяющие логически представить тензор $\mathcal{T}$ в виде матрицы M размерности $M \times N$, рассмотрев смещение в памяти $\overline{I}N + \overline{P}$ элемента $\mathrm{M}_{\overline{IP}}$ матрицы M:

$$\overline{I} = \sum_{k=0}^{d_I-1} i_k \prod_{l=k+1}^{d_I-1} n_{i_l},\ N = \prod_{l=0}^{d_P-1} n_{p_l},\ M = \prod_{l=0}^{d_I-1} n_{i_l},\ \overline{P} = \sum_{k=0}^{d_P-1} p_k \prod_{l=k+1}^{d_P-1} n_{p_l}.$$

Другое отличие GETT в применении автонастройки для нахождения наилучших значений параметров разбиения и способов выполнения упаковки элементов тензора. Использование автонастройки позволяет повысить производительность получаемого кода, но делает невозможным применение GETT в условиях недоступности целевой аппаратной платформы и ограниченности времени выполнения оптимизации. Кроме этого GETT не поддерживает тензоры, размер которых неизвестен на этапе компиляции.

Для оптимизации времени выполнения TC могут применяться последовательности стандартных оптимизаций циклов (перестановка циклов, разбиение циклов на блоки, размотка циклов, векторизация и др.). Такие подходы зависят от размерности тензоров и их расположения в памяти [60]. В частности, производительность TC для задач малой размерности может

быть улучшена за счет векторизации циклов [61]. Для задач большой размерности может применяться генератор кода циклов для графических ускорителей [62].

1.4. Выводы

За последние двадцать лет количество методов сокращения времени выполнения ТС значительно увеличилось и продолжает возрастать. В большинстве случаев они используют методы сокращения времени выполнения важных частных случаев ТС — MMM и MVM. Распространенными способами сокращения времени выполнения ТС являются ручная настройка и автонастройка, требующими доступа к целевой аппаратной платформе и потенциально большого времени выполнения. Примером такой ситуации являются оптимизации, выполняемые промышленными компиляторами по умолчанию в процессе кросс-компиляции. Моделирование вычисления MMM на ГП позволило вывести формулы для определения значений параметров данного алгоритма без доступа к целевой аппаратной платформе за константное время. Это позволяет сделать вывод о том, что целесообразно моделировать вычисления алгоритмов на ГП для создания новых оптимизаций, имеющих более широкое практическое применение. Обзор методов сокращения времени выполнения тензорных операций опубликован в статье [111].

Глава 2

Алгоритм автоматического сокращения времени выполнения тензорных операций

Во второй главе представлена новая модель ГП, описывающая инструкции предвыборки и операции из замкнутых полуколец с элементами из множества вещественных чисел. Представлены новые алгоритмы для вычисления обобщений MMM и MVM на замкнутые полукольца. С помощью моделирования выполнения описанных алгоритмов на ГП выведены формулы, позволяющие получить значения параметров описанных алгоритмов в зависимости от характеристик многоядерных процессоров общего назначения. Разработан оригинальный алгоритм автоматического сокращения времени выполнения тензорных операций. Доказаны утверждения о существовании значений параметров, при которых отсутствует простой конвейера векторных инструкций гипотетического процессора для представленных алгоритмов при возможности мгновенной загрузки данных из памяти на векторные регистры.

2.1. Модификация модели гипотетического процессора. Матрично-векторное произведение. Общая задача о путях

Для вывода формул, позволяющих найти значения параметров предлагаемых алгоритмов, построена новая математическая модель ГП. Описываемая модель модифицирует ГП, использованный в работе T. M. Low[2] и описанный в разделе 1.2 для нахождения параметров МММ. Новизна модели заключается в описании операций предвыборки данных в L_1 и обобщении инструкций FMA (Fused Multiply-Add) на замкнутые полукольца с элементами из множества вещественных чисел с целью применения для сокращения времени выполнения решений общей задачи о путях.

Автором предложена модификация ГП, определяющая инструкции предвыборки для загрузки данных в L_1 [1, 5].

- **Инструкции предвыборки:** N_{prefetch} — количество инструкций предвыборки, которое может быть выполнено за один такт. L_{prefetch} — количество тактов, составляющих задержку каждой инструкции. Каждая инструкция может загрузить данные, размер которых равен C_{L_i}.

Помимо инструкций предвыборки модификация определяет минимальное количество тактов L_{VLOAD}, которое должно быть совершено перед началом выполнения новой зависимой по данным векторной инструкции загрузки данных на векторный регистр процессора из L_1.

Предложенные модификации позволяют смоделировать выполнение предвыборки данных на ГП с целью определения ее шага и других параметров после отображения реализуемого алгоритма на произвольный реальный процессор общего назначения.

Для эффективного выполнения предвыборки данных в L_i уровень кэша нужно определить как часто должна выполняться предвыборка [112]. В

большинстве случаев во время выполнения одной инструкции предвыборки может быть загружено $C_{\mathrm{L_i}}$ байтов, составляющих одну линию $\mathrm{L_i}$ уровня кэша. Вследствие этого предвыборка данных, расположенных в памяти ближе чем на $C_{\mathrm{L_i}}$ байтов, создает дополнительные издержки. Помимо этого нужно учесть размер $\mathrm{L_i}$ уровня кэша, чтобы избежать вытеснения данных. В качестве примера рассмотрим предвыборку элементов матрицы A в $\mathrm{L_2}$, выполняемую циклом с индуктивной переменной j_b алгоритма 3. Предположим, что шаги циклов с индуктивными переменными i_b и j_b имеют значения M_c и N_b, соответственно. На каждой итерации описываемого цикла в $\mathrm{L_2}$ загружается M_c наборов по N_b элементов матрицы A, расположенных в памяти последовательно. Каждый из этих наборов отделен от предыдущего N - N_b элементами. Если уровень $\mathrm{L_2}$ множественно-ассоциативный и виртуально индексируемый, то адреса виртуальной памяти, различающиеся на $S_{\mathrm{L_2}}/W_{\mathrm{L_2}}$ байтов, будут отображаться в одно и то же множество кэша. Следовательно, если $(N - N_b) \bmod S_{\mathrm{L_2}}/(W_{\mathrm{L_2}}S_{\mathrm{DATA}}) \in [S_{\mathrm{L_2}}/(W_{\mathrm{L_2}}S_{\mathrm{DATA}}) - 1, S_{\mathrm{L_2}}/(W_{\mathrm{L_2}}S_{\mathrm{DATA}}) - N_b + 1] \cup [0]$ и $N_b \leq S_{\mathrm{L_2}}/(W_{\mathrm{L_2}}S_{\mathrm{DATA}})$, тогда необходимо M_c линий кэша в каждом его множестве, чтобы содержать элементы матрицы A без их вытеснения. В случае физически индексируемого кэша такое предположение может быть использовано только в качестве грубой оценки, так как адреса физической памяти, отображаемые в одно и то же множество кэша, могут иметь отличающиеся адреса виртуальной памяти. Так как в кэш-памяти должны хранится элементы векторов x и y, то необходимо учитывать и другие параметры алгоритма.

Для того чтобы выполнить предвобрку эффективно, требуется также вычислить шаг предвыборки, определяющий на сколько должны отстоять адреса загружаемых данных [112]. Предвыборка данных с шагом позволяет загрузить данные в кэш память до того момента, когда они потребуются, несмотря на задержку доступа к памяти. Шаг предвыборки может быть вычислен следующим образом:

$$D \geq \lceil L_{\text{prefetch}}/B \rceil\,, \tag{2.1}$$

где L_{prefetch} — задержка инструкций предвыборки данных в кэш-память и B — количество тактов процессора, требуемых для выполнения итерации цикла [113]. Для того чтобы определить значения параметра B, можно рассмотреть задержки инструкций, выполняемых одной итерацией цикла.

Описанные модификации необходимы для моделирования вычисления MVM. Вычислительная сложность MVM составляет $O(N^2)$, что в N раз меньше чем вычислительная сложность MMM. Вследствие этого каждый элемент умножаемой матрицы используется только $O(1)$ раз, а не $O(N)$ как в случае MMM. Следовательно, стоимость загрузки элементов матрицы из памяти на регистры процессора преобладает над количеством выполняемых операций. Предвыборка данных, загружающая элементы матрицы в кэш первого уровня до момента их использования, может уменьшить количество промахов кэша и, следовательно, уменьшить стоимость загрузки элементов матрицы. Как следствие, эффективная предвыборка необходима в случае MVM.

С целью сокращения времени выполнения решений общей задачи о путях для случая замкнутых полуколец с элементами из множества вещественных чисел вместо FMA, используемых в алгоритме 1, предлагаемая автором модификация ГП содержит операции из MMA (Matrix Multiply-and-Add), обобщающие FMA на замкнутые полукольца с элементами из множества вещественных чисел [10]. Векторная инструкция, выполняющая операции MMA для N_{VEC} элементов данных, будет обозначаться как VMMA.

Рассмотрим определение замкнутого полукольца [114]:

Определение 2.1. *Замкнутым полукольцом* (*closed semiring*) называется алгебра $\{S, \oplus, \otimes, 0, 1\}$, где S — множество элементов; $\oplus$ и $\otimes$ — бинарные операции над S, обладающие следующими свойствами.

1. $\langle S, \oplus \rangle$ — идемпотентный коммутативный моноид. То есть имеют место свойства:

- коммутативности: $a \oplus b = b \oplus a$.
- ассоциативности: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
- существования нейтрального элемента: $a \oplus 0 = 0 \oplus a = a$.
- идемпотентности: $a \oplus a = a$.

2. $\langle S, \otimes \rangle$ — моноид. То есть имеют место свойства:

- ассоциативности: $(a \otimes b) \otimes c = a \otimes (b \otimes c)$.
- существования нейтрального элемента: $a \otimes 1 = 1 \otimes a = a$.

3. 0 служит аннулятором для $\langle S, \otimes \rangle$: $a \otimes 0 = 0 \otimes a = 0$.

4. Умножение дистрибутивно относительно сложения:

- левая дистрибутивность: $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$.
- правая дистрибутивность: $(a \oplus b) \otimes c = a \otimes c \oplus b \otimes c$.

5. Если $a_1, a_2, ..., a_i, ...$ — счетная последовательность элементов из S, то сумма $a_1 \oplus a_2 \oplus ... \oplus a_i \oplus ...$ существует и единственна. Ассоциативность, коммутативность и идемпотентность выполняется для бесконечных сумм. Операция $\otimes$ дистрибутивна относительно счетных сумм.

Рассмотрим определение алгебры $\{S^{N\times N}, \bar{\oplus}, \bar{\otimes}, \bar{O}, \bar{I}\}$ квадратных матриц с элементами из замкнутого полукольца $\{S, \oplus, \otimes, 0, 1\}$ [10].

Определение 2.2. $\{S^{N\times N}, \bar{\oplus}, \bar{\otimes}, \bar{O}, \bar{I}\}$ — алгебра , описываемая следующим образом.

1. $S^{N\times N}$ — множество матриц размерности $N \times N$ с элементами из множества S замкнутого полукольца $\{S, \oplus, \otimes, *, 0, 1\}$.

2. $\bar{\oplus} : S^{N\times N} \times S^{N\times N} \to S^{N\times N}$ и $\otimes : S^{N\times N} \times S^{N\times N} \to S^{N\times N}$ — бинарные операции, определяемые как:

$A = [a_{ij}]_{i,j\in[1:N]} \in S^{N\times N}, B = [b_{ij}]_{i,j\in[1:N]} \in S^{N\times N}$, $A \bar{\oplus} B = [a_{ij} \oplus b_{ij}]_{i,j\in[1:N]}$, $A \bar{\otimes} B = [\sum\limits_{k\in[1:N]} a_{ik} \otimes b_{kj}]_{i,j\in[1:N]}$.

3. $\bar{O} = [a_{ij}]_{i,j \in [1:N]}$, где $a_{ij} = 0$ для $i, j \in [1 : N]$.

4. $\bar{I} = [a_{ij}]_{i,j \in [1:N]}$, где $a_{ij} = \begin{cases} 1, & \text{i = j} \\ 0, & \text{иначе} \end{cases}$.

Можно доказать, что $\{S^{N \times N}, \bar{\oplus}, \bar{\otimes}, \bar{*}, \bar{O}, \bar{I}\}$ является замкнутым полукольцом [114].

Рассмотрим определение операции MMA [10]:

Определение 2.3. MMA[$\otimes$, $\oplus$] — операция вида $C \leftarrow C \bar{\oplus} A \bar{\otimes} B$, где $\bar{\oplus}$ и $\bar{\otimes}$ — операции из замкнутого полукольца $\{S^{N \times N}, \bar{\oplus}, \bar{\otimes}, \bar{O}, \bar{I}\}$, определенного над замкнутым полукольцом $\{S, \oplus, \otimes, 0, 1\}$.

Примеры замкнутых полуколец и их приложений для решения задач APP (Algebraic Path Problem) или общей задачи о путях над взвешенными ориентированными графами представлены в табл. 2.1 [10, 115].

Таблица 2.1. Примеры замкнутых полуколец и их приложений.

S	$\oplus$	$\otimes$	0	1	Приложение
(0, 1)	$\vee$	$\wedge$	0	1	Транзитивное замыкание бинарного отношения
$\mathbb{R}_+ \cup +\infty$	min	+	$+\infty$	0	Нахождение кратчайших путей
$\mathbb{R}_+ \cup +\infty$	max	+	0	$+\infty$	Нахождение самого широкого пути
$[0, 1]$	max	$\times$	0	1	Нахождение путей наибольшей надежности
$[0, 1]$	min	$\times$	1	0	Нахождение путей наименьшей надежности
$\mathbb{R}_+ \cup +\infty$	min	max	$+\infty$	0	Минимальное остовное дерево
$\mathbb{R}_+ \cup +\infty$	max	min	0	$+\infty$	Нахождение путей наибольшей вместимости

Рассмотрим примеры MMA[$\otimes$, $\oplus$] и их приложений [10].

1. **Транзитивное замыкание бинарного отношения:**
   MMA[$\wedge$, $\vee$] $= c_{ij} \leftarrow c_{ij} \vee \left\{ \bigvee_k \{a_{ik} \wedge b_{kj}\} \right\}$
2. **Задача о кратчайшем пути:**
   MMA[+, min] $= c_{ij} \leftarrow \min \left\{ c_{ij}, \min_k \{a_{ik} + b_{kj}\} \right\}$
3. **Задача о самом широком пути:**
   MMA[+, max] $= c_{ij} \leftarrow \max \left\{ c_{ij}, \max_k \{a_{ik} + b_{kj}\} \right\}$
4. **Задача о пути с наибольшей надежностью:**
   MMA[$\times$, max] $= c_{ij} \leftarrow \max \left\{ c_{ij}, \max_k \{a_{ik} \times b_{kj}\} \right\}$
5. **Задача о пути с наименьшей надежностью:**
   MMA[$\times$, min] $= c_{ij} \leftarrow \min \left\{ c_{ij}, \min_k \{a_{ik} \times b_{kj}\} \right\}$

6. **Построение минимального остовного дерева:**
   MMA[max, min] $= c_{ij} \leftarrow \min\left\{c_{ij}, \min_k \{\max(a_{ik}, b_{kj})\}\right\}$
7. **Нахождение путей наибольшей вместимости:**
   MMA[min, max] $= c_{ij} \leftarrow \max\left\{c_{ij}, \max_k \{\min(a_{ik}, b_{kj})\}\right\}$

Рассмотрим постановку задачи APP [116]. Пусть $G = (V, E, w)$ — взвешенный ориентированный граф, где $V = \{1, 2, ..., N\}$ — множество вершин; $E \subseteq V \times V$ — ребра графа и $w : E \rightarrow S$ — весовая функция, определенная над замкнутым полукольцом $\{S, \oplus, \otimes, *, 0, 1\}$.

Вес пути $p = \langle i, k_1, k_2, ..., k_m, j\rangle$ из вершины i в вершину j равен произведению всех входящих в него ребер:

$$w(p) = w(i, k_1) \otimes w(k_1, k_2) \otimes ... \otimes w(k_m, j).$$

Вес путей длины 0, начинающихся и заканчивающихся в одной и той же вершине, определяется как $1 \in S$. $w(i, j) = 0$, если $(i, j) \notin E$.

Пусть $P(i, j)$ — множество всех путей из вершины i в вершину j. Задача APP состоит в нахождении суммы весов всех путей из вершины i в вершину j для всех i и j:

$$d_{i,j} = \bigoplus_{p \in P(i,j)} w(p).$$

Необходимость в решении задач APP возникает, в частности, в математических методах исследования операций [117–119], робототехнике [120–122], планировании размещения предприятий [123–125], транспортировке [126–128], проектировании сверхбольших интегральных схем [129–131], при нахождении путей в MapQuest [132] и Google Maps [133] и других картографических сервисах [134–136]. Задача APP возникает при решении важных задач вычислительной геометрии, таких как построение Диаграмм Вороного [137–139] и расчет траектории движения [140–142].

Рассмотрим постановку задачи APP в матричной форме [116]. Определим матрицу весов $A = [a_{ij}] \in S^{N \times N}$, где

$$a_{ij} = \begin{cases} w(i,j), & (i,j) \in E \\ 0, & \text{иначе} \end{cases}.$$

Определим матрицу $D = [d_{ij}] \in S^{N \times N}$, где d_{ij} — сумма весов всех путей из вершины i в вершину j.

D может быть вычислена как [116]:

$$D = \bigoplus_{m \geq 0} A^m = \bar{I} \bar{\oplus} A \bar{\oplus} A^2 \bar{\oplus} A^3 \bar{\oplus} ...$$

Если для замкнутого полукольца выполняется $a \oplus 1 = 1$, то D может быть вычислена как [143]:

$$D = \bar{I} \bar{\oplus} A \bar{\oplus} A^2 \bar{\oplus} A^3 \bar{\oplus} ... \bar{\oplus} A^{N-1}. \tag{2.2}$$

Можно проверить, что $a \oplus 1 = 1$ выполняется для всех замкнутых полуколец, приведенных в табл. 2.1.

Рассмотрим, как решается задача APP в случае выполнения $a \oplus 1 = 1$. Так как $a + a = a$, то $A^k \bar{\oplus} A^l = A^k$ для $k \geq l$. В таком случае $D = A^{N-1}$, $A^l = A^{N-1}$ для $l \geq N-1$ и задача APP может быть сведена к задаче возведения матрицы A в степень $N-1$. Такая задача может быть решена за $\Theta(N^4)$ выполнением $N-2$ операций MMA [144]. В силу уравнения 2.1 $A^{2^{\lceil \lg (N-1) \rceil}} = A^{N-1} = D$. Следовательно, A^{N-1} может быть вычислена последовательным возведением A в квадрат. Сложность такого алгоритма составляет $\Theta(N^3 \lg N)$ [144].

В общем случае для решения задачи APP может применяться алгоритм 4 [12,114,116,143], где $a^0_{ij} = a_{ij}$. Обозначим $A^{(i)} = [a^i_{i,j}] \in S^{N \times N}$. После выполнения алгоритма 4 $a^{N-1}_{ij} = d_{ij}$ и $A^{(N-1)} = D$ [144]. Пусть $P^k(i,j)$ — множество всех путей из вершины i в вершину j, для которых все промежу-

точные вершины принадлежат множеству вершин $K = \{1, 2, ..., k\}$. Пусть $d^k_{i,j}$ — сумма весов всех путей из $P^k(i, j)$. Обозначим $D^k = [d^k_{i,j}] \in S^{N\times N}$. В таких обозначениях каждая итерация цикла с индуктивной переменной k алгоритма 4 вычисляет $a^k_{ij} = d^k_{i,j}$ [144].

Алгоритм 4: Решение задачи APP

```
1 begin
2     for k = 0...N step 1 do
3         for i = 0...N step 1 do
4             for j = 0...N step 1 do
5                 a^k_ij = a^{k-1}_ij ⊕ a^{k-1}_ik ⊗ a^{k-1}_kj
6             end
7         end
8     end
9 end
```

Сложность алгоритма 4 составляет $\Theta(N^3)$. Алгоритм Флойда для нахождения кратчайших путей в ориентированном графе может быть получен, используя $\oplus = \min$ и $\otimes = +$ в алгоритме 4 [145]. Алгоритм Уоршелла для нахождения транзитивного замыкания в ориентированном графе может быть получен использованием $\oplus = \vee$ и $\otimes = \wedge$ [146]. Алгоритм Маггса-Плоткина для нахождения минимального остовного дерева за $O(N)$ на MMC (MESH-connected computers) [147] может быть получен использованием $\oplus = \min$ и $\otimes = \max$ [148].

Рассмотрим, как можно свести выполнение алгоритма 4 к умножению матриц для увеличения эффективности использования векторных инструкций процессора [12]. Описанные идеи реализованы в виде алгоритма 5 [12, 149–151].

Будем использовать $A_{I,J}$ для обозначения подматрицы матрицы $A \in S^{N\times N}$, соджержащей пересечение непустого множества строк с номерами $I = \{i_1, i_1 + 1, ..., i_n\}$ и непустого множества столбцов с номерами $J = \{j_1, j_1 + 1, ..., j_n\}$.

Предположим, что выполнено $k_1 - 1$ итераций цикла с индуктивной переменной k алгоритма 4. Пусть дано множество вершин $K = \{k_1, k_1 + 1, ..., k_n\}$, где $k_n > k_1$. Предположим, что $|V|$ кратно $B = |K|$ и выполняется свойство $a \oplus 1 = 1$.

Алгоритм 5: Решение задачи APP

```
1  begin
2      for k = 0...N step B do
3          K = {k, k+1, ..., k+B-1}
4          Вычисляем A_{K,K}^{(k+B)}, используя алгоритм 4
5          for i = 0, ..., k-1, k+B, ..., N step B do
6              I = {i, i+1, ..., i+B-1}
7              A_{I,K}^{(k+B)} = A_{I,K}^{(k)} ⊗̄ A_{K,K}^{(k+B)}
8          end
9          for i = 0, ..., k-1, k+B, ..., N step B do
10             I = {i, i+1, ..., i+B-1}
11             for j = 0, ..., k-1, k+B, ..., N step B do
12                 J = {j, j+1, ..., j+B-1}
13                 A_{I,J}^{(k+B)} = A_{I,J}^{(k)} ⊕̄ A_{I,K}^{(k+B)} ⊗̄ A_{K,J}^{(k)}
14             end
15         end
16         for i = 0, ..., k-1, k+B, ..., N step B do
17             J = {j, j+1, ..., j+B-1}
18             A_{K,J}^{(k+B)} = A_{K,K}^{(k+B)} ⊗̄ A_{K,J}^{(k)}
19         end
20     end
21 end
```

Рассмотрим, как вычислить $D_{K,K}^{k_n}$. По определению $d_{i \in K, j \in K}^{k}$

$$d_{i \in K, j \in K}^{k} = \begin{cases} d_{i,j}^{k_1 - 1} = a_{i,j}^{k_1 - 1}, & k = k_1 - 1 \\ d_{i,j}^{k-1} \oplus d_{i,k}^{k-1} \otimes d_{k,j}^{k-1}, & k \geq k_1 \end{cases}.$$

Следовательно, для того чтобы вычислить $D_{K,K}^{k_n}$, может быть использован алгоритм 4 для подматрицы $A_{K,K}^{(k_1-1)}$ матрицы $A^{(k_1-1)}$.

Обозначим $K' = V \backslash K$. Рассмотрим, как вычислить $D_{K',K}^{k_n}$, используя вычисленную ранее матрицу $D_{K,K}^{k_n}$. $d_{i \in K', j \in K}^{k_n} = d_{i,j}^{k_1 - 1} \oplus \bigoplus\limits_{k \in K} (d_{i,k}^{k_1 - 1} \otimes d_{k,j}^{k_n}) =$

$a_{i,j}^{k_1-1} \oplus \bigoplus_{k\in K}(a_{i,k}^{k_1-1} \otimes d_{k,j}^{k_n}) = a_{i,j}^{k_1-1} \oplus \bigoplus_{k\in K\setminus j}(a_{i,k}^{k_1-1} \otimes d_{k,j}^{k_n}) \oplus a_{i,j}^{k_1-1} \otimes d_{j,j}^{k_n} = a_{i,j}^{k_1-1} \oplus \bigoplus_{k\in K\setminus j}(a_{i,k}^{k_1-1} \otimes d_{k,j}^{k_n}) = \bigoplus_{k\in K}(a_{i,k}^{k_1-1} \otimes d_{k,j}^{k_n})$. Вследствие этого $D_{K',K}^{k_n}$ может быть вычислена как MMA вида $D_{K',K}^{k_n} = A_{K',K}^{k_1-1}\bar{\otimes}D_{K,K}^{k_n}$.

Рассмотрим, как вычислить $D_{K',K'}^{k_n}$, используя вычисленную ранее матрицу $D_{K',K}^{k_n}$. $d_{i\in K',j\in K'}^{k_n} = a_{i,j}^{k_1-1} \oplus \bigoplus_{k\in K}(d_{i,k}^{k_n} \otimes a_{k,j}^{k_1-1}) = A_{K',K'}^{k_1-1} \oplus D_{K',K}^{k_n}\bar{\otimes}A_{K,K'}^{k_1-1}$.

Рассмотрим, как вычислить $D_{K,K'}^{k_n}$, используя вычисленную ранее матрицу $D_{K,K}^{k_n}$. $d_{i\in K,j\in K'}^{k_n} = a_{i,j}^{k_1-1} \oplus \bigoplus_{k\in K}(d_{i,k}^{k_n} \otimes a_{k,j}^{k_1-1}) = a_{i,j}^{k_1-1} \oplus \bigoplus_{k\in K\setminus i}(d_{i,k}^{k_n} \otimes a_{k,j}^{k_1-1}) \oplus d_{i,i}^{k_n} \otimes a_{i,j}^{k_1-1} = a_{i,j}^{k_1-1} \oplus \bigoplus_{k\in K\setminus i}(d_{i,k}^{k_n} \otimes a_{k,j}^{k_1-1}) = \bigoplus_{k\in K}(d_{i,k}^{k_n} \otimes a_{k,j}^{k_1-1})$. Вследствие этого $D_{K,K'}^{k_n}$ может быть вычислена как MMA вида $D_{K,K'}^{k_n} = D_{K,K}^{k_n}\bar{\otimes}A_{K,K'}^{k_1-1}$.

После обобщения FMA на операции из MMA **гипотетический процессор** будет иметь следующий вид.

1. **Архитектура загрузки/сохранения и векторные регистры:** данные должны быть загружены в регистры процессора перед тем, как с ними могут быть выполнены вычисления. N_{REG} — количество векторных регистров. N_{VEC} — количество значений размерности S_{DATA}, которые может содержать векторный регистр. Если N_{REG} равно 0, N_{VEC} присваивается значение 1. L_{VLOAD} — количество тактов процессора, которое должно быть совершено перед началом выполнения новой зависимой по данным векторной инструкции загрузки данных на векторный регистр из L_1. Предполагается, что команды для работы с памятью могут выполняться одновременно с командами арифметики с плавающей точкой.

2. **Векторные инструкции:** N_{VMMA} — количество операций VMMA, вычисляемых процессором за один такт. Отдельная VMMA выполняет N_{VEC} невекторных умножений и сложений, составляющих MMA. N_{VFMA} определяет пропускную способность процессора. L_{VMMA} — минимальное количество тактов, которое должно быть совершено перед началом выполнения новой зависимой по данным VMMA. L_{VMMA} определяет задержку VMMA-инструкции.

3. **Кэш-память:** весь кэш данных — множественно-ассоциативный кэш. Каждый уровень кэша L_i характеризуется следующими параметрами: S_{L_i} — размер L_i; W_{L_i} — степень ассоциативности; N_{L_i} — количество множеств; C_{L_i} — размер линии кэша, где $S_{\mathrm{L}_i} = N_{\mathrm{L}_i} C_{\mathrm{L}_i} W_{\mathrm{L}_i}$. В случае полностью ассоциативного кэша $N_{\mathrm{L}_i} = 1$.

4. **Инструкции предвыборки:** N_{prefetch} — количество инструкций предвыборки, которое может быть выполнено за один такт. L_{prefetch} — количество тактов, составляющих задержку каждой инструкции. Каждая инструкция может загрузить данные, размер которых равен $C_{\mathrm{L_i}}$.

Реальные значения параметров ГП определяются техническими характеристиками целевой аппаратной платформы. Так, например, информация о пропускных способностях операций из MMA может быть найдена в справочной документации по оптимизациям программного обеспечения для данных процессоров [89, 90]. Как и в случае оригинального ГП рассматриваются только ГП с политикой вытеснения последнего по времени использования, которая может не соответствовать реальному процессору. Использование других политик вытеснения является предметом будущих исследований.

L_{prefetch} определяется задержкой доступа к памяти. Если значение L_{prefetch} целевой аппаратной платформы неизвестно, L_{prefetch} присваивается значение 300 тактов процессора. Как было показано в работах [113, 152], $L_{\mathrm{prefetch}} = 300$ является приемлемой оценкой среднего значения задержки доступа к памяти для многих современных платформ и широкого класса задач. В разделе 4.3 проверяется, что это справедливо и для случая MVM.

L_{VMMA} может быть вычислена как сумма задержек векторных инструкций, составляющих VMMA. N_{VMMA} может быть вычислено как наименьшее из пропускных способностей векторных инструкций, составляющих VMMA, если обе эти инструкции могут быть выполнены процессором одновременно. В противном случае N_{VMMA} может быть вычислена как

наименьшее из пропускных способностей векторных инструкций, составляющих VMMA, разделенное на два.

В большинстве случаев целевые аппаратные платформы реализуют векторные инструкции для выполнения операций $\times$, min, max, $+$, необходимых для решения задач из класса APP [89]. Такие инструкции поддерживают различные целые, вещественные и логические типы данных [153]. Для случая операций над матрицами, содержащими элементы из множества комплексных чисел, возможно сведение к вычислениям над вещественными матрицами [53]. В частности, произведение матриц $A \in \mathbb{C}^{N \times M}$ и $B \in \mathbb{C}^{M \times N}$, представимых в виде $A = A_r + A_i i$ и $B = B_r + B_i i$, соответственно, может быть вычислено как $AB = (A_r B_r - A_i B_i) + (A_i B_r + A_r B_i)i$, где $A_r, A_i \in \mathbb{R}^{N \times M}$ и $B_r, B_i \in \mathbb{R}^{M \times N}$.

2.2. Алгоритм вычисления обобщенного матричного произведения

В данном разделе предложен алгоритм 6 вычисления обобщения МММ на замкнутые полукольца с элементами из множества вещественных чисел. Алгоритм 6 является обобщением алгоритма вычисления МММ, описанного в работе T.M. Low [47].

На вход алгоритма 6 подаются матрицы A, B и C, имеющие размерность $M \times K$, $K \times N$ и $M \times N$, соответственно. Матрица C содержит результат выполнения МММ. В процессе выполнения алгоритма матрицы A и B разбиваются на блоки. Это позволяет многократно использовать элементы блоков, хранящихся в кэш-памяти. Размеры блоков N_c, K_c, M_c, N_r и M_r являются параметрами алгоритма. Их значения могут определяться с помощью автонастройки, ручной настройки, а также моделировния его вычисления на ГП.

Алгоритм 6 вычисления MMA имеет следующий вид:

Алгоритм 6: Вычисление обобщенного MMM

```
1  begin
2      for j = 0...N step N_c do
3          for p = 0...K step K_c do
4              Копирование B(p:p + K_c − 1, j:j + N_c − 1) в B_c
5              for i = 0...M step M_c do
6                  Копирование A(i:i + M_c − 1, p:p + K_c − 1) в A_c
7                  for j_c = 0...N_c step N_r do
8                      for i_c = 0...M_c step M_r do
9                          for p_c = 0...K_c step 1 do
10                             𝕀 = i_c : i_c + M_r − 1, 𝕁 = j_c : j_c + N_r − 1
11                             C_c(𝕀, 𝕁) ⊕= A_c(𝕀, p_c) ⊗ B_c(p_c, 𝕁)
12                         end
13                     end
14                 end
15             end
16         end
17     end
18 end
```

В отличие от алгоритма 1 вместо операций сложения и умножения, составляющих FMA, используются операции $\oplus$ и $\otimes$, составляющие описанную в разделе 2.1 операцию MMA[$\otimes$, $\oplus$]. Это позволяет использовать алгоритм 6 для вычисления MMA[$\otimes$, $\oplus$].

По построению алгоритма 6 для определения значений его параметров можно использовать рассуждения, применяющиеся в разделе 1.2 для вывода значений параметров алгоритма 1 через моделирование вычислений на ГП. По определению нового ГП, описанного в разделе 2.1, вместо L_{VFMA} и N_{VFMA} используются L_{VMMA} и N_{VMMA}, соответственно.

Таким образом, формулы, связывающие размеры блоков матриц M_r, N_r, K_c, M_c и N_c с техническими характеристиками реального процессора имеют следующий вид:

$$N_r = \lceil \sqrt{\gamma}/N_{\mathrm{VEC}} \rceil N_{\mathrm{VEC}}, M_r = \lceil \gamma/N_r \rceil, K_c = \left\lfloor \frac{W_{\mathrm{L1}}-1}{1+N_r/M_r} \right\rfloor N_{\mathrm{L_1}} C_{\mathrm{L_1}}/M_r S_{\mathrm{DATA}},$$

$$M_c = (W_{\mathrm{L_2}} - 2) S_{\mathrm{L_2}}/K_c S_{\mathrm{DATA}} W_{\mathrm{L_2}}, N_c = \lfloor C_{B_c}/K_c S_{\mathrm{DATA}} N_r \rfloor N_r,$$

где $\gamma = N_{\text{VEC}} L_{\text{VMMA}} N_{\text{VMMA}}, C_{B_c}$— количество байтов, доступных для B_c.

Формулы для N_r и M_r обеспечивают отсутствие простоя конвейера векторных инструкций ГП во время выполнения тела внутреннего цикла алгоритма 6. Формулы для K_c, M_c и N_c основаны на том, что элементы матриц, используемые чаще остальных, должны оставаться в кэш-памяти наименьшего из доступных уровней как можно дольше (см. раздел 1.2).

Можно сформулировать следующее утверждение.

Утверждение 2.1. Если данные могут быть мгновенно загружены из памяти на векторные регистры, то существуют значения параметров N_r и M_r алгоритма 6, при которых отсутствует простаивание конвейера векторных инструкций ГП в процессе выполнения алгоритма 6.

Для доказательства представленного утверждения необходимо учесть, что по определению ГП команды для работы с памятью могут выполняться ГП одновременно с командами арифметики с плавающей точкой, и повторить рассуждения о выводе значений параметров N_r и M_r из раздела 1.2 (ст. 29).

Частичное экспериментальное подтверждение этого утверждения для реальных многоядерных процессоров общего назначения можно найти в работе [38], где использование двойного блочного размещения матриц и алгоритма 1 позволяет уменьшить количество промахов к данным кэш-памяти и добиться производительности 97% от пиковой.

2.3. Алгоритмы вычисления обобщенного матрично-векторного произведения

В данном разделе предложены алгоритмы 7 и 8 вычисления обобщения MVM на замкнутые полукольца с элементами из множества вещественных чисел для случаев транспонированной и нетранспонированной матри-

цы, соответственно. Алгоритмы 7 и 8 являются обобщением алгоритмов вычисления MVM, описанных в работе S.A. Hassan [36].

Алгоритм 7 вычисления обобщенного MVM для случая транспонированной матрицы имеет следующий вид:

Алгоритм 7: Вычисление обобщенного MVM. Случай транспонированной матрицы

```
1  begin
2      for j_c = 0...N step N_c do
3          for i_c = 0...M step M_c do
4              for j_b = 0...N_c step N_b do
5                  Выполняем предвыборку M_c × N_b элементов матрицы A с
                     шагом D в L_1
                   for i_b = 0...M_c do
6                      I = i_c + i_b
7                      acc(0:N_r − 1) = x(I)
8                      for j_r = 0...N_b step N_r do
9                          𝕁 = j_c + j_b + j_r : j_c + j_b + j_r + N_r − 1
10                         y(𝕁) ⊕= A(I, 𝕁) ⊗ acc(0 : N_r − 1)
11                     end
12                 end
13             end
14         end
15     end
16 end
```

На вход алгоритма 7 подаются матрица A и векторы x и y, имеющие размерность $M \times N$, M и N, соответственно. Вектор y содержит результат выполнения MVM, обобщенного на замкнутые полукольца с элементами из множества вещественных чисел. Обобщенное MVM имеет вид $y = A^{\mathsf{T}} \bar{\otimes} x \bar{\oplus} y$. В процессе выполнения алгоритма 7 матрица A и векторы x и y разбиваются на блоки, как показано на рис. 2.1.

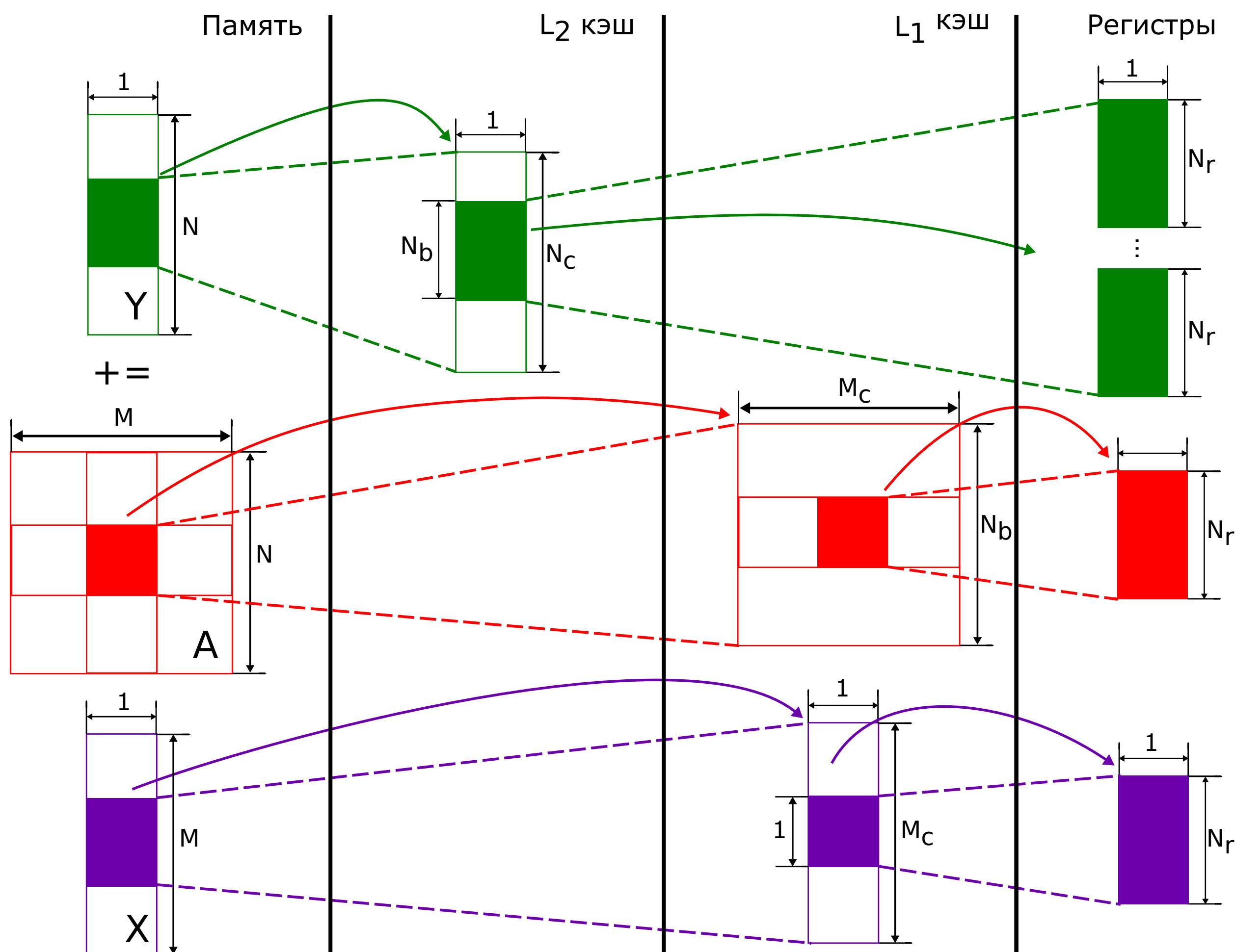


Рис. 2.1. Разбиение векторов x, y и матрицы A на блоки.

Цикл с индуктивной переменной j_c алгоритма 7 разбивает матрицу A и вектор y на подматрицу размерности $M \times N_c$ и подвектор размерности N_c. По сравнению с алгоритмом 2 цикл с индуктивной переменной j_c является дополнительным циклом, добавленным в алгоритме 7 с целью упрощения моделирования его вычислений на ГП.

Цикл с индуктивной переменной i_c разбивает подматрицу матрицы A и вектор x на подматрицу размерности $M_c \times N_c$ и подвектор размерности M_c. N_c элементов вектора y используются повторно на каждой итерации рассматриваемого цикла. Значения параметров N_c и M_c выбираются так, чтобы N_c элементов вектора y не вытеснялись из L_2.

Цикл с индуктивной переменной j_b разбивает подматрицу матрицы A и подвектор вектора y на подматрицу размерности $M_c \times N_b$ и подвектор

размерности N_b. Значение N_b определяет количество элементов вектора y, хранимых на векторных регистрах и используемых циклом с индуктивной переменной i_b. В отличие от алгоритма 2 значения параметров N_b и M_c могут быть различны. Как показано далее, это упрощает моделирование вычислений алгоритма 7 на ГП. Еще одно отличие заключается в том, что во время выполнения каждой итерации цикла с индуктивной переменной j_b подматрица $A(i_c : i_c + M_c - 1, J_c + DN_b : J_c + (D + 1)N_b - 1)$ матрицы A загружается в L_1, а не в L_2. Это позволяет воспользоваться свойствами L_1, который в большинстве случаев является виртуально индексируемым (см. раздел 2.1).

Цикл с индуктивной переменной i_b разбивает подматрицу матрицы A и подвектор вектора x на подматрицу размерности $1 \times N_b$ и подвектор размерности 1.

Цикл с индуктивной переменной j_r разбивает подматрицу матрицы A и подвектор вектора y на подматрицу размерности $1 \times N_r$ и подвектор размерности N_r. В процессе выполнения описываемого цикла подматрица $1 \times N_r$ умножается на элемент вектора x. Вследствие этого параметр N_r определяется размером векторных регистров ГП.

Рассмотрим алгоритм 8, являющийся модификацией алгоритма 3 для вычисления MVM для случая $y = Ax + y$. Описываемая модификация позволяет смоделировать процесс выполнения алгоритма 8 для вывода формул определения значений параметров N_c, M_c, N_r. Модификация также дает возможность использовать операции $\oplus$ и $\otimes$, составляющие описанную в разделе 2.1 операцию MMA[$\otimes$, $\oplus$], вместо операций сложения и умножения.

Алгоритм 8: Вычисление обобщенного MVM. Случай нетранспонированной матрицы

```
1  begin
2      for j_c = 0...N step N_c do
3          for i_c = 0...M step M_c do
4              acc[M_c][N_r]
5              for j_r = 0...N_c step N_r do
6                  if j_r mod 4C_L1/S_DATA == 0 then
7                      Предвыборка M_c × 4C_L1/S_DATA элементов A и 4C_L1/S_DATA
                        элементов x с шагом D в L_1
8                  end
9              end
10             J = j_c + j_r : j_c + j_r + N_r − 1
11             acc(0, 0 : N_r − 1) ⊕= A(i_c, J) ⊗ x(J)
12             ⋮
13             acc(M_c − 1, 0 : N_r − 1) ⊕= A(i_c + M_c − 1, J) ⊗ x(J)
14         end
15         y(i_c) ⊕= ⊕_{i=0}^{N_r−1} acc(0, i)
16         ⋮
17         y(i_c + M_c − 1) ⊕= ⊕_{i=0}^{N_r−1} acc(M_c − 1, i)
18     end
19 end
```

На вход алгоритма 8 подаются матрица A и векторы x, y, имеющие размерность $M \times N$, N и M, соответственно. Вектор y содержит результат выполнения MVM, обобщенного на замкнутые полукольца с элементами из множества вещественных чисел. Обобщенное MVM имеет вид $y = A \bar{\otimes} x \bar{\oplus} y$. В процессе выполнения алгоритма 8 матрица A и векторы x , y разбиваются на блоки как показано на рис. 2.2.

Цикл с индуктивной переменной j_c алгоритма 8 разбивает подматрицу матрицы A и вектор x на подматрицу размерности $M \times N_c$ и подвектор размерности N_c.

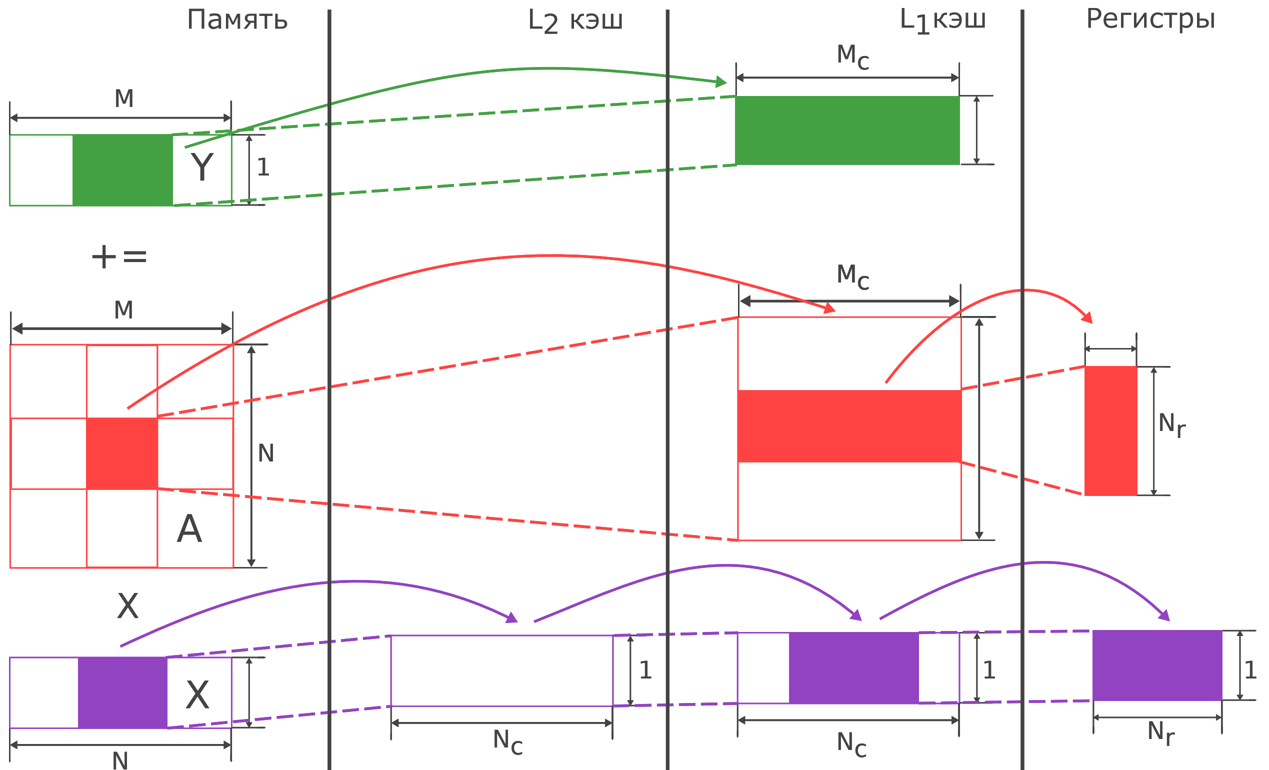


Рис. 2.2. Разбиение векторов x, y и матрицы A на блоки

Цикл с индуктивной переменной i_c разбивает матрицу A и вектор y на подматрицу размерности $M_c \times N_c$ и подвектор размерности M_c. В отличие от алгоритма 3 значения параметров N_b и M_c могут быть различны. Значения параметров N_c и M_c выбираются так, чтобы N_c элементов x не вытеснялись из L_2.

Цикл с индуктивной переменной j_r разбивает подматрицу матрицы A и подвектор вектора x на подматрицу размерности $M_c \times N_r$ и подвектор размерности N_r. Строки указанной подматрицы и полученный в результате разбиения подвектор участвуют в векторных операциях. Вследствие этого N_r зависит от размерности векторных регистров ГП. Пусть $\mathbb{C}_{\mathrm{L}_1} = C_{\mathrm{L}_1}/S_{\mathrm{DATA}}$. Каждые $\lceil 4\mathbb{C}_{\mathrm{L}_1}/N_r \rceil$ итераций подматрица $A(i_c : i_c + M_c - 1, J + D4\mathbb{C}_{\mathrm{L}_1} : J + (D+1)4\mathbb{C}_{\mathrm{L}_1} - 1)$ матрицы A и подвектор $x(J + D4\mathbb{C}_{\mathrm{L}_1} : J + (D+1)4\mathbb{C}_{\mathrm{L}_1} - 1)$ вектора x загружаются в L_1. Загрузка $4\mathbb{C}_{\mathrm{L}_1}/S_{\mathrm{DATA}}$ элементов матрицы A и $4\mathbb{C}_{\mathrm{L}_1}/S_{\mathrm{DATA}}$ элементов вектора x позволяет усреднить задержку доступа к памяти используемой целевой аппаратной платформы.

В отличие от алгоритма 3 не выполняется дополнительного разбиения подвектора вектора y и подматрицы матрицы A. Далее это используется для определения значений параметра D.

Использование данных. Рассмотрим цикл с индуктивной переменной j_r алгоритма 7. Тело цикла вычисляет N_r элементов вектора y, используя N_r элементов матрицы A и один элемент вектора x. Следовательно, элемент вектора x используется последовательными итерациями описываемого цикла. Элемент вектора x используется N_b/N_r раз, так как он участвует в вычислениях, выполняемых каждой итерацией данного цикла. Элементы матрицы A и вектора y не используются повторно.

Таблица 2.2. $y = A^{\mathsf{T}} \bar{\otimes} x \bar{\oplus} y$.

Номер цикла	Данные	Размер	Количество повторных использований
5	x	1	N_b/N_r
4	y	N_b	M_c
3	x	M_c	N_c/N_b
2	y	N_c	M/M_c
1	x	M	N/N_c

Проанализировав остальные циклы, можно определить какие данные должны храниться в кэш-памяти, чтобы уменьшить количество промахов кэша и увеличить производительность программы. Табл. 2.2 и табл. 2.3 содержат размеры используемых данных и указывают, сколько раз они используются в алгоритмах 7 и 8, соответственно. Применим эту информацию для определения значений параметров N_c, M_c, N_b и N_r, чтобы уменьшить количество вытеснений часто используемых данных из кэш-памяти.

Таблица 2.3. $y = A \bar{\otimes} x \bar{\oplus} y$. Анализ использования данных.

Номер цикла	Данные	Размер	Количество повторных использований
3	y	M_c	N_c/N_r
2	x	N_c	M/M_c
1	y	M	N/N_c

Выведены формулы, связывающие размеры блоков матриц и векторов N_r, N_b, N_c, M_c, а также шаг предвыборки D алгоритма 7 с техническими характеристиками реального процессора.

Определение значений параметров алгоритма 7. Выведены формулы, связывающие параметры N_r, N_b, N_c, M_c и D алгоритма 7 с техническими характеристиками реального процессора. Согласно табл. 2.2 указанные параметры определяют размеры и количество повторных использований элементов векторов x и y. Кроме этого, значения параметров M_c и N_b определяют количество элементов матрицы A, загружаемых в L_1 на каждой итерации цикла с индуктивной переменной j_b. Значение параметра D определяет шаг предвыборки.

Рассмотрим цикл с индуктивной переменной j_r, граничным значением N_b и шагом N_r. Согласно табл. 2.2 значения параметров N_b и N_r определяют количество повторных использований элемента вектора x и количество вычисляемых элементов вектора y. Так как N_r используется в качестве шага описываемого цикла и каждая VMMA может вычислить N_{VEC} умножений и сложений, то $N_r = N_{\mathrm{VEC}}$. Значение N_b должно быть достаточно большим, чтобы избежать простоя конвейера ГП в процессе вычисления элементов вектора y. В то же время, N_b определяет число используемых регистров, количество которых ограничено.

Каждая итерация цикла с индуктивной переменной j_r вычисляет каждый из N_r элементов вектора y, используя VMMA для умножения строки подматрицы матрицы A и элемента вектора x. Так как L_{VMMA} является наименьшим количеством тактов процессора, после выполнения которых могут быть повторно использованы зависимые по данным инструкции VMMA, то L_{VMMA} тактов должно быть выполнено, чтобы вычислить новое значение элемента вектора y. В процессе выполнения этих тактов нужно начать выполнение по крайней мере $N_{\mathrm{VMMA}} L_{\mathrm{VMMA}}$ VMMA инструкций, чтобы избежать простоя конвейера процессора. Следовательно,

$N_{\text{VMMA}}L_{\text{VMMA}}N_{\text{VEC}}$ элементов вектора y должно быть вычислено и $N_b \geq N_{\text{VFMA}}L_{\text{VFMA}}N_{\text{VEC}}$.

Согласно табл. 2.2 увеличение значения параметра N_b увеличивает количество повторно используемых элементов вектора y и количество повторных использований элементов вектора x. Чтобы избежать вытеснения данных из векторных регистров в память, рассмотрим какое количество векторных регистров требуется для хранения всех данных, используемых в цикле с индуктивной переменной j_r. $2 + N_b/N_{\text{VEC}}$ регистров требуется для хранения N_b элементов вектора y, копии элемента вектора x и N_{VEC} элементов матрицы A. Так как ГП имеет N_{REG} векторных регистров, то $N_b = (N_{\text{REG}} - 2)N_{\text{VEC}}$. Учитывая рассуждения о задержке VMMA, $N_b = \max(N_{\text{VFMA}}L_{\text{VFMA}}N_{\text{VEC}}, (N_{\text{REG}} - 2)N_{\text{VEC}})$. Обозначим $\mathbb{C}_{\text{L}_\text{i}} = C_{\text{L}_\text{i}}/S_{\text{DATA}}$. Для того чтобы выполнялись предположения относительно кэш-памяти процессора, представленные в разделе 2.1, ограничим значение параметра N_b как $N_b \leq N_{\text{L}_1}\mathbb{C}_{\text{L}_1}$. Таким образом,

$$N_r = N_{\text{VEC}}, \tag{2.3}$$

$$\begin{aligned} N_b = \min(\, &\max(N_{\text{VFMA}}L_{\text{VFMA}}N_{\text{VEC}}, (N_{\text{REG}} - 2)N_{\text{VEC}}), \\ &\lceil N_{\text{L}_1}\mathbb{C}_{\text{L}_1}/N_{\text{VEC}} \rceil\, N_{\text{VEC}}). \end{aligned} \tag{2.4}$$

Выведем формулы для вычисления значений параметров N_c и M_c. Согласно табл. 2.2 они определяют размеры и количество повторных использований элементов векторов x и y. $M_c \times N_b$ элементов матрицы A загружаются в кэш-память на каждой итерации цикла с индуктивной переменной j_b. Для хранения элементов матрицы A использован L_1, так как в отличие от L_2 он чаще всего виртуально индексируемый [154]. Виртуальная индексируемость позволяет гарантировать, что $M_c \times N_b$ элементов матрицы A не будут вытеснены из L_1, если $M_c \leq W_{\text{L}_1}$ (см. раздел 2.1). Увеличение M_c уменьшает издержки, связанные с выполнением инструкций предвыборки, и приближает значение задержки доступа памяти к ее среднему значению [36, 37, 155]. Согласно табл. 2.2 значение параметра M_c прямопро-

порционально тому, сколько раз используются элементы вектора y, хранящиеся на векторных регистрах и использующиеся каждой итерацией цикла с индуктивной переменной j_b. Так как элементы вектора y могут быть загружены на регистры перед выполнением цикла с индуктивной переменной i_b, M_c элементов вектора x используются между двумя последовательными итерациями цикла с индуктивной переменной j_b и L_1 обычно виртуально индексируем [154], то $M_c = W_{\mathrm{L}_1} - 1$.

Согласно табл. 2.2 увеличение значений параметра N_c увеличивает количество повторных использований элемента вектора x. Если $N_c > N_{\mathrm{L}_1}\mathbb{C}_{\mathrm{L}_1}$, то некоторые элементы матрицы A будут вытеснены из L_1 в L_2. Чтобы сохранить элементы вектора y между двумя последовательными итерациями цикла с индуктивной переменной i_c, по крайней мере одна кэш линия в каждом множестве L_2 должна быть свободна. Следовательно, $M_c = \min(W_{\mathrm{L}_2} - 1, W_{\mathrm{L}_1} - 1)$ и $N_c = N_{\mathrm{L}_2}\mathbb{C}_{\mathrm{L}_2}$. Присвоим M_c значение $\min(W_{\mathrm{L}_2} - 1, W_{\mathrm{L}_1})$, чтобы использовать большие значения M_c. Такой подход может привести к вытеснению некоторых элементов вектора x в L_2. Предполагается, что M_c возможно вытесненных элементов x могут храниться в L_2, так как $M_c \leq W_{\mathrm{L}_1} \ll N_{\mathrm{L}_1}\mathbb{C}_{\mathrm{L}_1} \ll N_{\mathrm{L}_2}\mathbb{C}_{\mathrm{L}_2}$. Следовательно, значения параметров M_c и N_c могут быть вычислены следующим образом:

$$M_c = \min(W_{\mathrm{L}_2} - 1, W_{\mathrm{L}_1}), \tag{2.5}$$

$$N_c = N_{\mathrm{L}_2}\mathbb{C}_{\mathrm{L}_2}. \tag{2.6}$$

Чтобы вычислить шаг предвыборки D, применяется уравнение 2.1. Вследствие этого $D = \lceil L_{\mathrm{prefetch}}/B \rceil$, где B — это количество тактов процессора, выполняемых телом цикла с индуктивной переменной j_b. Рассмотрим цикл с индуктивной переменной i_b. Чтобы запустить выполнение всех VMMA инструкций, требуется $\lceil N_b/(N_{\mathrm{VEC}}N_{\mathrm{VMMA}}) \rceil$ тактов процессора. Кроме этого, L_{VLOAD} тактов должно быть выполнено, чтобы загрузить элементы вектора x на векторный регистр. Для запуска всех инструкций предвы-

борки цикла с индуктивной переменной j_b требуется $\lceil M_c \lceil N_b/\mathbb{C}_{\mathrm{L}_1} \rceil / N_{\mathrm{prefetch}} \rceil$ тактов. Следовательно, B может быть вычислен как $\lceil M_c \lceil N_b/C_{\mathrm{L}_1} \rceil / N_{\mathrm{prefetch}} \rceil + M_c(\lceil N_b/(N_{\mathrm{VEC}} N_{\mathrm{VMMA}}) \rceil + L_{\mathrm{VLOAD}})$. Следовательно,

$$\begin{aligned} D = &\lceil L_{\mathrm{prefetch}}/(\lceil M_c \lceil N_b S_{\mathrm{DATA}}/C_{\mathrm{L}_1} \rceil / N_{prefetch} \rceil + \\ &M_c(\lceil N_b/N_{VEC} N_{VMMA} \rceil + L_{VLOAD})) \rceil. \end{aligned} \tag{2.7}$$

Таким оборазом, в случае алгоритма 7 формулы имеют вид:

$$N_b = \min(\max(N_{\mathrm{VFMA}} L_{\mathrm{VFMA}} N_{\mathrm{VEC}}, (N_{\mathrm{REG}} - 2) N_{\mathrm{VEC}}), \left\lceil \frac{N_{\mathrm{L}_1} C_{\mathrm{L}_1}}{N_{\mathrm{VEC}} S_{\mathrm{DATA}}} \right\rceil N_{\mathrm{VEC}}),$$

$$N_r = N_{\mathrm{VEC}}, M_c = \min(W_{\mathrm{L}_2} - 1, W_{\mathrm{L}_1}), N_c = N_{\mathrm{L}_2} C_{\mathrm{L}_2}/S_{\mathrm{DATA}},$$

$$\begin{aligned} D = &\lceil L_{\mathrm{prefetch}}/(\lceil M_c \lceil N_b S_{\mathrm{DATA}}/C_{\mathrm{L}_1} \rceil / N_{prefetch} \rceil + \\ &M_c(\lceil N_b/N_{VEC} N_{VFMA} \rceil + L_{VLOAD})) \rceil. \end{aligned}$$

В разделе 4.3 показано, что во всех случаях выведенные формулы позволяют получить значения параметров близкие к значениям, полученным ручной настройкой.

Определение значений параметров алгоритма 8. Выведены формулы, связывающие параметры N_r, N_c, M_c и D алгоритма 8 с техническими характеристиками реального процессора. Согласно табл. 2.3 указанные параметры влияют на размеры и количество повторных использований элементов векторов x и y. Значения параметра M_c определяют количество элементов матрицы A, загружаемых в L_1 каждой итерацией цикла с индуктивной переменной j_r. Значение параметра D определяет шаг предвыборки.

Рассмотрим цикл с индуктивной переменной j_r алгоритма 8. Так как значение параметра N_r определяет только шаг описываемого цикла и каждая инструкция VMMA может вычислить N_{VEC} умножений и сложений, то $N_r = N_{\mathrm{VEC}}$.

Как и в случае параметра N_b алгоритма 7, используется наибольшее из возможных значений параметра M_c, позволяющее избежать про-

стоя в конвейере процессора во время вычисления новых значений временного массива векторов acc телом цикла с индуктивной переменной j_r. Определим $\mathbb{C}_{\mathrm{L_i}} = C_{\mathrm{L_i}}/S_{\mathrm{DATA}}$. Следовательно, $M_c N_r \geq N_{\mathrm{VMMA}} L_{\mathrm{VMMA}} N_{\mathrm{VEC}}$. Для того чтобы $M_c \times 4\mathbb{C}_{\mathrm{L_1}}$ элементов матрицы A и $4\mathbb{C}_{\mathrm{L_1}}$ элементов вектора x загружались в $\mathrm{L_1}$ рассматриваемым циклом, значение M_c должно быть не больше $W_{\mathrm{L_1}}$. Такие значения позволяют сохранить загружаемые данные в $\mathrm{L_1}$ без их вытеснения (см. раздел 2.1). Следовательно, $N_r \geq N_{\mathrm{VEC}} \lceil N_{\mathrm{VFMA}} L_{\mathrm{VFMA}}/(W_{\mathrm{L_1}} - 1) \rceil$ и $M_c = \lceil N_{\mathrm{VFMA}} L_{\mathrm{VFMA}} N_{\mathrm{VEC}}/N_r \rceil$.

Согласно табл. 2.3 увеличение значения параметра N_c повышает количество повторных использований элементов вектора y. Однако если $N_c > N_{\mathrm{L_1}} \mathbb{C}_{\mathrm{L_1}}$, некоторые элементы вектора x будут вытеснены элементами вектора y из $\mathrm{L_1}$ в $\mathrm{L_2}$. Чтобы избежать вытеснения элементов вектора x из $\mathrm{L_2}$ между двумя последовательными итерациями цикла с индуктивной переменной i_c, не менее чем одна линия кэша каждого множества $\mathrm{L_2}$ должна быть свободна. Следовательно, $N_r \geq N_{\mathrm{VEC}} \lceil N_{\mathrm{VFMA}} L_{\mathrm{VFMA}}/\min(W_{\mathrm{L_1}} - 1, W_{\mathrm{L_2}} - 1) \rceil$. Так как редукция M_c элементов вектора y амортизируется N_c/N_r итерациями цикла с индуктивной переменной i_c, целесообразно уменьшить значение N_r и увеличить значение M_c. Следовательно, N_r можно вычислить как $N_r = N_{\mathrm{VEC}} \lceil N_{\mathrm{VFMA}} L_{\mathrm{VFMA}}/\min(W_{\mathrm{L_1}}, W_{\mathrm{L_2}} - 1) \rceil$. Такой подход может привести к вытеснению некоторых элементов вектора x из $\mathrm{L_1}$. Однако, элементы вектора x могут содержаться в $\mathrm{L_2}$ между двумя последовательными итерациями цикла с индуктивной переменной i_c, так как $M_c < W_{\mathrm{L_2}}$. Следовательно,

$$N_r = N_{\mathrm{VEC}} \lceil N_{\mathrm{VFMA}} L_{\mathrm{VFMA}}/\min(W_{\mathrm{L_1}}, W_{\mathrm{L_2}} - 1) \rceil, \tag{2.8}$$

$$M_c = \lceil N_{\mathrm{VFMA}} L_{\mathrm{VFMA}} N_{\mathrm{VEC}}/N_r \rceil, \tag{2.9}$$

$$N_c = N_{\mathrm{L_2}} \mathbb{C}_{\mathrm{L_2}}. \tag{2.10}$$

Чтобы вычислить шаг предвыборки D воспользуемся уравнением 2.1. Следовательно, $D = \lceil L_{\mathrm{prefetch}}/B \rceil$, где B — количество тактов процессора,

требуемых для выполнения цикла с индуктивной переменной j_r. Рассмотрим $4\mathbb{C}_{\mathrm{L}_1}/N_r$ итераций данного цикла. Чтобы начать выполнение всех инструкций предвыборки, требуется $\lceil (4\mathbb{C}_{\mathrm{L}_1}(M_c+1))/(N_{\mathrm{prefetch}}\mathbb{C}_{\mathrm{L}_1})\rceil =$ $\lceil 4(M_c+1)/N_{\mathrm{prefetch}}\rceil$ тактов. $4\mathbb{C}_{\mathrm{L}_1}L_{\mathrm{VLOAD}}/N_r$ тактов необходимо, чтобы загрузить элементы вектора x на векторный регистр. Чтобы начать выполнение всех VMMA инструкций, требуется $4\mathbb{C}_{\mathrm{L}_1}\left\lceil (M_cN_r)/(N_{\mathrm{VMMA}}N_{\mathrm{VEC}})\right\rceil/N_r$ циклов. Следовательно,

$$\begin{aligned} B = &\lceil 4(M_c+1)/N_{\mathrm{prefetch}}\rceil + \\ &4\mathbb{C}_{\mathrm{L}_1}(\lceil (M_cN_r)/(N_{\mathrm{VMMA}}N_{\mathrm{VEC}})\rceil + L_{\mathrm{VLOAD}})/N_r, \end{aligned} \tag{2.11}$$

$$\begin{aligned} D = &\lceil L_{\mathrm{prefetch}}/(\lceil 4(M_c+1)/N_{\mathrm{prefetch}}\rceil + \\ &4\mathbb{C}_{\mathrm{L}_1}(\lceil (M_cN_r)/(N_{\mathrm{VFMA}}N_{\mathrm{VEC}})\rceil + L_{\mathrm{VLOAD}})/N_r)\rceil. \end{aligned} \tag{2.12}$$

Таким образом, в случае алгоритма 8 формулы имеют вид:

$$N_r = N_{\mathrm{VEC}}\left\lceil \gamma/\min(W_{\mathrm{L}_1}, W_{\mathrm{L}_2}-1)\right\rceil, M_c = \left\lceil \gamma/N_r\right\rceil,$$

$$N_c = N_{\mathrm{L}_2}C_{\mathrm{L}_2}/S_{\mathrm{DATA}},$$

где $\gamma = N_{\mathrm{VMMA}}L_{\mathrm{VMMA}}N_{\mathrm{VEC}}$,

$$\begin{aligned} D = &\lceil L_{\mathrm{prefetch}}/(\lceil 4(M_c+1)/N_{\mathrm{prefetch}}\rceil + \\ &4\mathbb{C}_{\mathrm{L}_1}(\lceil (M_cN_r)/(N_{\mathrm{VFMA}}N_{\mathrm{VEC}})\rceil + L_{\mathrm{VLOAD}})/N_r)\rceil. \end{aligned}$$

В разделе 4.3 показывается, что во всех случаях выведенные формулы позволяют получить значения параметров близкие к значениям, полученным ручной настройкой.

Учитывая, что команды для работы с памятью могут выполняться ГП одновременно с командами арифметики с плавающей точкой, из приведенных ранее рассуждений о выборе значений параметров алгоритма 2.2 и алгоритма 2.3 следует следующее утверждение.

Утверждение 2.2. Если данные могут быть мгновенно загружены из памяти на векторные регистры, то существуют значения параметров N_r,

N_b и значения параметров M_c и N_r, позволяющие избежать простаивания конвейера векторных инструкций ГП во время выполнения алгоритма 7 и цикла с индуктивной переменной i_c алгоритма 8, соответственно.

2.4. Алгоритм сокращения времени выполнения свертки тензоров

Построим алгоритм сокращения времени выполнения ТС на основе алгоритмов вычисления MMM и MVM, обобщенных на замкнутые полукольца с элементами из множества вещественных чисел (см. разделы 2.2 и 2.3). В ходе выполнения алгоритма используются оптимизации циклов, применяемые к *полиэдральной модели* (*polyhedral model*) или *полиэдральному представлению* (*polyhedral representation*) программы, являющейся математическим фреймворком для моделирования и оптимизации доступа к памяти, производимого в группах вложенных циклов, не рассматривая отдельные вычисления [156].

Для определения компонент полиэдрального представления программы используются целочисленные многогранники и отношения Пресбургера [157, 158]. Рассмотрим определение отношений Пресбургера, представленное в работе [158].

Определение 2.1. Сигнатура $\Sigma = (P, F, C, \rho)$ — набор множеств, где P — множество предикатных символов; F — множество функциональных символов; C — множество символов констант, являющихся функциональными символами арности 0; ρ — функция, сопоставляющая элементам R и F их арность.

Определение 2.2. Терм — либо символ переменной, либо $f(t_1, \ldots, t_n)$, где f — функциональный символ арности n, а $t_1, \ldots, t_n$ — термы.

Определение 2.3. Формула логики первого порядка определяется как:

1. $E_1 \wedge E_2$, где E_i — формула логики первого порядка;

2. $E_1 \vee E_2$, где E_i — формула логики первого порядка;

3. $E_1 \to E_2$, где E_i — формула логики первого порядка;

4. $\neg E$, где E — формула логики первого порядка;

5. $\forall v E$, где v — переменная, E — формула логики первого порядка;

6. $\exists v E$, где v — переменная, E — формула логики первого порядка;

7. $P(t_1, \ldots, t_n)$, где P — предикатный символ арности n, а $t_1, \ldots, t_n$ — термы.

Определение 2.4. Язык Пресбургера — сигнатура, имеющая только один предикатный символ $\leq (t_1, t_2)$ и следующие функциональные символы:

1. $+(t_1, t_2)$, где t — терм;

2. $-(t_1, t_2)$;

3. множество символов констант для $\forall d \in \mathbb{N}$;

4. множество функциональных символов вида: $\forall d \in \mathbb{N} \lceil\rceil(t, d) = \lceil t/d \rceil$, где t — терм;

5. множество символов констант c_i.

Определение 2.5. Отношение Пресбургера — формула логики первого порядка, определенная на основе языка Пресбургера.

Рассмотрим множество инструкций программы, для которого может быть построено полиэдральное представление.

Определение 2.6. SCoP (Static Control Part) — максимальное по включению множество инструкций, представимых в следующем виде:

- инструкции не содержат команды передачи управления, за исключением операторов if и for.

- у каждого цикла только одна индуктивная переменная, константный шаг, ограничения на индуктивные переменные представимы в виде неравенств с аффинными функциями.

- глобальные переменные инвариантны во время выполнения инструкций SCoP.

Для каждого утверждения рассматриваемой программы полиэдральная модель описывает области итерирования, аффинные планы и функции доступа к памяти. В качестве примера программы, являющейся SCoP, рассмотрим реализацию MMM:

```
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        for (p = 0; p < K; p++)
S:          C[i][j] += A[i][p] * B[p][j];
```

Определение 2.7. Пусть $\mathbb{I}$ — вектор, состоящий из индуктивных переменных SCoP; $\mathbb{I}_{\mathrm{gp}}$ — вектор глобальных параметров SCoP. В этом случае область итерирования инструкции S определяется как $\{S(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}) \mid f\left(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}\right)\}$, где $S(\mathbb{I}, \mathbb{I}_{\mathrm{gp}})$ — точка целочисленного многогранника; $f(\mathbb{I}, \mathbb{I}_{\mathrm{gp}})$ — отношение Пресбургера, свободные пременные которых — элементы векторов $\mathbb{I}$ и $\mathbb{I}_{\mathrm{gp}}$.

Область итерирования для утверждения S рассматриваемой реализации MMM имеет вид $\{S(i, j, p) \mid 0 \leq i \leq M \wedge 0 \leq j \leq N \wedge 0 \leq p \leq K\}$.

Определение 2.8. Пусть $\mathbb{I}$ — вектор, состоящий из индуктивных переменных SCoP; $\mathbb{I}_{\mathrm{t}}$ — вектор временных следов; $\mathbb{I}_{\mathrm{gp}}$ — вектор глобальных параметров SCoP. В этом случае аффинный план инструкции S определяется как $\{S(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}) \to (\mathbb{I}_{\mathrm{t}}) \mid f\left(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}\right)\}$, где $S(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}) \to (\mathbb{I}_{\mathrm{t}})$ — точка целочисленного многогранника; $f(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}, \mathbb{I}_{\mathrm{t}})$ — отношение Пресбургера, свободные переменные которых являются элементами векторов $\mathbb{I}$, $\mathbb{I}_{\mathrm{gp}}$ и $\mathbb{I}_{\mathrm{t}}$.

Аффинный план для утверждения S рассматриваемой реализации MMM имеет вид $\{S(i, j, p) \to (t_1, t_2, t_3) \mid t_1 = i \wedge t_2 = j \wedge t_3 = p\}$.

Определение 2.9. Пусть $\mathbb{I}$ — вектор, состоящий из индуктивных переменных SCoP; $\mathbb{I}_{\mathrm{gp}}$ — вектор глобальных параметров SCoP; M — массив,

к которому происходит обращение инструкции S (чтение либо запись); $\mathbb{I}_{\mathrm{m}}$ — вектор значений индексов массива M, используемых в инструкции S. В этом случае функция доступа к памяти инструкции S для массива M определяется как $\{S(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}) \to M(\mathbb{I}_{\mathrm{m}}) \mid f(\mathbb{I}, \mathbb{I}_{\mathrm{gp}})\}$, где $S(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}) \to M(\mathbb{I}_{\mathrm{m}})$ — точка целочисленного многогранника; $f(\mathbb{I}, \mathbb{I}_{\mathrm{gp}}, \mathbb{I}_{\mathrm{m}})$ — отношение Пресбургера, свободные переменные которых — элементы векторов $\mathbb{I}$, $\mathbb{I}_{\mathrm{gp}}$ и $\mathbb{I}_{\mathrm{m}}$.

Функции доступа к памяти для утверждения S рассматриваемой реализации MMM:

$\{S(i, j, p) \to A(i_1, i_2) \mid i_1 = i \wedge i_2 = p\}$

$\{S(i, j, p) \to C(i_1, i_2) \mid i_1 = i \wedge i_2 = j\}$

$\{S(i, j, p) \to C(i_1, i_2) \mid i_1 = i \wedge i_2 = j\}$

Для сокращения времени выполнения TC автором вводится определение TC-подобного ядра, частным случаем которого является свертка тензоров [3].

Определение 2.10. *TC-подобное ядро* (*Tensor Contraction like kernel, TC-like kernel*) — множество полностью вложенных циклов, такое что.

1. Ядро удовлетворяет требованиям полиэдральной модели.

2. Без ограничения общности TC-подобное ядро содержит три непустых множества циклов, имеющих одну индуктивную переменную и единичный шаг. Данные множества образуют три непустых набора $I = i_0 \ldots i_{r-1}$, $J = j_0 \ldots j_{s-1}$ и $P = p_0 \ldots p_{t-1}$.

3. Тело цикла TC-подобного ядра, имеющего наибольшую глубину, может быть представлено в виде $\mathrm{C}_{\pi_C(IJ)} = \mathrm{E}(\mathrm{A}_{\pi_A(IP)}, \mathrm{B}_{\pi_B(PJ)})$, где $\mathrm{A}_{\pi_A(IP)}$, $\mathrm{B}_{\pi_B(PJ)}$, $\mathrm{C}_{\pi_C(IJ)}$ - обращения к тензорам A, B, C, соответственно; $\pi_C(IJ)$, $\pi_A(IP)$ и $\pi_B(PJ)$ — перестановки индексов; E — выражение, содержащее чтения из тензоров $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ и произвольное количество чтений из констант.

Для того, чтобы SCoP являлся TC-подобным ядром, можно привести следующие достаточные условия в соответствии с его определением: SCoP

содержит только одно утверждение S; имеется t зависимостей по управлению, t антизависимостей, t зависимостей по выходу; все зависимости определяются как $\forall i \in 0 \dots t-1\ S((p_0, \dots, p_i, n_{p_{i+1}}-1, \dots, n_{p_{t-1}}-1) * \pi(IJ)) \rightarrow S((p_0, \dots, p_i+1, 0, \dots) * \pi(IJ))$, где $t_1 * t_2$ — упорядоченный набор, состоящий из элементов упорядоченных наборов t_1 и t_2, с сохранением отношений порядка, установленных в t_1 и t_2.

В соответствии с определением ТС-подобного ядра циклы I и J могут быть переставлены без нарушения зависимостей. Если ассоциативная операция используется для вычисления C, то можно переставлять циклы P без нарушения зависимостей. Все чтения из памяти за исключением констант имеют вид $S(\dots) \rightarrow \mathrm{A}_{\pi_A(IP)}$, $S(\dots) \rightarrow \mathrm{B}_{\pi_B(PJ)}$, $S(\dots) \rightarrow \mathrm{C}_{\pi_C(IJ)}$. Только последнее обращение к памяти выполняет запись в нее.

Для распознавания ТС-подобного ядра достаточно проверить описанные ранее достаточные условия и обращения к памяти. Проверяя последнее обращение к памяти, можно получить набор $\pi_C(IJ)$. Зависимости программы содержат информацию о наборе P. Проверяя остальные обращения к памяти, можно вычислить наборы I и J и проверить ассоциативность операции, использующейся для вычисления C.

Рассмотрим алгоритм 9, позволяющий выполнить сокращение времени выполнения ТС-подобного ядра, используя описанный в разделе 2.2 подход к вычислению МММ. Для простоты будем предполагать, что тензоры ТС-подобного ядра логически представлены в виде матриц, используя формулы из раздела 1.3.

В процессе выполнения первых трех шагов алгоритма 9 создаются три цикла с индуктивными переменными i, j и p, соответствующие циклам алгоритма 6, используя перестановки циклов и разбиение на блоки. На четвертом шаге используется разбиение на блоки для получения циклов с индуктивными переменными i_c, j_c и p_c. На пятом шаге выделяются безусловные области итерирования циклов с индуктивными переменными

i_r и j_r. Это позволяет выполнить полную размотку выделенных областей. На шестом шаге в программу добавляются временные массивы A_c и B_c, хранящие элементы матриц A и B с целью дальнейшего использования в арифметических операциях. Генерируются инструкции, копирующие элементы матриц A и B в во временные массивы A_c и B_c, соответственно. На седьмом шаге выполняется векторизация размотанных циклов.

Алгоритм 9: Оптимизация TC-подобного ядра

Входные данные:

Утверждение S полиэдрального представления программы. S является частью TC-подобного ядра и, вследствие этого, представимо в виде C[i][j] = E(A[i][p], B[p][j], C[i][j]), где i, j, p — индукционные переменные циклов; A[i][p], B[p][j], C[i][j] — обращения к матрицам A, B, C, соответственно; E — выражение, содержащее операции чтения из матриц A, B, C.

1. Отождествить каждый цикл с его индукционной переменной.
2. Переставить циклы так, чтобы i, j, p приобрели наибольшую глубину и следующий порядок: j, p и i, где i имеет наименьшую глубину.
3. Разбить i, j, p на блоки размера M_c, N_c, K_c, соответственно; получить циклы i_c, j_c и p_c; переставить i_c и p_c.
4. Разбить i_c, j_c, p_c на блоки размера M_r, N_r и 1, соответственно; получить циклы i_r, j_r, p_r; удалить p_r.
5. Выделить безусловные области итерирования i_r , j_r и выполнить их размотку.
6. Выполнять копирование элементов матриц A и B во временные массивы A_c, B_c.
7. Векторизовать код в p_c.

Выходные данные: оптимизированный код.

Для каждого отдельно рассматриваемого TC-подобного ядра размерность тензоров, участвующих в тензорной свертке, не влияет на выполнение алгоритма 9. Вследствие этого алгоритм 9 имеет константную временную сложность относительно размерности тензоров. Алгоритм имеет некон-

стантную временную сложность относительно рангов тензоров, от которых зависит количество операций с индексами матриц, логически представляющих тензоры с помощью формул из раздела 1.3.

В случае тривиальной реализации MMM алгоритм 9 позволяет получить реализацию MMM, описываемую алгоритмом 6. Для нахождения значений параметров алгоритма 9 используются формулы, представленные в разделе 2.2. Описанный подход схож с методом, применяемым в фреймворке TBLIS. TBLIS содержит готовые реализации TC, созданные с использованием ручной настройки и частей оптимизированных реализаций MMM библиотеки BLIS (см. раздел 1.3). В данный момент такие части не описываются интерфейсом BLAS и доступны только в библиотеке BLIS, поддерживающей ограниченное количество целевых аппаратных платформ. В большинстве случаев рассматриваемые части реализации MMM создаются вручную специалистом на языке ассемблера. Применение описанного подхода позволяет автоматически получить оптимизированную реализацию TC, включая указанные части реализаций MMM.

Если предположить, что один из наборов индуктивных переменных I и J TC-подобного ядра пуст, использованные в случае алгоритма 9 оптимизации циклов могут быть применены для получения реализаций MVM, описанных алгоритмами 7 и 8. Отличием является необходимость генерации инструкций предвыборки вместо инструкций копирования во временные массивы. В случае алгоритма 8 перед выполнением векторизации необходимо сгенерировать операцию редукции.

2.5. Выводы

Вторая глава посвящена разработке оригинального алгоритма автоматического сокращения времени выполнения тензорных операций. Для решения более широкого класса задач выполнено обобщение алгоритмов

вычисления MMM и MVM на замкнутые полукольца с элементами из множества вещественных чисел. Вычисление представленных алгоритмов для выполнения MVM может быть смоделировано на ГП для вывода формул, определяющих значения их параметров. Построен алгоритм автоматического сокращения времени выполнения свертки тензоров за константное время относительно размерности измерений тензоров и без доступа к целевой аппаратной платформе. В ходе выполнения алгоритма сокращение времени выполнения свертки тензоров сводится к сокращению времени выполнения обобщенных матричных и матрично-векторных произведений. Для этого используются оптимизации циклов доступные в большинстве современных компиляторов. Описывается ГП, использующийся для нахождения значений параметров алгоритмов. Предложенный ГП позволяет смоделировать выполнение предвыборки данных в кэш-память с целью определения ее шага и других параметров; обобщить FMA на замкнутые полукольца с элементами из множества вещественных чисел для сокращения времени выполнения решений общей задачи о путях для случая замкнутых полуколец с элементами из множества вещественных чисел. Результаты, представленные в этой главе, опубликованы в работах [159–161].

Глава 3

Программная система автоматической оптимизации тензорных операций

Данная глава посвящена разработке программной системы автоматической оптимизации тензорных операций (ПС АОТО) на основе моделей, алгоритмов и формул, предложенных во второй главе. ПС АОТО автоматически оптимизирует время выполнения тензорных операций без доступа к целевой аппаратной платформе и без выполнения автонастройки и ручной настройки. Приводится описание архитектуры ПС АОТО. Рассматриваются возможные реализации ПС АОТО. Предложен метод автоматического распараллеливания программ ПС АОТО на многоядерных процессорах общего назначения с общей памятью.

ПС АОТО позволяет выполнить следующее:

1. Компиляция программы.
2. Построение полиэдрального представления.
3. Распознавание TC-подобных ядер и их оптимизация.

3.1. Архитектура программной системы автоматической оптимизации тензорных операций

ПС АОТО имеет три взаимодействующие между собой части, представленные на рис. 3.1.

1. Фронтенд, выполняющий лексический, синтаксический и семантический анализ программы, а также построение промежуточного кода, являющегося программой в кодах виртуальной машины на языке эквивалентном исходному.

2. Инфраструктура для полиэдральных оптимизаций, выполняющая построение полиэдрального представления и его модификации, такие как оптимизация циклов и доступов к памяти. В результате работы инфраструктуры для полиэдральных оптимизаций создается промежуточный код, соответствующий оптимизированному полиэдральному представлению. В рамках данной работы была создана и реализована оптимизация полиэдрального представления программы, позволяющая распознать ТС-подобное ядро, определить значения реализации ТС-подобного ядра, оптимизировать время выполнения ТС-подобного ядра (см. раздел 2.4).

3. Бэкенд, выполняющий оптимизацию промежуточного кода программы, используя низкоуровневые и высокоуровневые оптимизации. Для выполнения оптимизаций, в частности, применяется инфраструктура для полиэдральных оптимизаций. В дальнейшем бэкенд генерирует ассемблерный код для одной из поддерживаемых целевых аппаратных платформ.

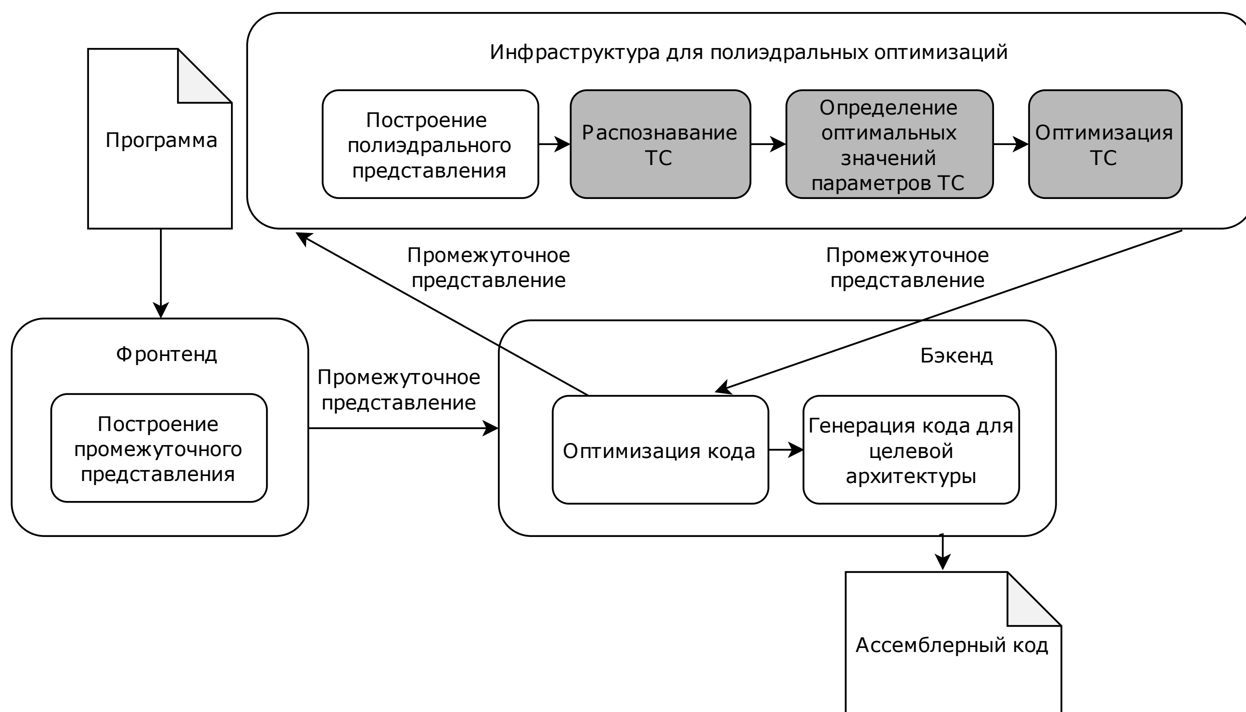


Рис. 3.1. Диаграмма программной системы; темным цветом выделены части, созданные в рамках данной работы.

В силу того что, в большинстве случаев фронтенды используемого компилятора генерируют промежуточный код на одном выбранном заранее языке, оптимизации промежуточного кода могут применяться независимо от языка, на котором написана компилируемая программа. В дальнейшем оптимизированный код может быть преобразован в машинный код для любых поддерживаемых бэкендом архитектур процессоров. Вследствие этого представленная структура ПС АОТО позволяет обеспечить тестируемость описанных частей и их взаимозаменяемость. Это облегчает поддержку различных языков, используемых для написания оптимизируемых программ, и обеспечивает генерацию машинного кода для множества целевых аппаратных платформ.

3.2. Реализация программной системы автоматической оптимизации тензорных операций

Детали реализации. В рамках данной работы фреймворк Polly применен в качестве инфраструктуры для построения полиэдрального представления, а также в качестве основы для реализации распознавания TC-подобного ядра и оптимизации его времени выполнения, описанных в разделе 2.4. Для случая MMM, обобщенных на замкнутые полукольца с элементами из множества вещественных чисел, указанная оптимизация была внедрена в основной код Polly проекта LLVM.

Фреймворк Polly позволяет выполнить оптимизации промежуточного представления программы независимо от выбранного фронтенда. С целью использования возможностей многоядерных процессоров фреймворк Polly генерирует процедуры библиотеки libiomp, реализующей технологию параллельного программирования OpenMP. Polly использует библиотеку ISL (Integer Set Library), по умолчанию применяющую алгоритм направленный на одновременное увеличение крупнозернистого параллелизма и локальности данных [162–164]. Описанный алгоритм не влияет на реализацию TC. С целью повышения эффективности использования верхних уровней кэш-памяти процессора для случаев отличных от MMM Polly выполняет разбиение циклов на блоки размерности 32-а помимо описанной ранее оптимизации.

Внешняя библиотека LLVM Core использована в качестве бэкенда, выполняющего оптимизацию промежуточного кода программы и генерацию ассемблерного кода для целевой аппаратной платформы (рис. 3.1). Библиотека LLVM Core позволяет сгенерировать код для таких архитектур как, например, x86-64, x86, ARM, SPARC, MIPS, Xcore, PowerPC, NVPTX, Qualcomm Hexagon, SystemZ. Для LLVM Core созданы различные фронтенды, заменяющие Clang для компиляции программ, написанных на языках

C, C++, Java, Objective-C, C#, Python, JavaScript, Ruby, Rust, Haskell, Fortran, Ада, D, GLSL и Kotlin.

В качестве фронтенда, выполняющего лексический, синтаксический и семантический анализ, использован фронтенд Clang. Фронтенд Clang поддерживает исходный код программ ны языках C, C++, Objective-C, Objective-C++ и OpenCL C. Clang используется совместно с библиотекой LLVM Core, генерируя оптимизируемый в дальнейшем промежуточный код.

Одним из альтернативных подходов к реализации архитектуры ПС АОТО может быть использование коллекции компиляторов GCC и фреймворка для полиэдральной компиляции Graphite.

Компиляция и запуск. Разработанная программная система является приложением командной строки. Первая версия ПС АОТО, поддерживающая автоматическую оптимизацию времени выполнения МММ, обобщенного на замкнутые полукольца с элементами из множества вещественных чисел, доступна по адресу http://github.com/llvm/llvm-project.git. После загрузки нужно скомпилировать ПС АОТО:
mkdir llvm_build && cd llvm_build cmake -DLLVM_ENABLE_PROJECTS= 'polly;clang' ../llvm_git/llvm && make.

Для компиляции программ, используя ПС АОТО, нужно запустить скомпилированный фронтенд Clang с нужными опциями и значениями параметров целевой аппаратной платформы. ПС АОТО принимает на вход следующие значения параметров ГП для многоядерного процессора общего назначения: W_{L_i}, S_{L_i}, N_{VMMA} и L_{VMMA}. Остальные параметры ГП за исключением L_{prefetch} и N_{prefetch} выводятся ПС АОТО автоматически, используя в том числе, информацию, доступную в Clang о целевой аппаратной платформе. По умолчанию применяются параметры процессора Intel Core-i7-3820.

Рассмотрим пример аргументов командой строки со значениями параметров N_{VMMA}, L_{VMMA}, W_{L_1}, W_{L_2}, S_{L_1}, S_{L_2}.
llvm_build/bin/clang -mllvm -polly -target-throughput-vector-fma=N_{VMMA}
-mllvm -polly-target-latency-vector-fma=L_{VMMA} -mllvm
-polly-target-1st-cache-level-associativity=W_{L_1} -mllvm
-polly-target-2nd-cache-level-associativity=W_{L_2} -mllvm
-polly-target-1st-cache-level-size=S_{L_1} -mllvm
-polly-target-2nd-cache-level-size=S_{L_2} -ffp-contract=fast -ffast-math.

Полная версия ПС АОТО, поддерживающая автоматическую оптимизацию времени выполнения тензорных сверток, доступна по ссылке http://bitbucket.org/gareevroman/polly-groman-fork.git. Поддержка оптимизации времени выполнения MVM, обобщенного на замкнутые полукольца с элементами из множества вещественных чисел, не является целью данной работы. Ее реализация планируется в будущих версиях ПС АОТО.

3.3. Автоматическое распараллеливание программ на многоядерных процессорах общего назначения с общей памятью

Подход, используемый в Polly. Инфраструктура Polly автоматически проверяет все сгенерированные с ее помощью циклы и создает вызовы процедур библиотеки libiomp для самого внешнего цикла, распараллеливание которого не нарушает зависимости по данным [165].

В случае MMM циклы с индуктивными переменными i, j, p алгоритма 6 выполняют копирование элементов матриц во временные массивы A_c и B_c, создавая зависимости по данным, препятствующие распараллеливанию описываемых циклов (см. раздел 2.2 и раздел 2.4). Вследствие этого

цикл с индуктивной переменной j_c алгоритма 6 является внешним циклом, распараллеливаемым Polly.

Согласно работе [166], посвященной высокопроизводительным многопоточным реализациям МММ, рассматриваемый цикл с индуктивной переменной j_c алгоритма 6 является хорошим кандидатом для распараллеливания, так как в большинстве случаев отношение значения параметра N_c к значению параметра N_r высоко, и распараллеливание данного цикла позволяет снизить издержки копирования элементов во временные массивы.

Если L_2 совместно не используется потоками, созданными распараллеливанием цикла с индуктивной переменной j_c, то во время выполнения векторных операций произойдет копирование элементов временного массива A_c в L_2 каждого используемого потока. Альтернативным подходом может быть распараллеливание цикла с индуктивной переменной i. Вследствие этого каждый поток будет работать со своей располагаемой в L_2 подматрицей матрицы A. Для того чтобы избежать гонки потоков после распараллеливания цикла с индуктивной переменной i, для каждого созданного потока должна быть использована своя копия временного массива A_c. В случае TC-подобного ядра применимы такие же рассуждения, так как для оптимизации его времени выполнения в ПС АОТО используется сведение к МММ (см. раздел 2.4).

Формула для автоматического выбора цикла для распараллеливания. Автором предлагается функция следующего вида

$$F(L) = N_{\mathrm{O}}/(C_{\mathrm{PAR}} + C_{\mathrm{W}}C_{\mathrm{R}}), \tag{3.1}$$

где L — оцениваемый цикл; N_{O} — количество операций с плавающей запятой, вычисляемых циклом L; C_{PAR} — количество тактов процессора, требуемых для создания группы из N_{THREADS} потоков; C_{W} — количество секунд, требуемых для выполнения цикла L после распараллеливания; C_{R} — тактовая частота процессора.

Сформулируем и докажем следующее утверждение.

Утверждение 3.1. Если цикл L содержится в группе полностью вложенных циклов, то значение многопоточной производительности программы (ГФлоп/сек) может быть вычислено как $F(L)C_{\mathrm{R}}$.

Доказательство. Пусть оцениваемый цикл L имеет глубину d, а S_i — шаг цикла с глубиной i. Количество операций с плавающей запятой, вычисляемых группой полностью вложенных циклов, в которой содержится цикл L, может быть вычислено как $N = N_O S_{d-1} \dots S_1$ по определению N_O. Для распараллеливания цикла L требуется C_{PAR}/C_R секунд. Следовательно, $T = (C_{\mathrm{PAR}}/C_R + C_W)S_{d-1} \dots S_1$ секунд требуется на выполнение группы полностью вложенных циклов после распараллеливания содержащегося в ней цикла L. Следовательно, многопоточная производительность программы, которая представлена группой полностью вложенных циклов, содержащих цикл L, может быть вычислена по определению как $N/T = F(L)C_R$. □

Если L содержится в группе не полностью вложенных циклов, $F(L)C_R$ является грубой оценкой производительности. На практике могут использоваться формулы для приближенного вычисления значений N_{O} и $C_{\mathrm{W}}C_{\mathrm{R}}$. Для вывода формул используются характеристики векторных инструкций, описываемых ГП. C_{PAR} оценивается с использованием пакета EPCC [167].

Рассмотрим приближенные оценки N_{O} и $C_{\mathrm{W}}C_{\mathrm{R}}$ для циклов алгоритма 7 и 8. Указанные далее значения C_{RACE_i}вызваны гонкой потоков и выведены на основе эмпирических наблюдений.

Можно получить следующие приближенные оценки для алгоритма 7. В случае цикла с индуктивной переменной j_r

$$N_O = 2N_b, C_W C_R = \lceil \lceil N_b/(N_{\mathrm{VMMA}} N_{\mathrm{VEC}}) \rceil / N_{\mathrm{THREADS}} \rceil .$$

В случае цикла с индуктивной переменной i_b

$$N_O{=}2N_b M_c, C_W C_R{=} \lceil M_c/N_{\mathrm{THREADS}} \rceil (\lceil N_b/(N_{\mathrm{VEC}} N_{\mathrm{VMMA}}) \rceil + C_{\mathrm{RACE}_1} + L_{\mathrm{VMMA}}),$$

где $C_{\mathrm{RACE}_1} = \sqrt{N_{\mathrm{THREADS}}} N_{\mathrm{THREADS}} L_{\mathrm{VMMA}}$.

В случае цикла с индуктивной переменной j_b

$$N_O = 2N_c M_c, C_W C_R = C_W C_R = \lceil N_c/(N_b N_{\mathrm{THREADS}}) \rceil ,$$

где $\delta = \lceil M_c \lceil N_b/\mathbb{C}_{\mathrm{L}_1} \rceil / N_{\text{prefetch}} \rceil + M_c(\lceil N_b/(N_{\text{VEC}} N_{\text{VFMA}}) \rceil + L_{\text{VLOAD}})$.

В случае цикла с индуктивной переменной i_c

$$N_O = 2N_c M, C_W C_R = \lceil M/(M_c N_{\text{THREADS}}) \rceil (\delta + M_c C_{\text{RACE}_2}) N_c/N_b,$$

где $\delta = \lceil M_c \lceil N_b/\mathbb{C}_{\mathrm{L}_1} \rceil / N_{\text{prefetch}} \rceil + M_c(\lceil N_b/(N_{\text{VEC}} N_{\text{VFMA}}) \rceil + L_{\text{VLOAD}})$, $C_{\text{RACE}_2} = \sqrt{N_{\text{THREADS}}} N_{\text{THREADS}} L_{\text{VFMA}}$.

В случае цикла с индуктивной переменной j_c

$$N_O = 2NM, C_W C_R = \lceil N/(N_c N_{\text{THREADS}}) \rceil \delta \frac{N_c M}{N_b M_c},$$

где $\delta = \lceil M_c \lceil N_b/\mathbb{C}_{\mathrm{L}_1} \rceil / N_{\text{prefetch}} \rceil + M_c(\lceil N_b/(N_{\text{VEC}} N_{\text{VFMA}}) \rceil + L_{\text{VLOAD}})$.

Чтобы добиться более эффективного распараллеливания цикла с индуктивной переменной j_c, можно применить другой подход к определению значений параметров M_c и N_c. Если $M_c = W_{\mathrm{L}_1} - 1$ и $N_c = N_{\mathrm{L}_1} \mathbb{C}_{\mathrm{L}_1}$, то на каждой итерации цикла с индуктивной переменной i_c N_c элементов вектора y будут храниться в L_1, а не L_2. Такой подход приводит к неэффективному использованию L_2 и уменьшает количество повторных использований N_r элементов вектора y, хранимых на регистрах. Однако это позволяет использовать L_1 всех участвующих в вычислении MMM ядер процессора и устраняет проблемы общего доступа к L_2 при распараллеливании.

В результате рассмотренного подхода имеем

$$C_W C_R = \lceil N/(N_c N_{\text{THREADS}}) \rceil (\delta \frac{N_c M}{N_b M_c} + L_X),$$

где L_X может быть оценено как $L_X = (\delta N_c M)/(5 N_b M_c)$ согласно результатам экспериментов.

Можно получить следующие приближенные оценки N_{O} и $C_{\mathrm{W}} C_{\mathrm{R}}$ для циклов алгоритма 8. В случае цикла с индуктивной переменной j_r

$$N_O = 2N_c M_c, C_W C_R = \left\lceil \frac{N_c}{N_r N_{\text{THREADS}}} \right\rceil (\gamma + C_{\text{RACE}_1}),$$

где $\gamma = (N_r \lceil 4(M_c + 1)/N_{\text{prefetch}} \rceil)/(4\mathbb{C}_{\mathrm{L}_1}) + \lceil (M_c N_r)/(N_{\text{VFMA}} N_{\text{VEC}}) \rceil + L_{\text{VLOAD}}$, $C_{\text{RACE}_1} = \sqrt{N_{\text{THREADS}}} N_{\text{THREADS}} L_{\text{VFMA}}$.

В случае цикла с индуктивной переменной i_c

$$N_O = (2N_c + N_r)M,$$

$$C_W C_R = \lceil M/(M_c N_{\text{THREADS}}) \rceil (\gamma N_c/N_r + M_c N_r L_{\text{ADD}}),$$

где $\gamma = (N_r \lceil 4(M_c+1)/N_{\text{prefetch}} \rceil)/(4\mathbb{C}_{\text{L}_1}) + \lceil (M_c N_r)/(N_{\text{VFMA}} N_{\text{VEC}}) \rceil + L_{\text{VLOAD}}$.

В случае цикла с индуктивной переменной j_c

$$N_O = (2N + N_r N/N_c)M,$$

$$C_W C_R = \lceil N/(N_c N_{\text{THREADS}}) \rceil (\gamma N_c/N_r + C_{\text{RACE}_2} + M_c N_r L_{\text{ADD}}) M/M_c,$$

где $\gamma = (N_r \lceil 4(M_c+1)/N_{\text{prefetch}} \rceil)/(4\mathbb{C}_{\text{L}_1}) + \lceil (M_c N_r)/(N_{\text{VFMA}} N_{\text{VEC}}) \rceil + L_{\text{VLOAD}}$, $C_{\text{RACE}_2} = \sqrt{N_{\text{THREADS}}} N_{\text{THREADS}} L_{\text{VLOAD}} M_c N_r L_{\text{ADD}}$.

Проверка корректности функции 3.1 выполнена в разделе 4.3. Поддержка автоматического распараллеливания на основе функции 3.1 не является целью данной работы. Ее реализация планируется в будущих версиях ПС АОТО.

3.4. Выводы

В третьей главе описана архитектура ПС АОТО. ПС АОТО позволяет выполнить компиляцию программы, построить полиэдральное представление, распознавать ТС-подобные ядра и автоматически оптимизировать их время выполнения. ПС АОТО не выполняет автонастройки и не требует доступа к целевой аппаратной платформе. В качестве основы используются модели и алгоритмы, описанные в главе 1. Рассмотрены возможные реализации ПС АОТО. В рамках данной работы ПС АОТО реализована с использованием библиотеки LLVM Core, применяющейся для создания широко используемых промышленных компиляторов для различных языков программирования и целевых аппаратных платформ. Предложен метод автоматического распараллеливания программ ПС АОТО на многоядерных процессорах общего назначения с общей памятью. Представленные результаты опубликованы в работах [159, 168].

Глава 4

Оценка эффективности разработанных алгоритмов

Данная глава посвящена оценки эффективности ПС АОТО для оптимизации ТС, МММ и MVM. Выполнено сравнение с оптимизированными библиотеками, компиляторами и фреймворками.

В качестве примеров приложения алгоритмов рассмотрена оптимизация решения трехмерной структурной обратной задачи гравиметрии о восстановлении раздела между средами по известному скачку плотности и гравитационному полю, измеренному на некоторой площади земной поверхности; оптимизация решения общей задачи о путях.

4.1. Обратная задача гравиметрии о восстановлении раздела между средами

4.1.1. Постановка и решение задачи гравиметрии

Трехмерная структурная обратная задача гравиметрии о восстановлении поверхности раздела между средами состоит в нахождении функции $\zeta = \zeta(x, y)$, описывающей поверхность раздела сред постоянной плотности σ_1 и σ_2 по гравитационному полю $\Delta g(x, y, 0)$, измеренному на некоторой площади земной поверхности, и скачку плотности на границе раздела $\Delta\sigma = \sigma_1 - \sigma_2$. Пусть $z = h$ — асимптотическая плоскость для данной поверхности раздела ζ такая, что $\lim\limits_{x,y\to\pm\infty} \zeta(x, y) = h$.

Задача описывается двумерным нелинейным интегральным уравнением первого рода [169]:

$$f\Delta\sigma \int\limits_{-\infty}^{\infty}\int\limits_{-\infty}^{\infty} \left\{ \frac{1}{\sqrt{((x-x')^2 + (y-y')^2 + \zeta^2(x', y'))}} - \right. \\ \left. - \frac{1}{\sqrt{((x-x')^2 + (y-y')^2 + h^2)}} \right\} dx'dy' = \Delta g(x, y, 0), \tag{4.1}$$

где f — гравитационная постоянная.

Предположим, что значения поля вне некоторой прямоугольной области $\Pi = \{(x, y) : a \leqslant x \leqslant b, c \leqslant y \leqslant d\}$ близки к нулю. Проведем дискретизацию области Π на сетке $N \times M : (x_i, y_j), \quad i = 1, ..., N, \quad j = 1, \ldots, M$ с шагами Δx и Δy.

После аппроксимации интегрального оператора по квадратурным формулам получим уравнение

$$f\Delta\sigma\Delta x\Delta y \times \sum_{i=1}^{N}\sum_{j=1}^{M} \left[\frac{1}{\sqrt{(x_i - x_u)^2 + (y_j - y_v)^2 + \zeta^2(x_i, y_j)}} - \right. \\ \left. - \frac{1}{\sqrt{(x_i - x_u)^2 + (y_j - y_v)^2 + h^2}} \right] = \Delta g(x_u, y_v, 0), \tag{1a}$$

$$u = 1, ..., M, \quad v = 1, ..., N.$$

Правую часть $\Delta g(x_u, y_v, 0)$ представим в виде вектора

$$F = \{F_{(v-1)M+u}\} = \{\Delta g(x_u, y_v, 0)\},$$

$$u = 1, ..., M, \quad v = 1, ..., N$$

и искомую поверхность представим в виде вектора

$$z = \{z_{(j-1)M+i}\} = \{\zeta(x_i, y_j), \ \ i = 1, ..., M, \ \ j = 1, ..., N\}.$$

Тогда систему (1a) запишем в операторном виде

$$A(z) = F. \tag{4.2}$$

Для решения системы (4.2) будем использовать метод Левенберга—Марквардта [169]

$$z^{k+1} = z^k - \gamma\left[\left(A'(z^k)\right)^{\mathsf{T}} A'(z^k) + \alpha I\right]^{-1} \times A'(z^k)^{\mathsf{T}}(A'(z^k) - F), \tag{4.3}$$

где z^k — приближенное решение на k-ой итерации, $A'(z^k)$ — производная Фреше оператора A в точке u^k, I — единичный оператор, γ — демпфирующий множитель, α — параметр регуляризации.

В качестве начального приближения используется $z^0 \equiv h$. Критерием останова является условие $\left\|A(z^k) - F\right\| / \|F\| \leqslant \varepsilon_1$ при некотором $\varepsilon_1 > 0$.

Вместо выполнения трудоемкой процедуры обращения матрицы воспользуемся следующим приемом. На каждом шаге итерационного метода Левенберга—Марквардта [169] решаем систему линейных алгебраических уравнений

$$Bz = b, \tag{4.4}$$

где $B = \left(A'(z^k)\right)^{\mathsf{T}} A'(z^k) + \alpha I,$

$$b = \left[\left(A'(z^k)\right)^{\mathsf{T}} A'(z^k) + \alpha I\right] z^k - \gamma A'(z^k)^{\mathsf{T}}(A'(z^k) - F).$$

Для решения (4.4) используется метод минимальных невязок [170]

$$z^{l+1} = z^l - \frac{\langle B(Bz^l - b), Bz^l - b\rangle}{\left\|B(Bz^l - b)\right\|^2}(Bz^l - b), \tag{4.5}$$

где z^l — приближенное решение на l-ой итерации метода минимальных невязок.

В качестве начального приближения используется $z^0 \equiv 0$. Критерием останова является условие $\|Bz^l - b\|/\|b\| \leqslant \varepsilon_2$ при некотором $\varepsilon_2 > 0$.

Замечание 1. Метод Левенберга—Марквардта (4.3) и метод минимальных невязок (4.5) на каждом шаге итерационного процесса (4.3) требуют выполнения значительного количества операций MMM и MVM для нетранспонированных матриц размерности $NM \times NM$. Поэтому предложенный алгоритм автоматической оптимизации времени выполнения TC-подобных ядер может быть использован для автоматизированной и эффективной оптимизации соответствующей программы.

Замечание 2. Существуют множество методов решения СЛАУ. В большинстве случаев библиотеки реализуют ограниченный набор методов для анализа и решения каждого типа СЛАУ. Если требуется сравнить различные методы решения одного типа СЛАУ, соответствующего рассматриваемой задаче, методы реализуются вручную.

4.1.2. Результаты экспериментов

Определим ускорение $S = T_{serial}/T_{parallel}$, где T_{serial} – время выполнения последовательного кода программы с применением стандартных оптимизаций компилятора третьего уровня (O3), $T_{parallel}$ – время выполнения параллельного кода программы с использованием библиотечных процедур, либо кода, оптимизированного с помощью предлагаемого алгоритма.

Для выполнения экспериментов использовались два 18-ядерных Intel Xeon E5-2697 v4 суперкомпьютера «Уран». Характеристика одного процессора Intel Xeon E5-2697 v4: частота 2.3 ГГц, $S_{\mathrm{L}_1} = 32$ Кбайт, $S_{\mathrm{L}_2} = 256$ Кбайт, $S_{\mathrm{L}_3} = 30$ Мбайт, $W_{\mathrm{L}_{1,2}} = 8$, $W_{\mathrm{L}_3} = 20$.

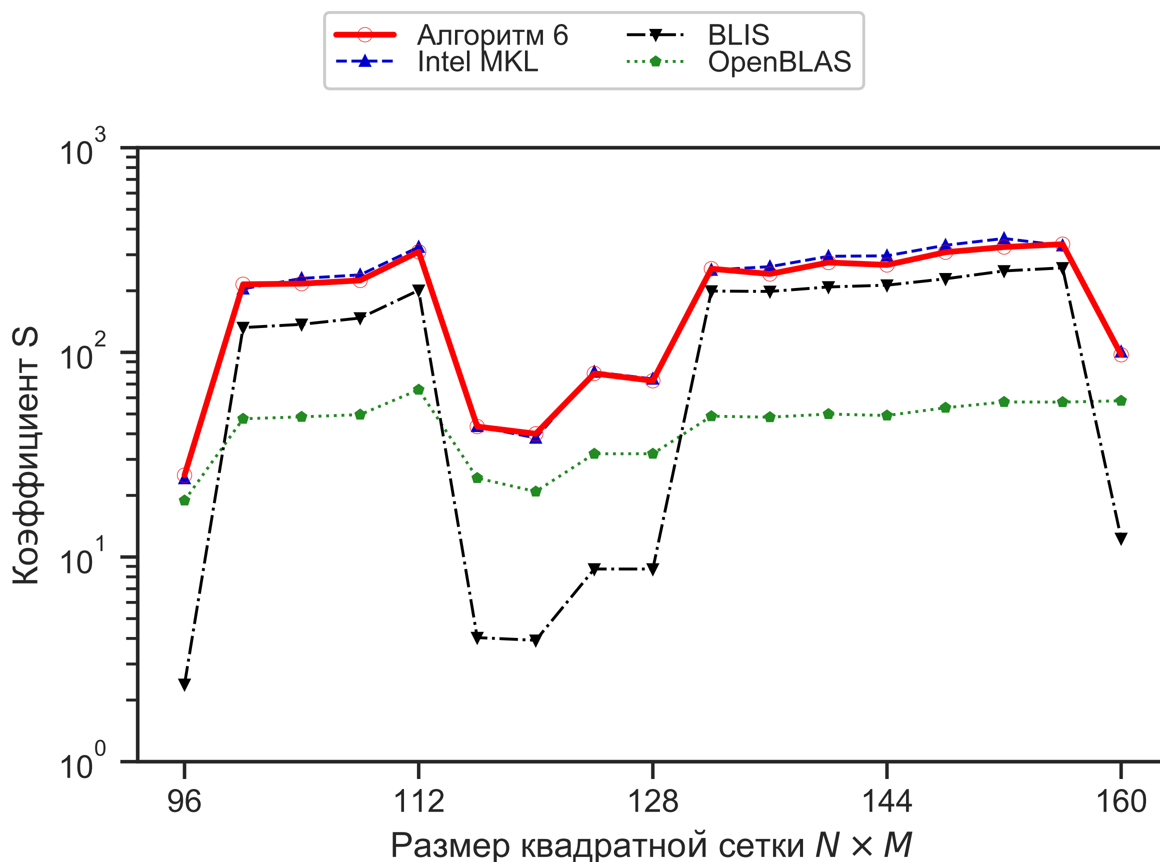


Рис. 4.1. Ускорение, полученное для обратной задачи гравиметрии

На рис. 4.1 изображены коэффициенты ускорения для решения обратной задачи гравиметрии методом Левенберга—Марквардта с использованием метода минимальных невязок на каждом шаге итерационного процесса для 36 потоков на квадратных сетках с размерами, изменяющимися от 96 до 160 с шагом 4. Табл. 4.1 содержит соответствующее время решения в минутах для обратной задачи гравиметрии на сетках размерности 96×96, 128×128 и 160×160.

Представленные коэффициенты ускорения получены для решения задачи гравиметрии на основе алгоритма 8, а также для реализаций решения задачи на основе кода библиотек Intel MKL, OpenBLAS, и BLIS. Все вычисления MVM, выполняемые описанным ранее решением задачи гравиметрии в уравнениях (4.4) и (4.5), были сделаны с использованием рассматриваемого оптимизированного кода MVM для случая нетранспонированной матрицы. Входные данные имели тип double. Использовались плотные мат-

рицы. Условием останова было выбрано $\left\|A(z^k) - F\right\| / \|F\| \leqslant \varepsilon_1 = 10^{-3}$ и $\left\|Bz^l - b\right\| / \|b\| \leqslant \varepsilon_2 = 10^{-5}$.

Таблица 4.1. Время решения в минутах для обратной задачи гравиметрии на сетках размерности $M \times N$.

Реализация \ M = N	96	128	160
Алгоритм 6	0.573	1.222	3.62
Intel MKL	0.59	1.196	3.5
OpenBLAS	0.762	2.777	6.068
BLIS	6.029	10.159	28.576
Без оптимизации	14.39	88.76	351.992

Эксперименты показали, что реализация решения задачи гравиметрии на основе алгоритма 8 сравнима с реализациями на основе кода библиотек Intel MKL, OpenBLAS и BLIS для всех рассмотренных размеров сеток. Отметим, что на разных сетках разное число итераций, поэтому время счета различно. В среднем результаты алгоритма отличаются не более чем на 1% и превосходят библиотеки OpenBLAS и BLIS.

4.2. Оценка алгоритма вычисления обобщенного матричного произведения

Выполним сравнение однопоточной и многопоточной производительностей ПС АОТО с библиотеками, содержащими оптимизированную реализацию МММ (Intel MKL, ARMPL, OpenBLAS, BLIS), и современными компиляторами (GCC, IBM XLC, ICC) и фронтендом для компиляции Clang.

Для выполнения экспериментов использовались процессоры IBM Power 8, Intel Xeon Phi, Intel Sandy Bridge, APM883208-X1 и Intel Kaby Lake (см. табл. 4.2), имеющие различную архитектуру (ppc64le, x86_64 и aarch64). Табл. 4.3 содержит информацию о версии и дополнительных опциях используемого программного обеспечения. В случае компилятора ICC использовалась опция -qno-opt-matmul, чтобы предотвратить распознавание и за-

мену MMM на вызов реализации MMM, доступной в библиотеке Intel MKL (см. табл. 4.3).

Таблица 4.2. Особенности целевых аппаратных платформ.

Название	ЦПУ	Частота (ГГц)	ОЗУ (Гбайт)	N_{VEC} для double	L_{VFMA}	N_{VFMA}	S_{L_1} (Кбайт)	W_{L_1}	S_{L_2} (Кбайт)	W_{L_2}
Intel Sandy Bridge	Intel Core i7-3820	3.6(3.8)	16	4	8	1	32	8	256	8
Intel Kaby Lake	Intel Core i7-7700	3.6(4.2)	32	4	4(6)	2	32	8	256	8
Intel Xeon Phi	Intel Xeon Phi 7210	1.3	110	8	6(7)	2	32	8	1024	16
IBM Power 8	POWER8NVL	4.023	256	2	5.5(12)	2	64	8	512	8
ARM	APM883208-X1	2.4	32	2	9(36)	0.5	32	8	256	8
Intel Xeon E5	Intel Xeon E5-2630 v4	2.2(3.1)	64	4	5(6)	2	32	8	256	8

Таблица 4.3. Версии и дополнительные опции используемого программного обеспечения.

Название	Версия	Дополнительные опции
polly	6.0.0	-O3 -march=native -mllvm -polly
AOTO	6.0.0	-O3 -march=native -mllvm -polly -polly -target-throughput-vector-fma=N_{VMMA} -mllvm -polly-target-latency-vector-fma=L_{VMMA} -mllvm -polly-target-1st-cache-level-associativity=W_{L_1} -mllvm -polly-target-2nd-cache-level-associativity=W_{L_2} -mllvm -polly-target-1st-cache-level-size=S_{L_1} -mllvm -polly-target-2nd-cache-level-size=S_{L_2} -ffp-contract=fast -ffast-math
clang	6.0.0	-O3 -march=native
gcc	4.9.2	-O3 -march=native
icc	17.0.2	-O3 -march=native
IBM XLC	13.1.5	-O3 -qarch=auto -qtune=auto
Intel MKL	11.3.3	
BLIS	0.2.2	
OpenBLAS	0.2.19	
ARMPL	2.4.0	

Поддержка технологии Intel Turbo Boost была включена на время выполнения экспериментов. Верхние теоретические границы производительности, представленные на графиках, учитывают тактовую частоту процессора с использованием технологии Intel Turbo Boost. Значения тактовой частоты процессора с использованием технологии Intel Turbo Boost показаны в скобках.

Значения L_{VFMA}, указанные в скобках, больше реальных значений L_{VFMA}. Указанные значения были определены эмпирически так, чтобы оцениваемая реализация ПС АОТО эффективно использовала как можно больше векторных регистров процессора. Как показано в разделе 4.2.1, причиной является неэффективная автоматическая векторизация, применяемая рассматриваемой реализацией ПС АОТО.

Результаты, приведенные в данной главе, являются средними арифметическими значениями, полученных для проводимых экспериментов. Каждый эксперимент проводился повторно, пока доверительный интервал с уровнем доверия 95% не был в 10% от сообщаемого среднего арифметического значения.

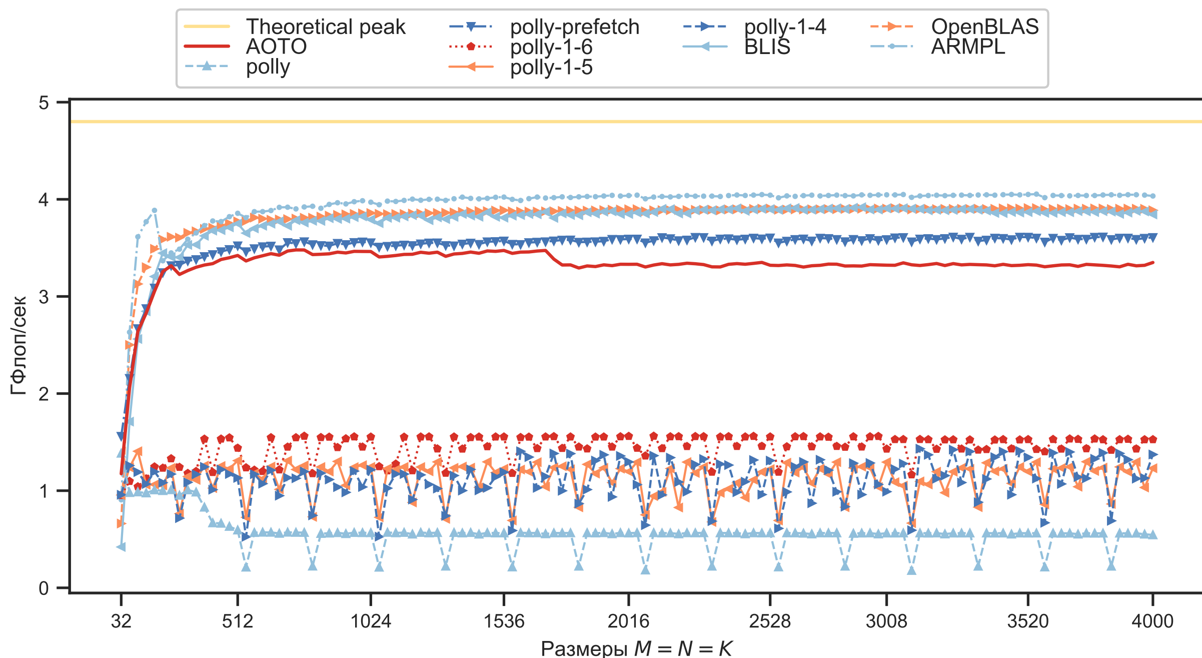


Рис. 4.2. MMA[×, +] для ARM.

4.2.1. Оценка погрешности векторизации для MMA[×, +]

Чтобы оценить погрешность векторизации библиотеки LLVM Core, применим алгоритм 9 к SCoP, содержащему утверждение S и являющемуся полиэдральным представлением реализации MMM, описанной в разделе 2.4.

Таблица 4.4. Время вычисления MMM в секундах для плотных квадратных матриц и типа double на ARM.

$N = M = K$ / Реализация	1024	2016	3008	4000
AOTO	0.63	4.92	16.39	38.22
OpenBLAS	0.56	4.2	13.93	32.93
BLIS	0.57	4.24	13.94	33.29
ARMPL	0.54	4.05	13.45	31.73
polly-prefetch	0.61	4.56	15.11	35.48
polly-1-6	1.72	11.39	35.62	83.83
polly-1-5	2.96	12.66	50.04	103.98
polly-1-4	4.076	15.43	44.53	93.23
polly	10.17	28.94	96.73	235.12

Рис. 4.2 представляет график однопоточной производительности для ARM (см. табл. 4.2), шагов алгоритма 9 и операции MMA[$\times$, $+$] вида $C \leftarrow C + A \times B$, реализованную в наборе тестов Polybench 3.2 [171]. Табл. 4.4 содержит соответствующее время вычисления в секундах. Рассматривались плотные квадратные матрицы, содержащие элементы типа double. Polybench 3.2 является набором прикладных программ, часто применяющихся в различных научных дисциплинах и требующих оптимизации.

Для простоты будет заменять левые части равенств аффинных функций их правыми частями, если другие ограничения в отношениях Пресбургера отсутствуют. Например, $\{S(i, j, p) \rightarrow (t_1, t_2, t_3) \mid t_1 = i \wedge t_2 = j \wedge t_3 = p\}$ обозначим как $\{S(i, j, p) \rightarrow (i, j, p)\}$.

```
for (j = 0; j <= N/N_c; j += 1)
  for (p = 0; p <= K/K_c; p += 1)
    for (i = 0; i <= M/M_c; i += 1)
      for (j_c = 0; j_c <= N_c/N_r; j_c += 1)
        for (i_c = 0; i_c <= M_c/M_r; i_c += 1)
          for (p_c = 0; p_c <= K_c; p_c += 1)
            for (j_r = 0; j_r <= N_r; j_r += 1)
              for (i_r = 0; i_r <= M_r; i_r += 1)
                S(M_c * i + M_r * i_c + i_r,
                  N_c * j + N_r * j_c + j_r,
                  K_c * p + p_c);
```

Рис. 4.3. SCoP после применения первых трех шагов алгоритма 9.

```
for (j = 0; j <= 31; j += 1)
  for (p = 0; p <= 31; p += 1) {
    for (tmp1 = 32 * j; tmp1 <= 32 * j + 31; tmp1 += 1)
      for (tmp2 = 32 * p; tmp2 <= 32 * p + 31; tmp2 += 1)
        Copy0(0, tmp1, tmp2);
    for (i = 0; i <= 31; i += 1) {
      for (tmp1 = 32 * i; tmp1 <= 32 * i + 31; tmp1 += 1)
        for (tmp3 = 32 * p; tmp3 <= 32 * p + 31; tmp3 += 1)
          Copy1(tmp1, 0, tmp3);
      for (j_c = 0; j_c <= 15; j_c += 1)
        for (i_c = 0; i_c <= 15; i_c += 1)
          for (p_c = 0; p_c <= 31; p_c += 1) {
            S(32 * i + 2 * i_c, 32 * j + 2 * j_c,
              32 * p + p_c);
            S(32 * i + 2 * i_c + 1, 32 * j + 2 * j_c,
              32 * p + p_c);
            S(32 * i + 2 * i_c, 32 * j + 2 * j_c + 1,
              32 * p + p_c);
            S(32 * i + 2 * i_c + 1, 32 * j + 2 * j_c + 1,
              32 * p + p_c);
          }
    }
  }
```

Рис. 4.4. SCoP после применения первых шести шагов алгоритма 9.

После применения первых трех шагов алгоритма 9 в результате перестановок циклов и их разбиения на блоки аффинный план утверждения S будет преобразован из $\{S(i,j,p) \to (i,j,p)\}$ в $\{S(i,j,p) \to \lfloor \frac{j}{N_c} \rfloor, \lfloor \frac{p}{K_c} \rfloor, \lfloor \frac{i}{M_c} \rfloor,$ $j \bmod N_c, i \bmod M_c, p \bmod K_c\}$ (см. рис. 4.3). После разбиения циклов на блоки, выполняемого четвертым шагом алгоритма 9, аффинный план утверждения S будет иметь вид $\{S(i,j,p) \to \lfloor \frac{j}{N_c} \rfloor, \lfloor \frac{p}{K_c} \rfloor, \lfloor \frac{i}{M_c} \rfloor, \lfloor \frac{j \bmod N_c}{N_r} \rfloor \lfloor \frac{i \bmod M_c}{M_r} \rfloor,$

$p \bmod K_c, j \bmod N_r, i \bmod M_r\}$. Преобразованный SCoP представлен на листинге 4.3. Для простоты $M \bmod M_c = N \bmod N_c = K \bmod K_c = 0$. Первые четыре шага алгоритма 9 позволяют достичь 23% верхней теоретической границы на производительность (см. polly-1-4 на рис. 4.2).

Размотка циклов, выполняемая пятым шагом алгоритма 9, не влияет на производительность (см. polly-1-5 на рис. 4.2). Копирование элементов матриц во временные массивы на шестом шаге алгоритма 9 позволяет достичь 26% верхней теоретической границы на производительность (см. polly-1-6 на рис. 4.2). Преобразованный SCoP представлен на рис. 4.4.

Векторизация, выполняемая LLVM Core, позволяет достичь 73% верхней теоретической границы на производительность (см. AOTO на рис. 4.2). Для дальнейшего увеличения производительности можно воспользоваться автоматической предвыборкой данных, доступной в LLVM Core для некоторых платформ и отключенной по умолчанию. Ее использование позволяет достичь 75% верхней теоретической границы на производительность (см. polly-prefetch на рис. 4.2). Однако этого недостаточно для достижения производительности оптимизированных библиотек.

Согласно работе [172], для создания высокопроизводительных реализаций MMA[×, +] необходимо эффективное использование векторных регистров и эффективный порядок выполнения команд. Применим руководство, представленное в работе [172], чтобы вручную оптимизировать времени выполнения реализации MMA[×, +] и определить то, что мешает достижению производительности рассматриваемых оптимизированных библиотек.

Рис. 4.5 представляет график однопоточной производительности для Sandy Bridge (см. табл. 4.2) и операции MMA[×, +]. Рассматривались плотные квадратные матрицы, содержащие элементы типа double. Графики представляют собой наилучшую производительность, полученную с исполь-

зованием компилятора GCC и фронтенда CLang. Табл. 4.5 содержит соответствующее время вычисления в секундах.

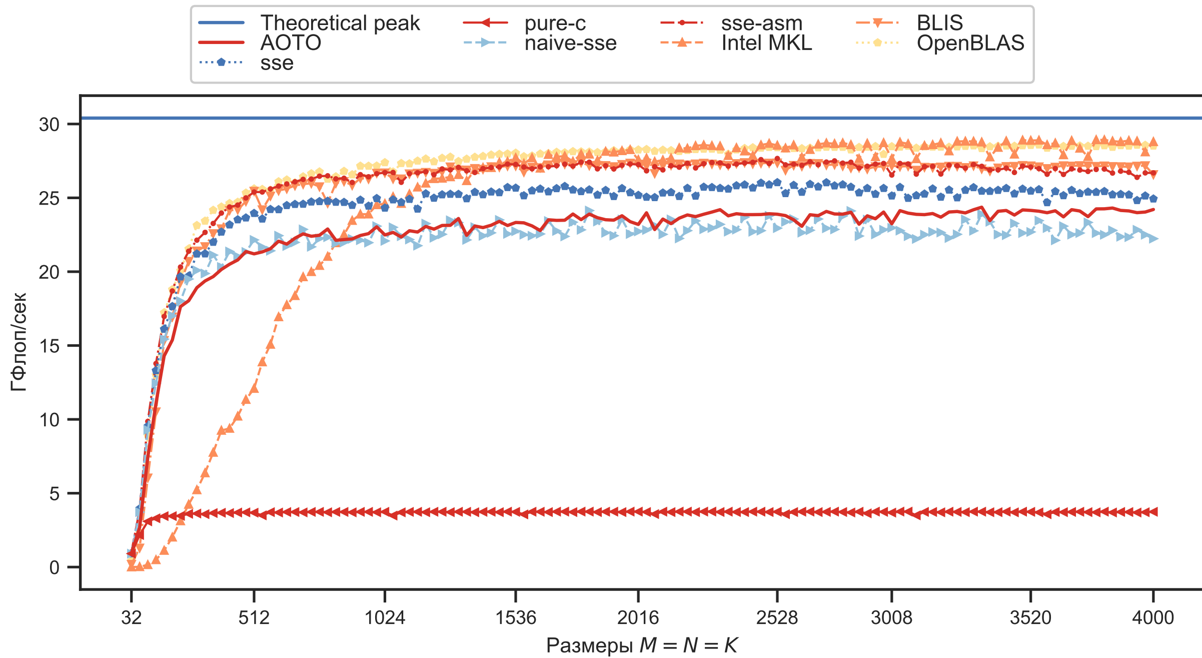


Рис. 4.5. MMA[×, +] для Sandy Bridge.

Таблица 4.5. Время вычисления MMM в секундах для плотных квадратных матриц и типа double на Sandy Bridge.

Реализация \ $N = M = K$	1024	2016	3008	4000
AOTO	0.095	0.68	2.26	5.29
OpenBLAS	0.08	0.58	1.91	4.49
BLIS	0.082	0.6	2	4.82
Intel MKL	0.086	0.59	1.89	4.44
sse-asm	0.08	0.61	1.99	4.79
sse	0.086	0.65	2.12	5.13
naive-sse	0.093	0.72	2.39	5.76
pure-c	0.62	4.39	14.49	34.19

Линия, отмеченная pure-c, представляет оценку однопоточной производительности для утверждения S реализации MMM, представленной в разделе 2.4, после применения первых шести шагов алгоритма 9, но без размотки циклов, выполняемой на пятом шаге алгоритма 9. Реализация достигает 12.31% верхней теоретической границы на производительность.

Дальнейшая размотка циклов пятым шагом алгоритма 9 и векторизация, выполненная с использованием x86 SIMD интринсиков, позволяют до-

стичь 79.74% верхней теоретической границы на производительность (см. naive-sse на рис. 4.5). Результататы вычислений, выполняемых внутренним циклом полученной реализации, могут быть представлены как $C_c \leftarrow C_c + \left[A_c(I, p_c)\mathbb{B}_c\right]$, где I $\in i_c \dots i_c + M_r - 1$, $\mathbb{B}_c$ — подматрица матрицы B вида $B_c(p_c, j_c{:}j_c + N_r$ - 1). Для достижения большей производительности элементы матрицы A также должны храниться на векторных регистрах [172]. Однако во всех случаях векторизация, выполняемая LLVM Core, не позволяет сделать это, генерируя $C_c \leftarrow C_c + \left[A_c(I, p_c)\mathbb{B}_c\right]$.

Для дальнейшего улучшения производительности элементы матрицы A загружаются на векторный регистр. Далее вычисляются все возможные перестановки данного регистра и результаты их переменожения с $\mathbb{B}_c$ [172]. Таким образом, вычисления, выполняемые внутренним циклом полученной реализации, могут быть описаны как $C_c \leftarrow C_c + A_c(i_c : i_c + M_r - 1, p_c) \cdot B_c(p_c, j_c : j_c + N_r - 1)$. Представленная реализация достигает 83.91% верхней теоретической границы на производительность (см. sse на рис. 4.5).

Порядок выполнения ассемблерных инструкций устанавливается компилятором при преобразовании x86 SIMD интринсиков [172]. Для того чтобы порядок выполнения используемых команд был наиболее эффективен, преобразуем x86 SIMD интринсики в ассемблерные инструкции вручную [172]. Кроме этого, выполним предвыборку данных и осуществим дополнительную размотку внутреннего цикла для использования большего числа векторных регистров для хранения элементов матриц A и B [172]. Полученная реализация достигает производительности BLIS и составляет 86.92% верхней теоретической границы на производительность (см. sse-asm на рис. 4.5).

4.2.2. Результаты экспериментов для MMA[×, +]. Случай однопоточной производительности

Рассмотрим операцию MMA[×, +] вида $C \leftarrow C + A \times B$, реализованную в наборе тестов Polybench 3.2 [171].

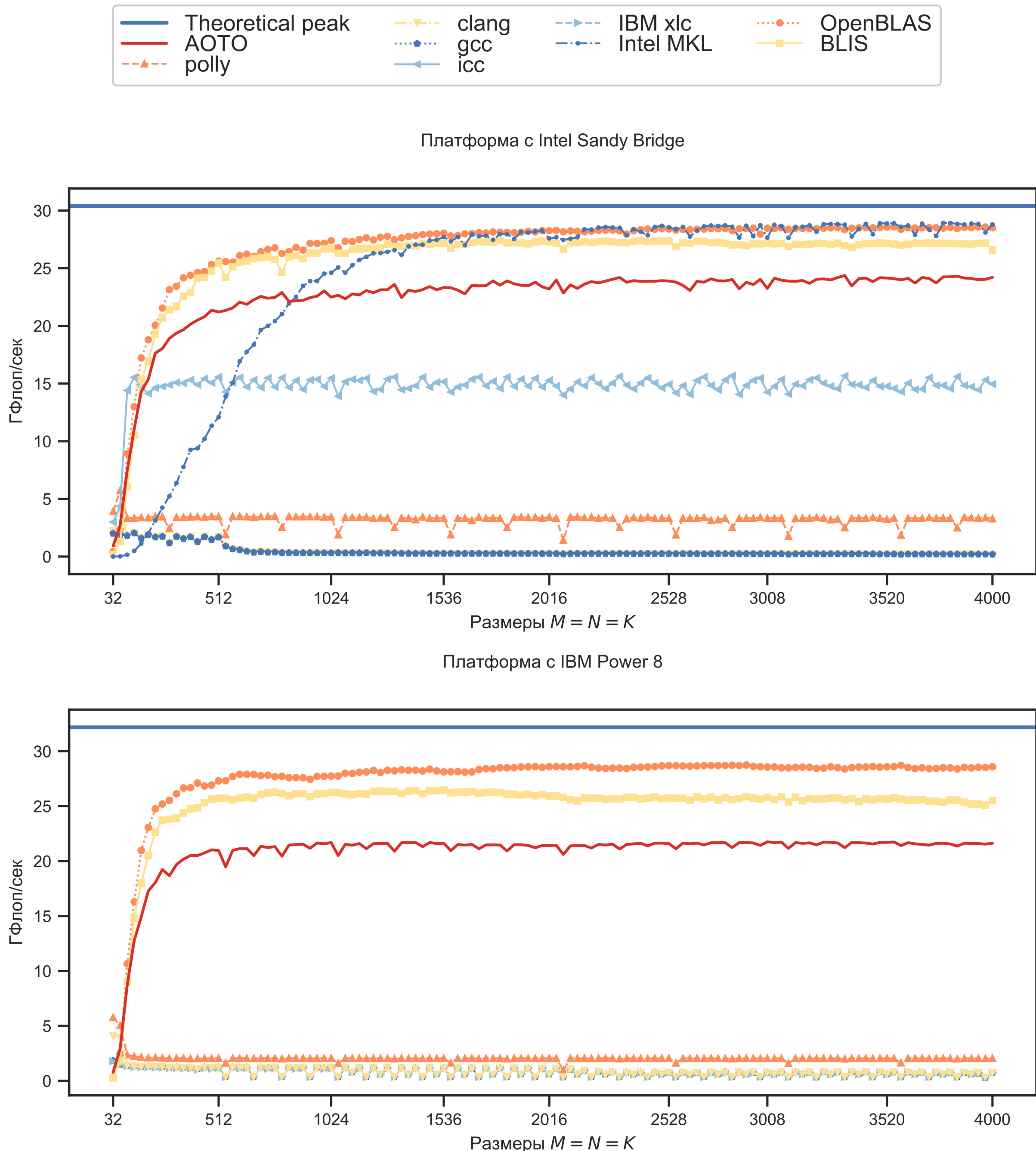


Рис. 4.6. MMA[×, +] для матриц, содержащих элементы типа double.

Рис. 4.6, рис. 4.7, рис. 4.8 и рис. 4.9 представляют результаты оценки однопоточной производительности для плотных квадратных и не квадратных матриц, содержащих элементы типа double. Размерности измерений матриц изменялись одновременно от 32 до 4000 с шагом 32. В случае

не квадратных матриц измерение K было фиксировано и равнялось 2000. Табл. 4.6 и табл. 4.7 содержат время вычисления в секундах для квадратных и не квадратных матриц, соответственно.

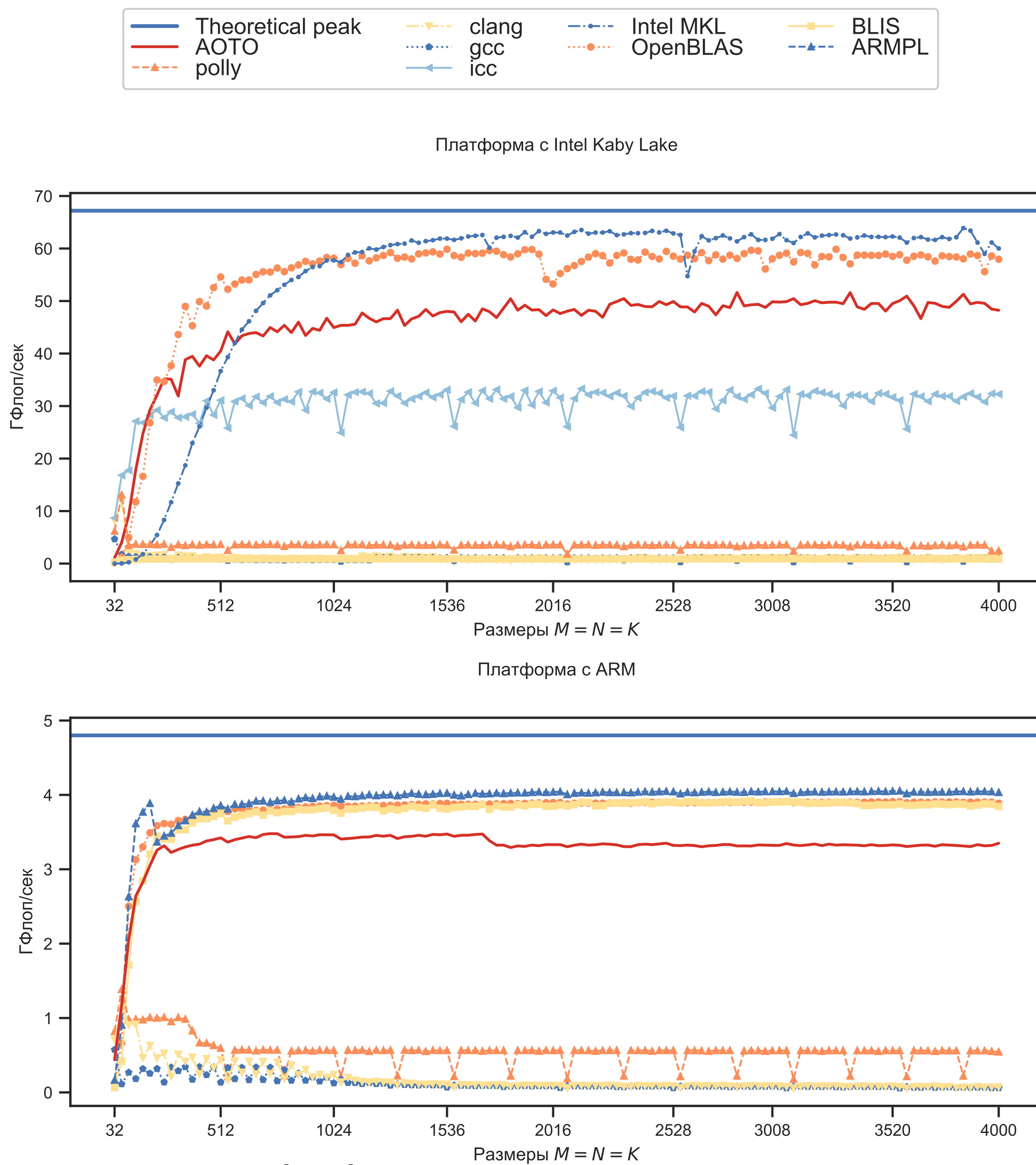
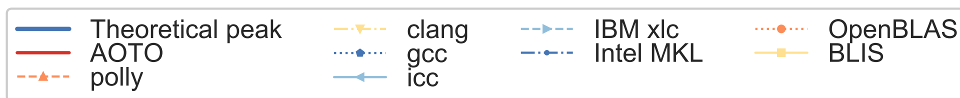
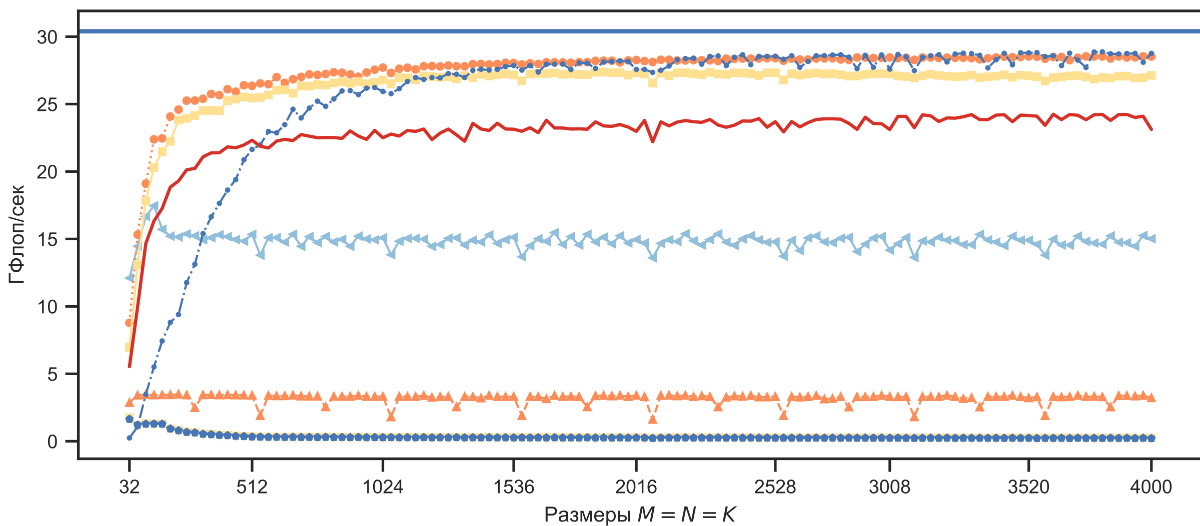


Рис. 4.7. MMA[×, +] для матриц, содержащих элементы типа double.

Таблица 4.6. Время вычисления MMM в секундах для плотных квадратных матриц и типа double на Sandy Bridge.

Реализация \ $N = M = K$	1024	2016	3008	4000
AOTO	0.095	0.68	2.26	5.29
polly	1.14	4.94	16.37	39
clang	6.85	64.33	232.38	658.04
gcc	6.53	64.42	227.52	659.24
icc	0.15	1.11	3.68	8.56
OpenBLAS	0.08	0.58	1.91	4.49
BLIS	0.082	0.6	2	4.82
Intel MKL	0.086	0.59	1.89	4.44





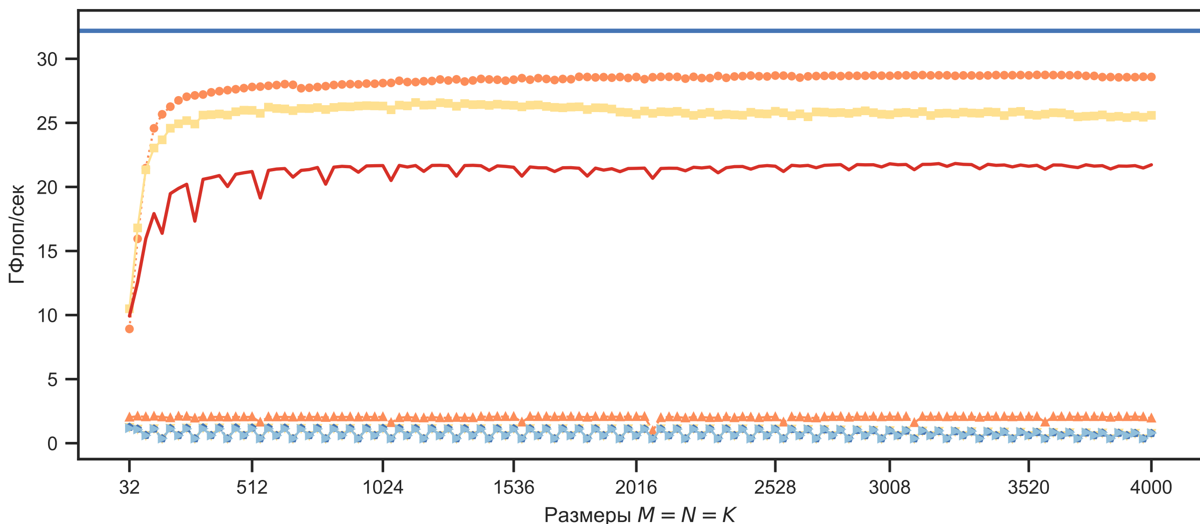


Рис. 4.8. MMA[×, +] для матриц, содержащих элементы типа double (k = 2000).

Таблица 4.7. Время вычисления MMM в секундах для плотных матриц и типа double на Sandy Bridge.

$N = M, K = 2000$ / Реализация	1024	2016	3008	4000
AOTO	0.18	0.68	1.5	2.77
polly	2.34	4.94	10.99	19.9
clang	14.89	63.8	146.96	300.58
gcc	14.79	63.91	147.23	306.52
icc	0.3	1.11	2.47	4.26
OpenBLAS	0.15	0.58	1.27	2.24
BLIS	0.16	0.59	1.34	2.36

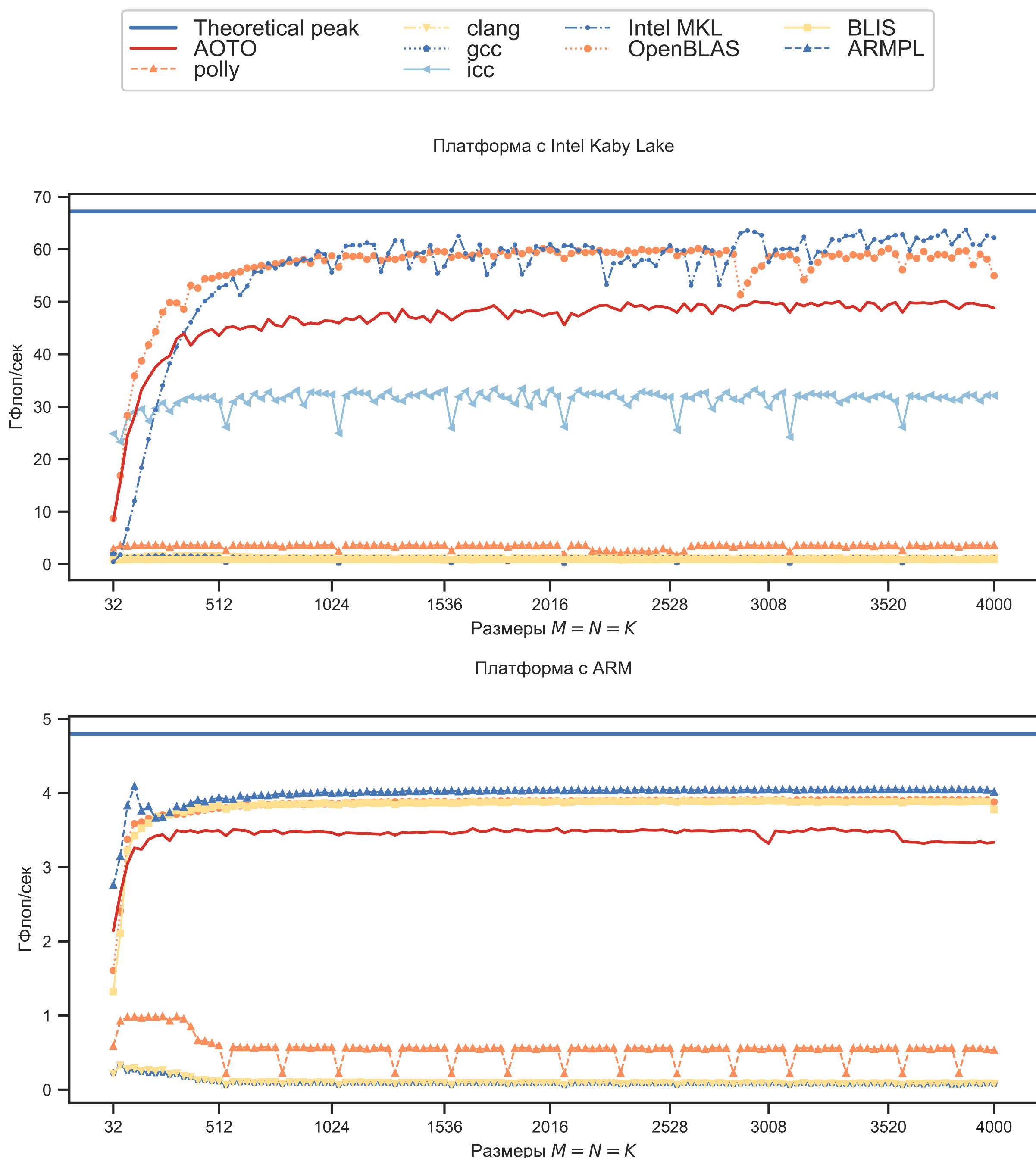


Рис. 4.9. MMA[×, +] для матриц, содержащих элементы типа double (k = 2000).

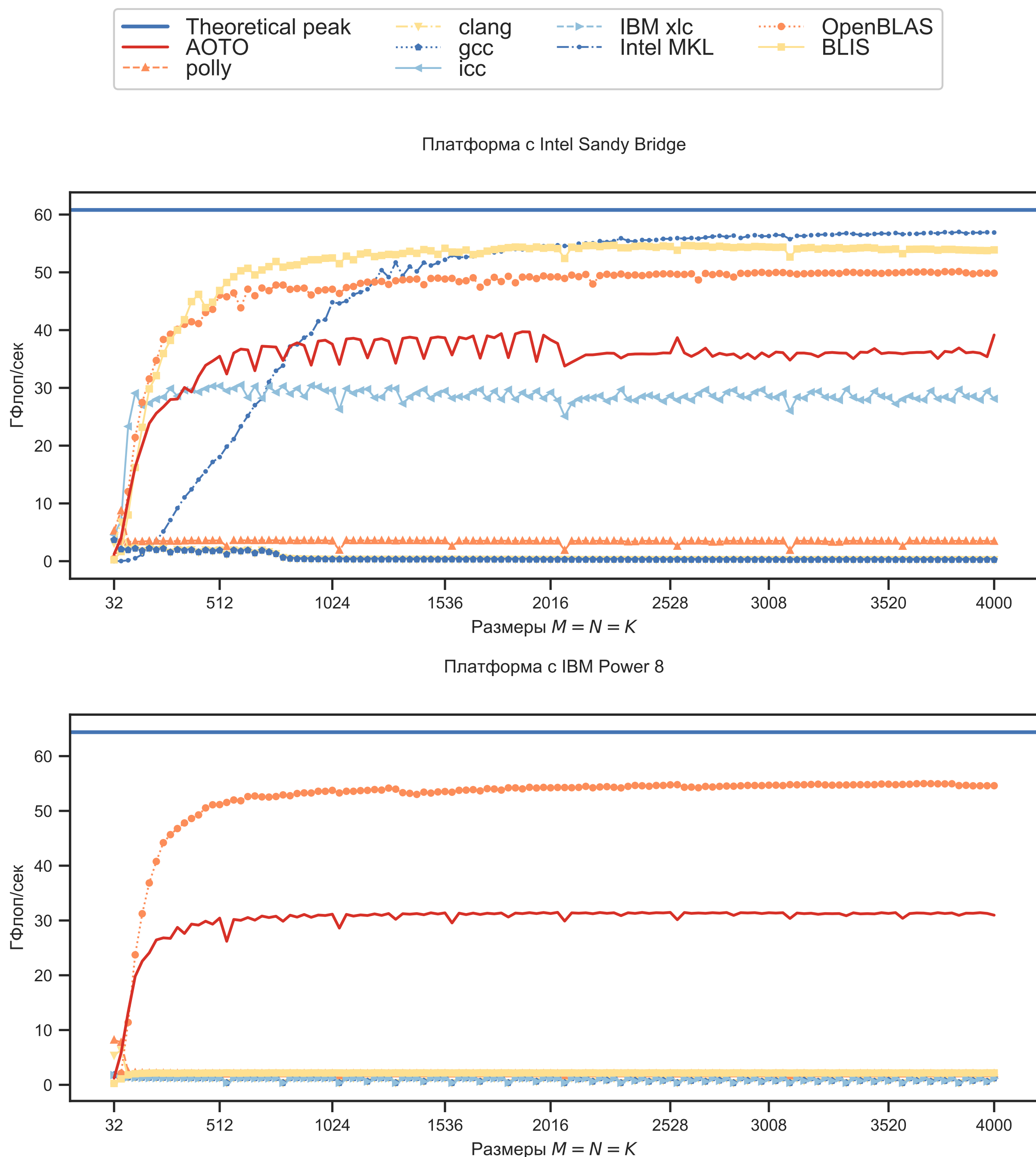


Рис. 4.10. MMA[×, +] для матриц, содержащих элементы типа float.

Таблица 4.8. Время вычисления MMM в секундах для плотных квадратных матриц и типа float на Sandy Bridge.

Реализация \ $N = M = K$	1024	2016	3008	4000
AOTO	0.06	0.43	1.5	3.27
Intel MKL	0.048	0.29	0.96	2.25
ICC	0.082	0.588	1.92	4.55
OpenBLAS	0.046	0.33	1.089	2.57
BLIS	0.042	0.3	1	2.38
polly	1.12	4.66	15.49	36.91
clang	6.45	57.08	202.34	530.95
gcc	6.43	57.03	202.43	523.059
Intel MKL	0.163	0.589	1.262	2.22

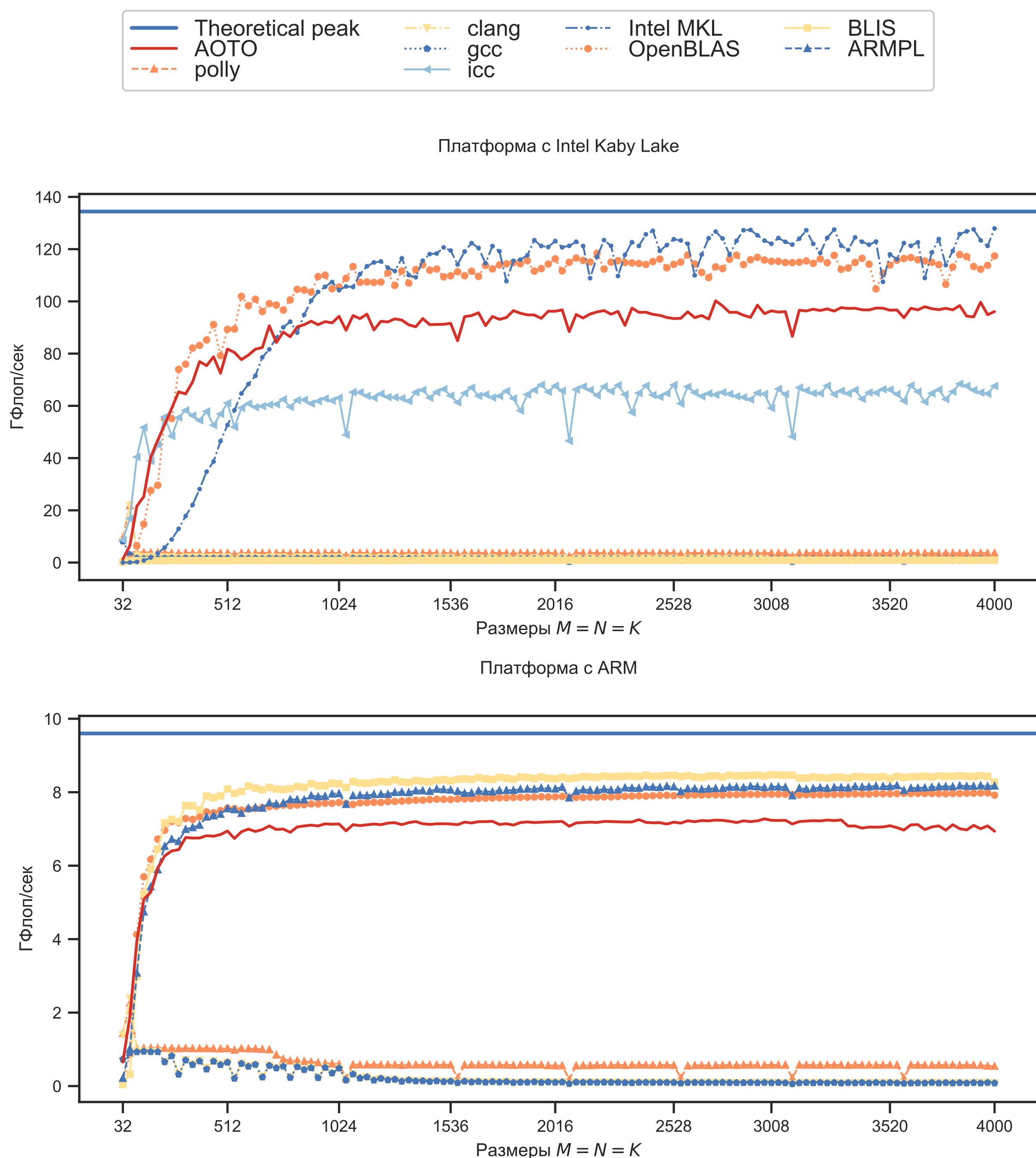


Рис. 4.11. MMA[×, +] для матриц, содержащих элементы типа float.

Рис. 4.10 и рис. 4.11 показывают результаты для плотных квадратных матриц, содержащих элементы типа float. Табл. 4.8 содержит соответствующее время вычисления в секундах. Можно сделать вывод о том, что ПС АОТО достигает 1.63-кратного ускорение по сравнению с компилятором ICC и 20-кратного ускорения по сравнению с остальными современными промышленными компиляторами. Отметим, что для матриц, размер которых меньше 80^2, оптимизация времени выполнения достигает производительность реализации MMA[×, +] доступной в библиотеке Intel MKL. Однако

если матрицы содержат меньше 32^2 элементов, целесообразно использовать другие подходы к оптимизации времени выполнения MMA[$\times$, $+$]. Это соответствует результатам, представленным в работе [52].

В случае Kaby Lake, квадратных матриц и типа double програмная система AOTO достигает 83.33% производительности OpenBLAS. В случае IBM Power 8 програмная система AOTO достигает только 75.54%. Как было показано ранее, разница объясняется неоптимальным использованием регистров. Оценим количество векторных регистров, используемых на каждой итерации цикла с индуктивной переменной p_c алгоритма 6. Как было рассмотрено ранее в данном разделе, вычисляется $M_r N_r$ элементов матрицы C и используется $M_r + N_r$ элементов матриц A и B. Так как только элементы второго операнда хранятся на векторном регистре, то используется не более чем $(M_r N_r + N_r)/N_{\text{VEC}}$ векторных регистров. В случае Kaby Lake это количество равно $(8 \cdot 6 + 8)/4 = 12$, что составляет 87.5% всех регистров, доступных на данной целевой аппаратной платформе. В случае IBM Power 8 используется только 48.43% доступных векторных регистров. В случае Sandy Bridge и ARM используется 62.5% и 65.62% доступных регистров, соответственно. Вследствие этого в случае с Sandy Bridge и ARM програмная система AOTO достигает только 80.35% и 84.61% производительности OpenBLAS на этих платформах, соответственно. Для типа float и процессоров Kaby Lake, IBM Power 8, Sandy Bridge и ARM програмная система AOTO достигает 83.33%, 54.54%, 70%, и 86.25% производительности OpenBLAS, соответственно. При этом процентное соотношение используемых регистров 87.5%, 42.18%, 56.25%, и 65.62% для целевых аппаратных платформ с Kaby Lake, IBM Power 8, Sandy Bridge и ARM, соответственно.

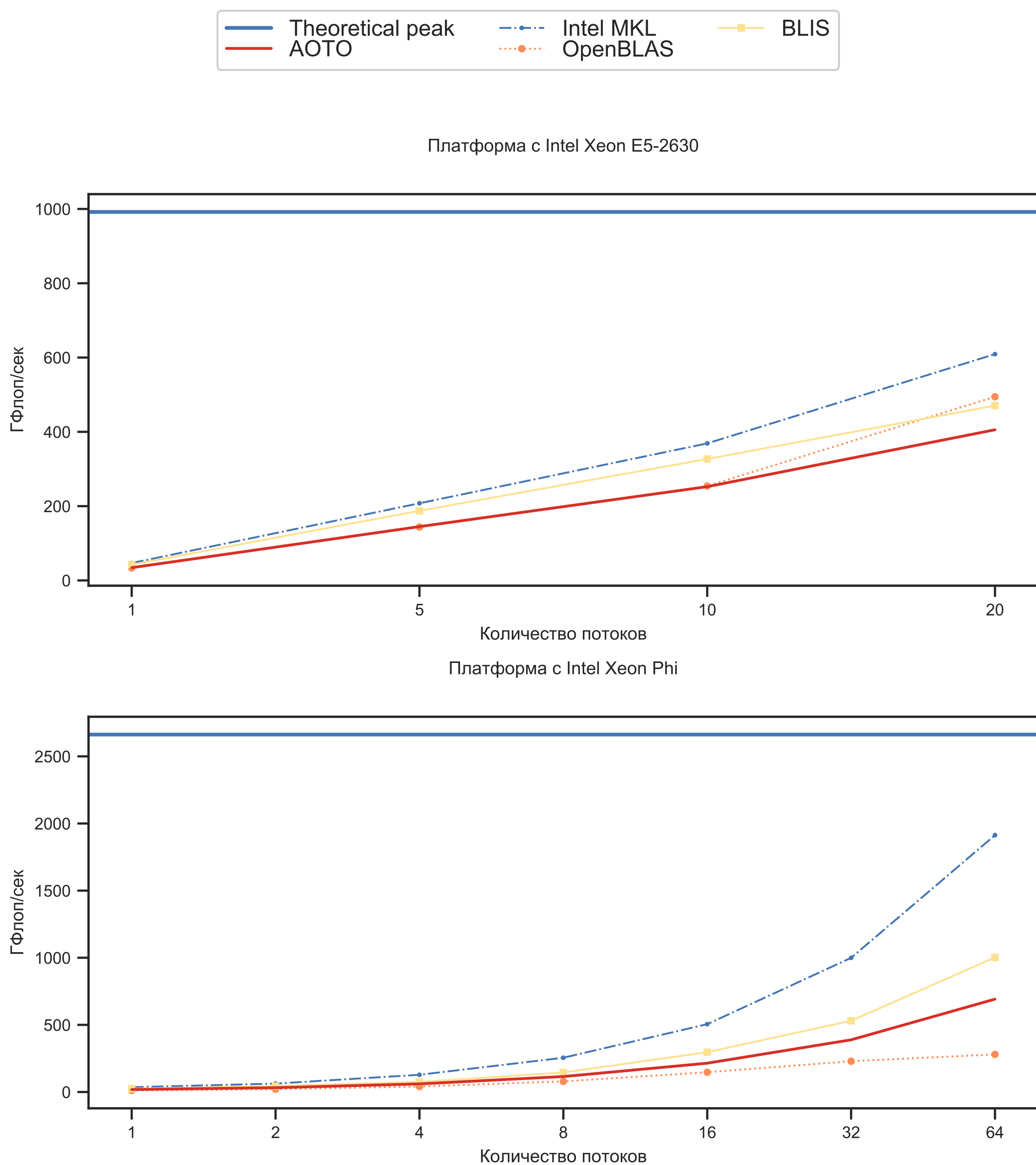


Рис. 4.12. MMA[×, +] для типа double.

Таблица 4.9. Время вычисления MMM в секундах для квадратных матриц размерности 8000 × 8000 для разного количества потоков на Intel Xeon E5-2630 v4.

Реализация \ Количество потоков	1	5	10	20
AOTO	29.83	7.07	4.06	2.73
Intel MKL	21.84	4.93	2.78	1.68
OpenBLAS	30.33	7.13	4.03	2.07
BLIS	24.06	5.46	3.13	2.18

4.2.3. Результаты экспериментов для MMA[$\times$, $+$]. Случай многопоточной производительности

Для оценки многопоточной производительности рассмотрим плотные квадратные матрицы размерности 8000×8000, содержащие элементы типа double, и различного количества потоков. Для тестирования использовались два 10-ядерных Intel Xeon E5-2630 v4 и один 64-ядерный Intel Xeon Phi, загруженный в плоском режиме (см. табл. 4.2). На рис. 4.12 представлены результаты тестирования. Табл. 4.9 содержит время вычисления в секундах для MMM, квадратных матриц размерности 8000×8000 и разного количества потоков на Intel Xeon E5-2630 v4.

В случае Intel Xeon E5-2630 v4 программная система AOTO позволяет достичь 86.18% многопоточной производительности реализаций MMA[$\times$, $+$], доступных в рассматриваемых оптимизированных библиотеках. В случае Intel Xeon Phi предложенный подход превосходит по производительности OpenBLAS, не имеющий реализации MMM для этой платформы. Вследствие этого на рассматриваемой платформе OpenBLAS применяет общую для широкой группы платформ реализацию MMM, уступающую специализированным реализациям в производительности. В случае Intel Xeon Phi программная система AOTO достигает только 69.02% производительности библиотеки BLIS, так как применяемая стратегия распараллеливает цикл с индуктивной переменной j_c алгоритма 6, являющегося самым внешним и не содержащим зависимости по данным. Каждые два ядра Intel Xeon Phi имеют отдельный разделяемый между ними L_2. В результате распараллеливания описываемого цикла происходит копирование элементов временного массива A_c в L_2 каждого из ядер (см. раздел 3.3).

4.2.4. MMA[×, +] и целочисленные типы данных. Случай однопоточной производительности

MMM матриц, содержащих элементы из множества целых чисел, имеет отдельную область применения. В частности, целочисленные 16-битные типы данных используются в глубоких нейронных сетях с целью уменьшения издержек передачи данных во время выполнения MMM за счет уменьшения точности вычислений [173]. MMM для целочисленных и логических типов данных не описываются интерфейсом BLAS. Вследствие этого реализации MMM для таких типов данных недоступны для многих целевых платформ. Рассмотрим применение ПС AOTO для оптимизации времени выполнения MMM в случае целочисленных типов данных.

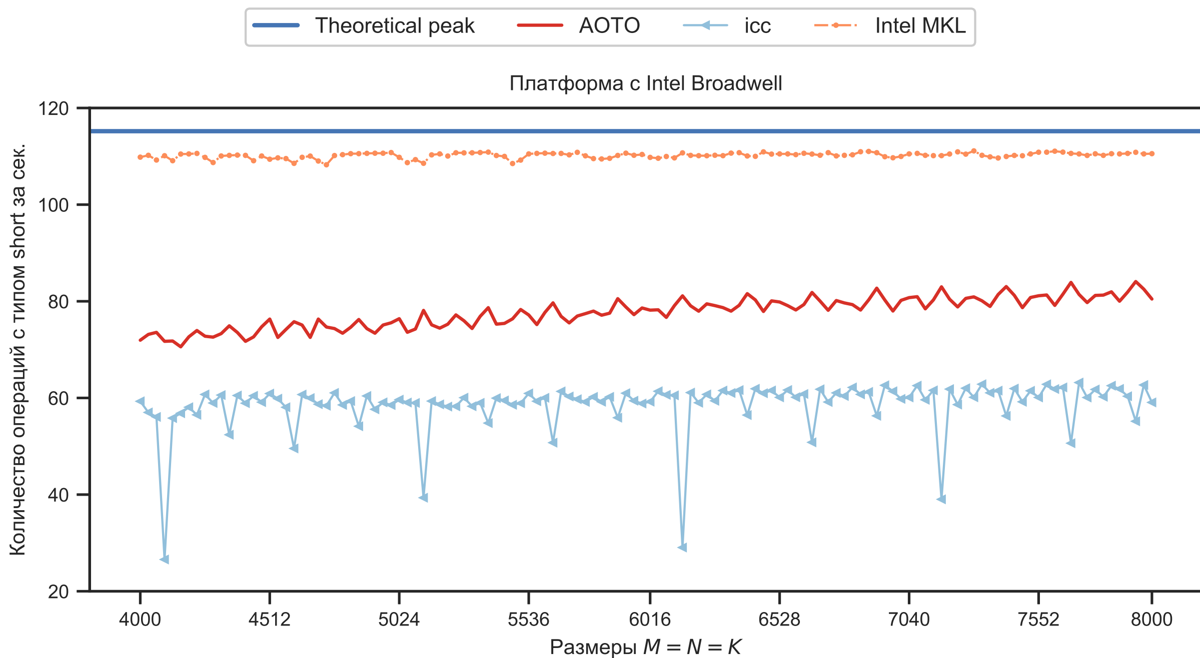


Рис. 4.13. MMA[×, +] для типа short.

Эксперимент проводился на платформе с процессором Intel Xeon E5-2697 v4 с частотой 2.3 ГГц, $S_{L_1} = 32$ Кбайт, $S_{L_2} = 256$ Кбайт, $S_{L_3} = 30$ Мбайт, $W_{L_{1,2}} = 8$, $W_{L_3} = 20$. Рис. 4.13 содержит график однопоточной производительности реализаций MMM для плотных квадратных матриц и целочисленного 16-битного типа данных. Размерности измерений матриц

изменялись одновременно от 4000 до 8000 с шагом 32. Табл. 4.10 содержит время вычисления в секундах для целочисленных 16-битных типов данных.

Таблица 4.10. Время MMM в секундах для типа short.

$N = M = K$ / Реализация	4000	6016	8000
AOTO	1.78	5.57	12.73
Intel MKL	1.17	3.97	9.26
ICC	2.16	7.35	17.33

Можно сделать вывод о том, что по сравнению с компилятором ICC программная система AOTO достигает 1.36-кратного ускорения в случаях 16-битных типов данных. Отметим, что MMM для целочисленных типов данных недоступно в библиотеках OpenBLAS и BLIS, рассмотренных в разделе 4.2.2. Вследствие этого ПС AOTO может применяться в случае процессоров от AMD, для которых используются библиотеки OpenBLAS и BLIS в силу отсутствия специализированной библиотеки, реализующей интерфейс BLAS. В случае библиотеки Intel MKL программная система AOTO достигает 72.74% однопоточной производительности реализации MMM для рассматриваемого типа данных. Как было показано в разделе 4.2.1, устранение проблем векторизации, выполняемой библиотекой LLVM Core в ПС AOTO может позволить достичь производительности оптимизированных библиотек.

4.2.5. MMA[×, +] и целочисленные типы данных. Случай многопоточной производительности

Сравним многопоточную производительность реализации MMM для 16-битных типов данных доступную в библиотеке Intel MKL с реализациями, полученными с использованием ПС AOTO, компилятора ICC. Эксперимент проводился на платформе с двумя 18-ядерными процессорами Intel Xeon E5-2697 v4 с частотой 2.3 ГГц, $S_{\mathrm{L}_1} = 32$ Кбайт, $S_{\mathrm{L}_2} = 256$ Кбайт,

$S_{L_3} = 30$ Мбайт, $W_{L_{1,2}} = 8$, $W_{L_3} = 20$. Рис. 4.14 содержит график многопоточной производительности реализаций MMM для плотных квадратных матриц и целочисленного 16-битного типа данных. Размерности измерений матриц изменялись одновременно от 4000 до 8000 с шагом 32. Использовались 36 потоков. Табл. 4.11 содержит время вычисления в секундах для целочисленных 16-битных типов данных.

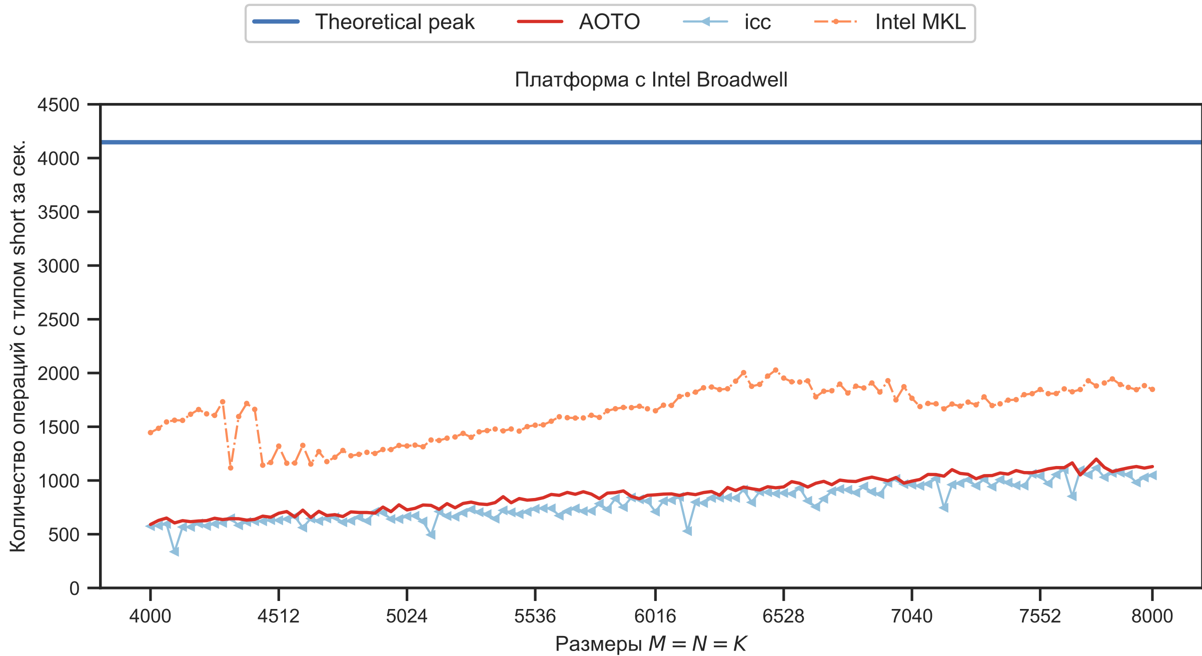


Рис. 4.14. MMA[×, +] для типа short.

Таблица 4.11. Время MMM в секундах для типа short.

Реализация \ $N = M = K$	4000	6016	8000
AOTO	0.217	0.5	0.91
Intel MKL	0.0885	0.264	0.554
ICC	0.222	0.61	0.98

Можно сделать вывод о том, что в случаях 16-битных типов данных ПС AOTO достигает 1.22-кратного ускорения по сравнению с компилятором ICC. В случае библиотеки Intel MKL программная система AOTO достигает 60.87% многопоточной производительности реализации MMM для рассматриваемого типа данных. Как было показано в разделе 4.2.1, производительность может быть улучшена за счет устранения проблем векторизации, выполняемой библиотекой LLVM Core в ПС AOTO.

4.2.6. Частные случаи общей задачи о путях. Однопоточная производительность

С середины 1970-ых годов было установлено, что определенные над различными замкнутыми полукольцами операции MMA могут использоваться для решения задач из класса APP [10]. Рассмотрим следующие задачи из класса APP: нахождение путей наименьшей и наибольшей надежности, нахождение путей наибольшей вместимости, нахождение путей наибольшей стоимости, нахождение кратчайших путей (см. раздел 2.1).

Таблица 4.12. Весь набор тестов APP для типа double.

Задача	Размеры	gcc (%)	clang (%)	icc (%)	polly (%)	AOTO (%)	Верхняя граница (ГФлоп/сек)
MMA[+, max]	128	10.99	10.86	**87.83**	22.17	75.26	15.2
	1024	2.17	2.17	**89.41**	12.3	88.42	
	2000	1.84	1.64	89.87	21.91	**90.99**	
	4000	1.32	1.32	89.67	21.97	**91.64**	
MMA[/, max]	128	42.11	42.76	81.58	42.76	**82.24**	1.52
	1024	21.71	21.05	88.16	44.74	**88.16**	
	2000	16.45	16.45	88.82	44.74	**88.82**	
	4000	13.16	12.5	89.47	45.39	**89.47**	
MMA[min, max]	128	10.07	11.05	**94.14**	21.91	76.84	15.2
	1024	2.11	2.11	88.03	11.71	**88.16**	
	2000	1.64	1.64	88.16	21.84	**92.04**	
	4000	1.32	1.25	90.92	21.51	**92.43**	
MMA[×, max]	128	5.26	5.39	47.76	11.02	**56.18**	30.4
	1024	1.09	1.05	46.94	6.15	**76.61**	
	2000	0.82	0.86	47.63	10.86	**76.97**	
	4000	0.66	0.66	48.42	10.92	**79.14**	
MMA[×, min]	128	5.33	5.43	49.54	11.15	**55.69**	30.4
	1024	1.05	1.05	47.53	5.92	**76.94**	
	2000	0.82	0.82	46.94	10.85	**77.43**	
	4000	0.66	0.63	48.59	10.76	**78.52**	
MMA[×, −]	128	5.33	5.43	48.36	11.18	**55.03**	30.4
	1024	1.09	1.05	46.64	6.09	**76.38**	
	2000	0.82	0.82	47.8	10.89	**77.73**	
	4000	0.63	0.66	47.34	11.05	**78.52**	
MMA[+, min]	128	10.99	10.99	**88.36**	22.57	76.78	15.2
	1024	2.11	2.11	**89.21**	12.04	88.42	
	2000	1.64	1.64	91.18	21.91	**91.38**	
	4000	1.25	1.25	**90.92**	21.78	90.79	

Для решения задач из класса APP использовался алгоритм последовательного возведения матрицы весов в квадрат с применением операций MMA (см. раздел 2.1). В качестве дополнения рассматриваются операции MMA[×, −] и MMA[/, max], чьи соответствующие бинарные операции не образуют замкнутое полукольцо. Будем обозначать задачи из класса APP соответствующими операциями MMA.

Таблица 4.13. Время выполнения всего набора тестов APP в секундах для типа double.

Задача	Размеры	gcc	clang	icc	polly	AOTO
MMA[+, max]	128	0.018	0.018	0.0022	0.0087	0.0025
	1024	65.68	65.76	1.58	11.49	1.59
	2000	710.74	715.17	12.88	52.83	12.73
	4000	7757.87	7874.29	112.73	459.73	110.27
MMA[/, max]	128	0.046	0.045	0.023	0.045	0.023
	1024	65.52	66.56	16.05	31.69	16.05
	2000	690.77	715.48	130.37	258.75	130.25
	4000	7741.98	7941.24	1130.5	2240.26	1127.04
MMA[min, max]	128	0.019	0.0015	0.002	0.008	0.0025
	1024	67.22	0.017	1.6	12.06	1.6
	2000	713.48	67.54	13.14	52.96	12.58
	4000	7721.54	717	111.15	469.8	109.33
MMA[×, max]	128	0.018	0.0179	0.002	0.0088	0.0017
	1024	65.16	66.41	1.5	11.5	0.92
	2000	695.18	687.73	12.16	53.31	7.52
	4000	7712.35	7650.57	104.35	462.37	63.85
MMA[×, min]	128	0.018	0.0177	0.0019	0.0087	0.0017
	1024	66.95	67.53	1.49	11.94	0.92
	2000	702.23	702.1	12.34	53.38	7.48
	4000	7860.59	7941.89	103.99	469.13	64.36
MMA[×, −]	128	0.0181	0.0178	0.0019	0.0086	0.0017
	1024	65.68	68.068	1.51	11.58	0.92
	2000	703.52	705.28	12.11	53.24	7.45
	4000	7920.18	7788.63	106.76	456.99	64.34
MMA[+, min]	128	0.018	0.018	0.0022	0.0086	0.0025
	1024	66.19	66.32	1.58	11.76	1.59
	2000	706.39	703.77	12.69	52.83	12.67
	4000	7942.11	7970.94	111.11	464.59	111.31

Таблица 4.14. Весь набор тестов APP для типа float.

Задача	Размеры	gcc (%)	clang (%)	icc (%)	polly (%)	AOTO (%)	Верхняя граница (ГФлоп/сек)
MMA[+, max]	128	6.05	6.15	**91.94**	11.38	70.99	30.4
	1024	1.09	1.09	84.01	6.18	**89.28**	
	2000	1.02	1.02	87.73	11.58	**89.38**	
	4000	0.79	0.79	86.61	11.55	**89.34**	
MMA[/, max]	128	23.68	23.68	80.92	23.68	**90.13**	3.04
	1024	10.86	11.18	89.47	24.01	**94.41**	
	2000	10.2	10.2	88.49	24.01	**95.07**	
	4000	7.89	7.89	90.79	24.01	**95.39**	
MMA[min, max]	128	5.33	6.18	**90.89**	11.35	71.22	30.4
	1024	1.09	1.09	85.26	6.15	**89.47**	
	2000	1.02	1.02	87.27	11.64	**89.74**	
	4000	0.79	0.79	86.41	11.58	**90.39**	
MMA[×, max]	128	2.93	2.93	47.71	5.66	**62.38**	60.8
	1024	0.54	0.54	43.19	3.11	**62.38**	
	2000	0.51	0.51	46.3	5.74	**61.38**	
	4000	0.39	0.39	46.05	5.76	**62.23**	
MMA[×, min]	128	2.93	2.98	**47.6**	5.67	43.37	60.8
	1024	0.54	0.54	43.32	3.06	**56.74**	
	2000	0.51	0.51	46.88	5.71	**62.1**	
	4000	0.39	0.39	46.73	5.76	**62.86**	
MMA[×, −]	128	2.91	2.98	**46**	5.74	42.35	60.8
	1024	0.54	0.54	43.63	3.11	**58.38**	
	2000	0.51	0.51	47.27	5.74	**62.83**	
	4000	0.39	0.39	46.25	5.77	**59.77**	
MMA[+, min]	128	6.09	6.18	**91.94**	11.38	70.99	30.4
	1024	1.08	1.08	84.01	6.18	**89.28**	
	2000	1.02	1.02	87.73	11.58	**89.38**	
	4000	0.79	0.79	86.61	11.55	**89.34**	

В табл. 4.12 и табл 4.14 представлены результаты измерений однопоточной производительности ПС AOTO и современных компиляторов на целевой аппаратной платформе с процессором Intel Sandy Bridge (табл. 4.2) для плотных квадратных матриц, содержащих элементы типа double и float, соответственно. В случае компилятора ICC использовалась опция -qopt-matmul, чтобы обеспечить распознавание и замену MMM на вызов реализации MMM доступной в библиотеке Intel MKL (см. табл. 4.3). Рассматривались матрицы размерности 128×128, 1024×1024, 2000×2000 и 4000×4000. Табл. 4.13 и табл. 4.15 содержат реальное время вычисления

в секундах для типа double и float, соответственно. Высокопроизводительные реализации интерфейса GraphBLAS не рассматривались, в силу их отсутствия для данной платформы.

Таблица 4.15. Время выполнения всего набора тестов APP в секундах для типа float.

Задача	Размеры	gcc	clang	icc	polly	AOTO
MMA[+, max]	128	0.0159	0.016	0.0011	0.0085	0.0014
	1024	64.81	64.81	0.84	11.43	0.79
	2000	567.59	567.59	6.59	49.99	6.48
	4000	6395.74	6395.74	58.34	437.46	56.05
MMA[/, max]	128	0.04	0.04	0.011	0.041	0.011
	1024	64.34	63.73	7.88	29.42	7.48
	2000	572.13	568.69	65.55	240.88	60.86
	4000	6305.08	6296.93	556.046	2097.06	529.17
MMA[min, max]	128	0.018	0.0156	0.0011	0.0085	0.0014
	1024	64.28	64.16	0.83	11.46	0.79
	2000	571.76	572.73	6.63	49.68	6.45
	4000	6270.27	6275.81	58.46	436.66	55.88
MMA[×, max]	128	0.016	0.016	0.001	0.0085	0.0012
	1024	64.39	64.22	0.82	11.37	0.56
	2000	566.4	568.37	6.25	50.45	4.72
	4000	6330.78	6317.34	54.85	439.47	40.59
MMA[×, min]	128	0.016	0.016	0.001	0.0085	0.0011
	1024	64.48	64.38	0.82	11.56	0.62
	2000	568.29	567.18	6.17	50.65	4.66
	4000	6290.39	6358.57	54.069	438.79	40.18
MMA[×, −]	128	0.0165	0.016	0.001	0.008	0.0011
	1024	64.55	64.26	0.809	11.34	0.6
	2000	567.27	567.99	6.12	50.41	4.61
	4000	6383.67	6330.16	54.62	437.13	42.27
MMA[+, min]	128	0.016	0.016	0.0011	0.0085	0.0014
	1024	64.23	64.3	0.84	11.41	0.79
	2000	566.35	567.39	6.6	49.95	6.48
	4000	6306.33	6289.9	58.33	437.7	56.56

Для вычисления верхней теоретической границы каждой оцениваемой операции MMA на количество операций с плавающей точкой в секунду (ГФлоп/сек), применяется следующая формула [174]: верхняя теоретиче-

ская граница = (Частота процессора в ГГц) × (Количество инструкций, выполняемых процессором за один такт). Вследствие этого теоретическая граница определяется значением N_{VMMA}. Например, если элементы матриц имеют тип double, 8 операций с плавающей точкой могут быть выполнены за один такт процессора в случае MMA[×, min], MMA[×, max], и MMA[×, −]. В случае MMA[min, max], MMA[+, max] и MMA[+, min] за один такт могут быть выполнены только 4 операции с плавающей точкой. В случае MMA[/, max] только 0.4 операций с плавающей точкой могут быть выполнены за один такт процессора, так как соответствующее N_{VMMA} равно 0.05.

Для задач нахождения путей наименьшей и наибольшей надежности ПС АОТО достигла больше 69% верхней теоретической границы производительности. Для остальных оцениваемых задач она достигла 85% верхней теоретической границы производительности. Как было показано в разделе 4.2.1, устранение проблем векторизации, выполняемой библиотекой LLVM Core в ПС АОТО, может позволить достичь лучших значений производительности для решения задач APP.

В случае операций MMA[×, −] и MMA[/, max] ПС АОТО достигла 78% и 89% верхней теоретической границы производительности. Таким образом, данный подход может быть применен также в случаях, когда составляющие MMA бинарные операции не образуют замкнутое полукольцо.

Для всех рассматриваемых задач ПС АОТО достигла 1.56-кратного ускорения по сравнению с компилятором ICC. Можно сделать вывод о том, что стратегия оптимизации, используемая компилятором ICC по умолчанию, позволяет компилятору ICC достичь однопоточную производительность ПС АОТО для MMA, операции которой не могут выполняться Sandy Bridge одновременно (MMM[+,max], MMA[min,max], MMA[+,min]). Отметим, что распознавание и замена MMM на вызов реализации MMM доступной в библиотеке Intel MKL не повлияло в данном случае на производительность компилятора ICC.

4.2.7. Частные случаи общей задачи о путях. Многопоточная производительность

Рассмотрим многопоточную производительность ПС AOTO и компилятора ICC для решения задач из класса APP. Для выполнения эксперимента использовалась платформа с 4-ядерным процессором Intel Sandy Bridge (табл. 4.2). В случае компилятора ICC использовалась опция -qopt-matmul, чтобы обеспечить распознавание и замену MMM на вызов реализации MMM доступной в библиотеке Intel MKL (см. табл. 4.3).

Рис. 4.15 содержит график многопоточной производительности для задачи нахождения кратчайших путей и задачи нахождения путей наибольшей надежности для четырех потоков. Рассматривались плотные квадратные матрицы, содержащие элементы типа double. Размерность матриц изменялась от 32 до 4000 с шагом 2. Табл. 4.16 содержит реальное время вычисления в секундах для задачи нахождения кратчайших путей и задачи нахождения путей наибольшей надежности. Задачи из класса APP обозначены соответствующими операциями MMA.

Таблица 4.16. Время выполнения решений APP в секундах для типа double.

Задача	Размеры	icc	AOTO
MMM[+, max]	32	0.000028	0.00017
	1024	0.427	0.429
	2048	3.75	3.59
	4000	28.99	28.19
MMM[×, max]	32	0.000036	0.00019
	1024	0.41	0.31
	2048	3.59	2.51
	4000	28.36	20.17

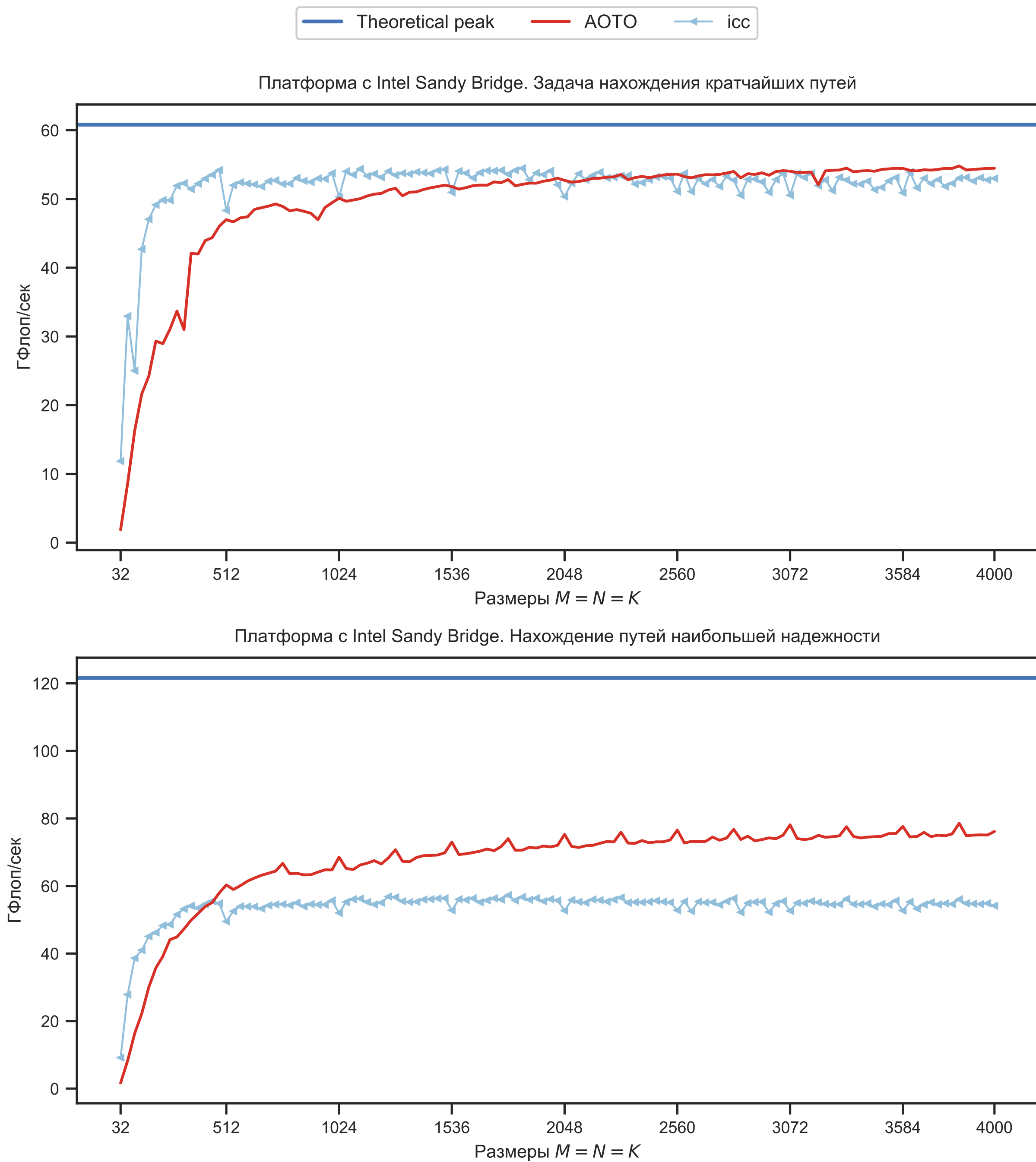


Рис. 4.15. Задача APP для типа double.

В случае задачи нахождения кратчайших путей ПС АОТО достигает 90.11% пиковой производительности. Компилятор ICC достигает 89.47% пиковой производительности. Можно сделать вывод о том, что стратегия оптимизации, используемая компилятором ICC по умолчанию, позволяет компилятору ICC достичь многопоточную производительность ПС АОТО для MMM[+, max], операции которой не могут выполняться Sandy Bridge одновременно. Как в случае ПС АОТО, так и в случае компилятора ICC

наблюдается почти 4-кратное ускорение по сравнению с однопоточным решением.

В случае задачи нахождения путей наибольшей надежности ПС АОТО достигает 64.55% пиковой производительности и 3.3-кратного ускорения по сравнению с однопоточным решением. Как было показано в разделе 4.2.1, устранение проблем векторизации, выполняемой библиотекой LLVM Core в ПС АОТО, может позволить достичь лучших значений производительности для решения задач APP. Компилятор ICC достигает 47.09% пиковой производительности и почти 3.8-кратного ускорения по сравнению с однопоточным решением.

4.3. Оценка алгоритма вычисления обобщенного матрично-векторного произведения

Сравним однопоточную и многопоточную производительности реализаций алгоритмов 7 и 8 с производительностью реализаций MVM, доступных в библиотеках Intel MKL, BLIS и OpenBLAS. Проверим, что формулы для определения значений параметров реализаций алгоритмов 7 и 8 позволяют получить значения, полученные ручной настройкой, и близкие к ним.

Таблица 4.17. Особенности целевых аппаратных платформ.

Название	ЦПУ	Частота (ГГц)	ОЗУ (Гбайт)	N_{VEC} для double	L_{VFMA}	N_{VFMA}	N_{prefetch}	L_{VLOAD}	N_{REG}	L_{ADD}
Core	Intel Xeon E5450	3.0	16	2	8	1	1	4	16	3
Sandy Bridge	Intel Xeon E5-2660	2.2	96	4	8	1	2	4	16	3
Broadwell	Intel Xeon E5-2697 v4	2.3	256	4	5	2	2	4	16	3
K10	AMD Opteron 8354	2.2	16	2	8	1	2	4	16	4

Таблица 4.18. Особенности целевых аппаратных платформ. Кэш-память.

Название	ЦПУ	S_{L_1} (Кбайт)	W_{L_1}	S_{L_2} (Кбайт)	W_{L_2}	S_{L_3} (Кбайт)	W_{L_3}	C_{L_i} (Мбайт)	N_{L_1}	N_{L_2}	N_{L_3}
Core	Intel Xeon E5450	32	8	6144	24	-	-	64	64	4096	-
Sandy Bridge	Intel Xeon E5-2660	32	8	256	8	20	20	64	64	512	16384
Broadwell	Intel Xeon E5-2697 v4	32	8	256	8	30	20	64	64	512	24576
K10	AMD Opteron 8354	64	2	512	16	2	32	64	512	512	1024

Для выполнения экспериментов использовались процессоры AMD Opteron 8354 (rev. B3), Intel Xeon E5-2697 v4, Intel Xeon E5450 и Intel Xeon E5-2660, имеющие различную архитектуру (K10, Broadwell, Core, и Sandy Bridge) и организацию кэш-памяти (различное количество уровней кэш-памяти, различный размер кэш-уровней и их ассоциативность) (см. табл. 4.17 и табл. 4.18). Результаты экспериментов, приведенные в данной главе, являются средним арифметическим результатов, полученных для проводимых экспериментов. Каждый эксперимент проводился повторно, пока доверительный интервал с уровнем доверия 95% не был в 10% от сообщаемого среднего арифметического значения.

Таблица 4.19. Время вычисления MVM вида $y = A^\mathsf{T}x + y$ в секундах для плотных квадратных матриц размерности $N \times M$ на Sandy Bridge.

Реализация \ $N = M$	6400	12800	19200	25600
Алгоритм 5	0.0253	0.105	0.226	0.4178
Алгоритм 5 (другой подход)	0.031	0.106	0.267	0.426
Intel MKL	0.0241	0.116	0.252	0.465
BLIS	0.0556	0.2385	0.521	0.8654
OpenBLAS	0.0416	0.1169	0.3348	0.56

4.3.1. Однопоточная производительность

Сравним однопоточную производительность реализаций алгоритмов 7 и 8 с однопоточной производительностью реализаций MVM, доступных в библиотеках Intel MKL, BLIS и OpenBLAS. Рис. 4.16 и рис. 4.17 содержат графики однопоточной производительности для MVM вида $y = A^\mathsf{T}x + y$.

Рис. 4.18 и рис. 4.19 содержат графики однопоточной производительности для MVM вида $y = Ax + y$, соответственно.

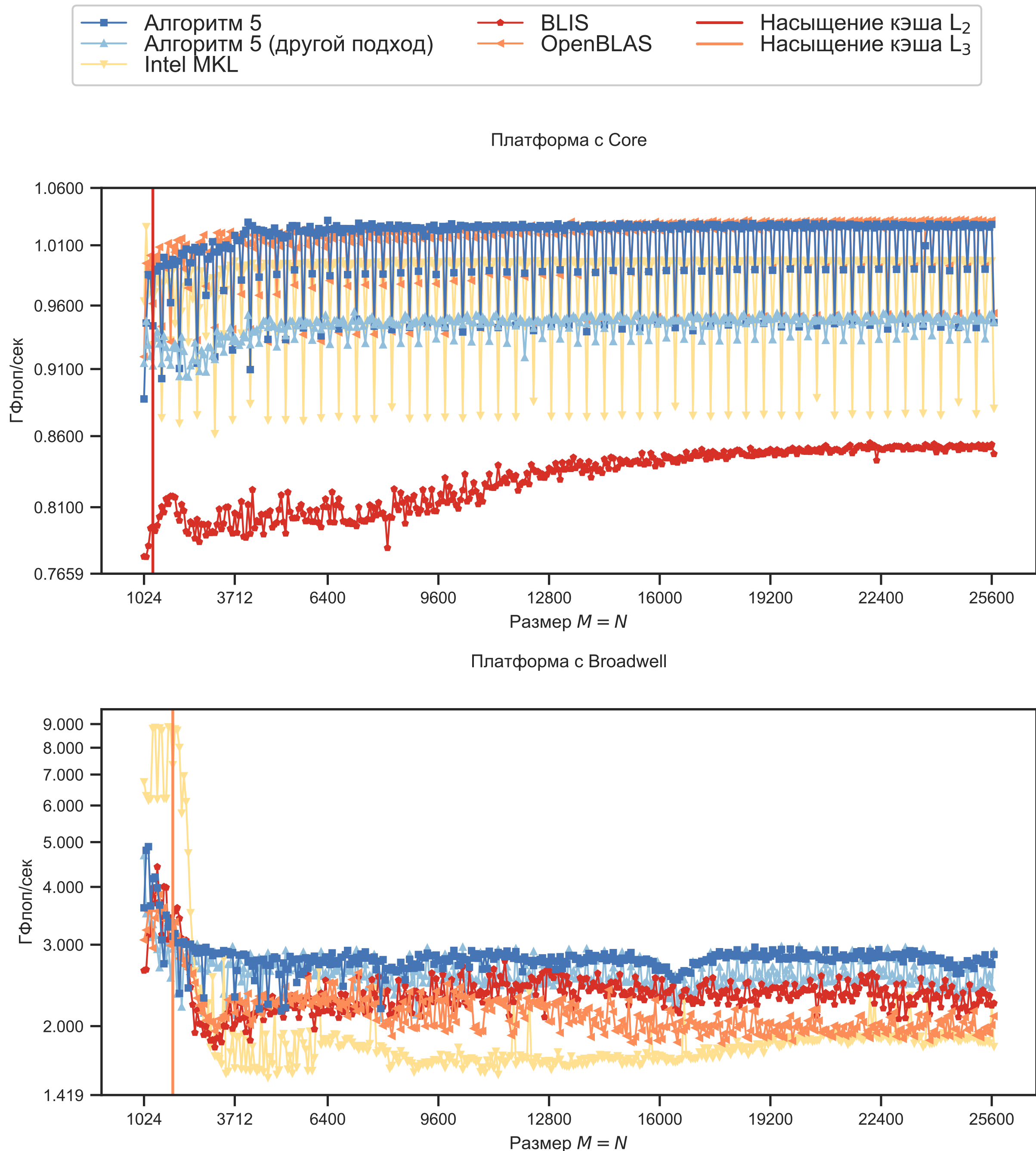


Рис. 4.16. $y = A^{\mathsf{T}}x + y$. Однопоточная производительность.

Табл. 4.19 и табл. 4.20 содержат время вычисления в секундах для MVM вида $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно, на Sandy Bridge. Рассматривались квадратные плотные матрицы. Матрицы и векторы содержали элементы типа double. Размерность матриц изменялась от 1024 до 25600 с шагом 64.

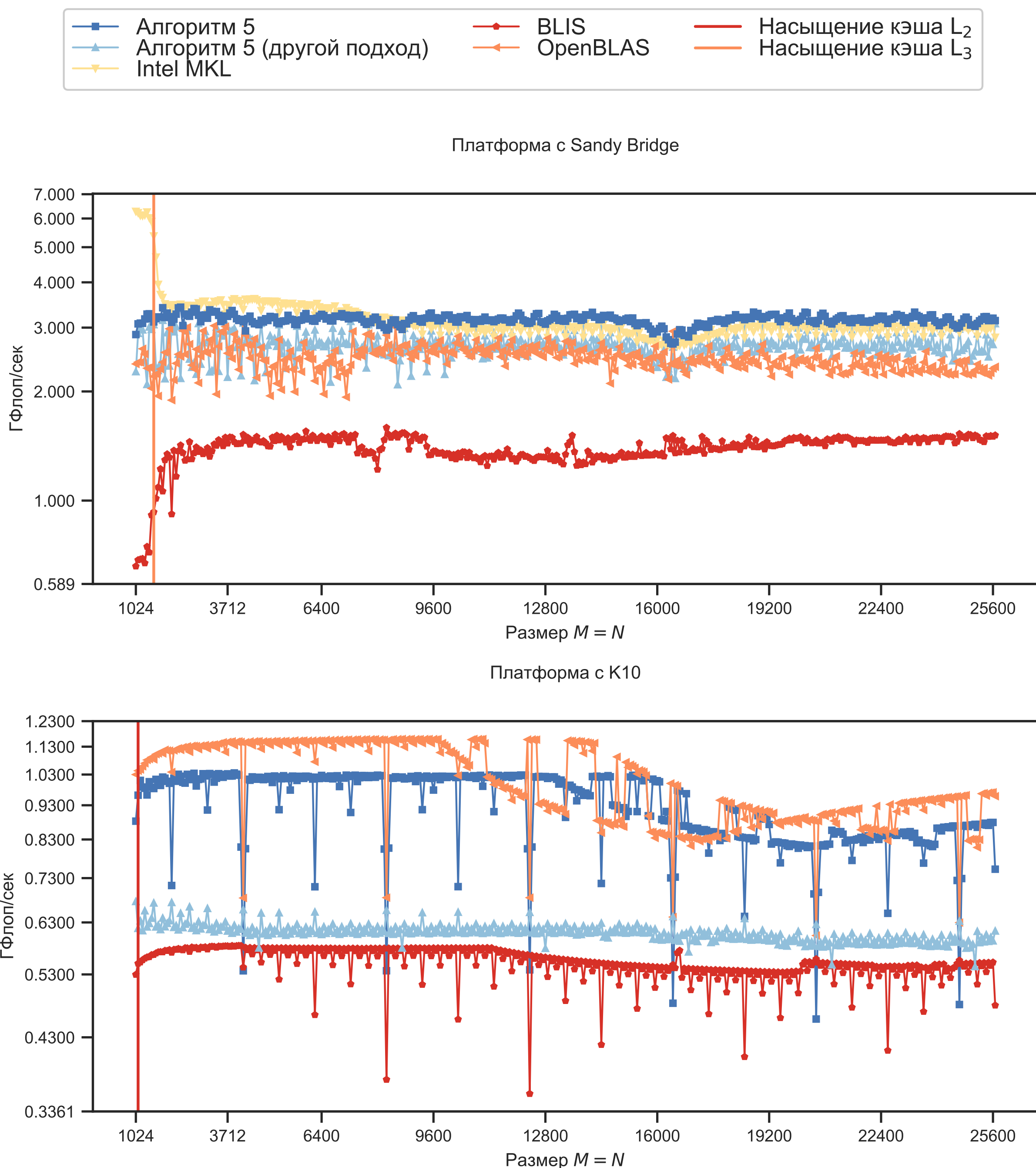


Рис. 4.17. $y = A^{\mathsf{T}}x + y$. Однопоточная производительность.

В случае рис. 4.16 и рис. 4.17 реализации алгоритма 7 в среднем достигается 1.038-кратное ускорение по сравнению с результатами, полученными для OpenBLAS, Intel MKL, и BLIS. В случае рис. 4.18 и рис. 4.19 реализация алгоритма 8 в среднем достигает 1.098-кратного ускорения по сравнению с рассмотренными библиотеками. Можно сделать вывод о том, что реализации алгоритма 7 и алгоритма 8, не требующие ручной настрой-

ки и автонастройки, сопоставимы по производительности с современными оптимизированными библиотеками, реализующими BLAS.

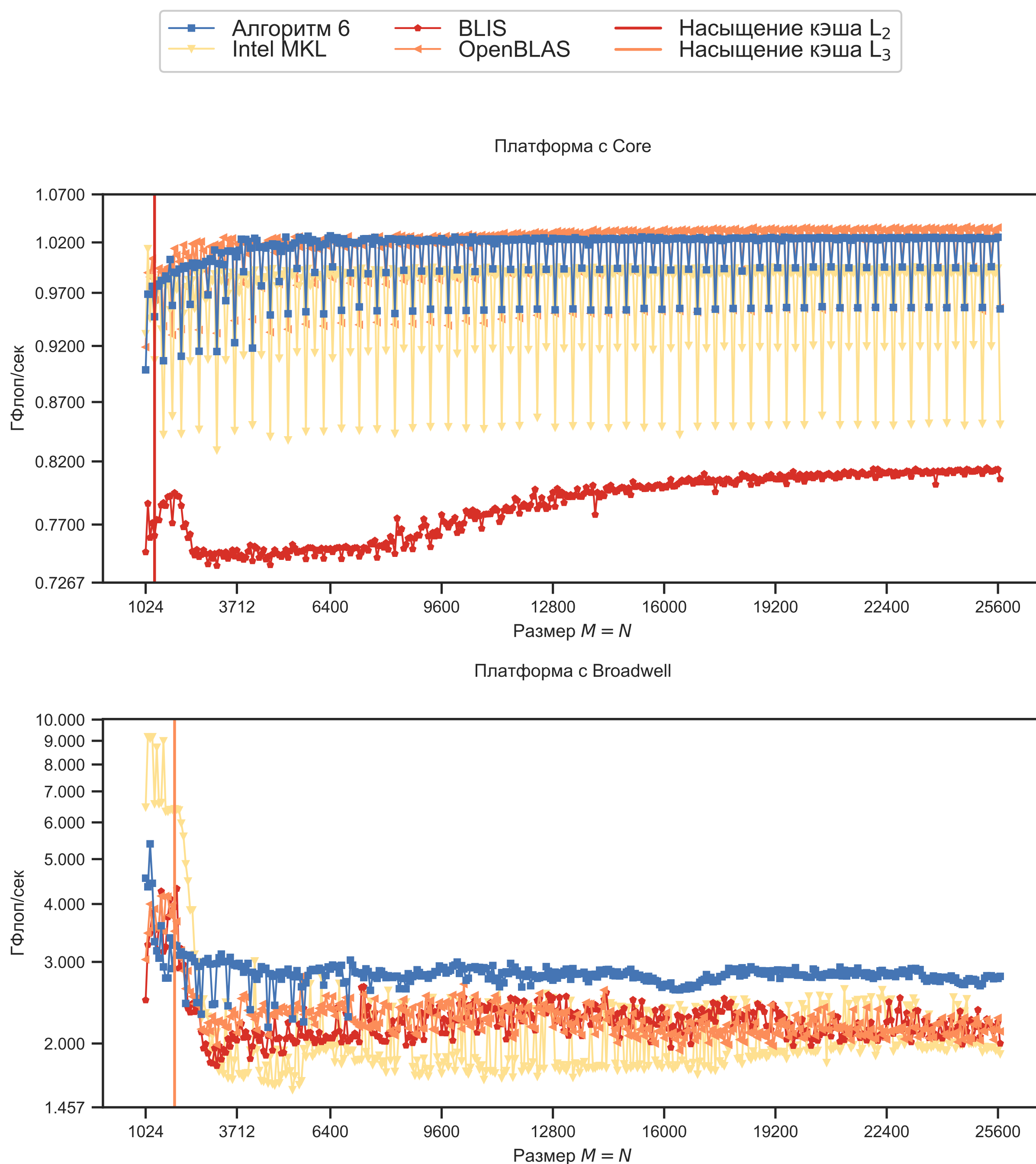
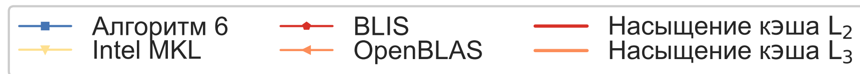
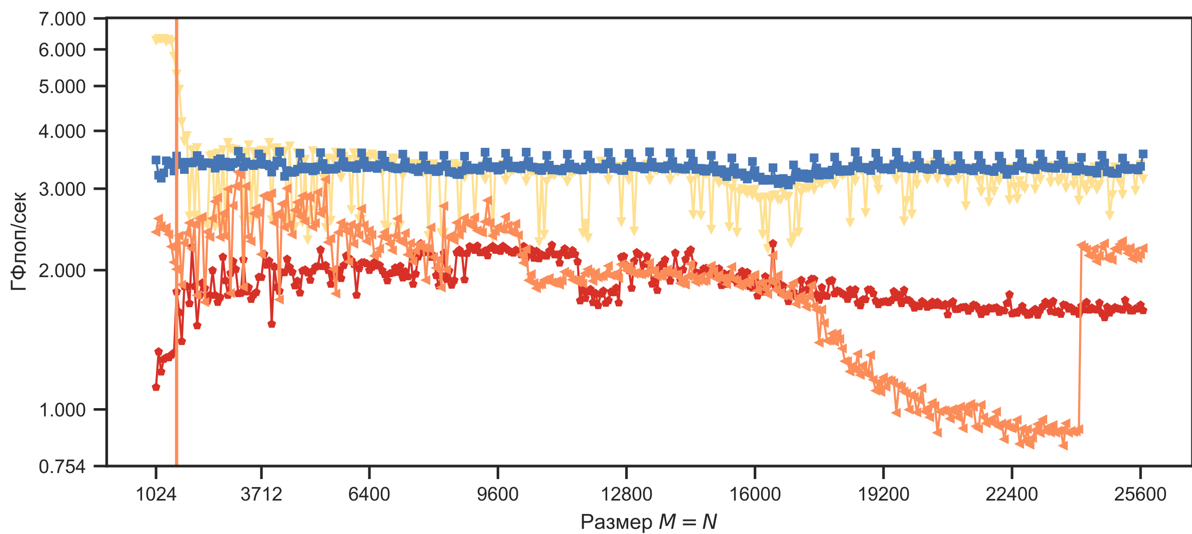


Рис. 4.18. $y = Ax + y$. Однопоточная производительность.

Таблица 4.20. время вычисления MVM вида $y = Ax + y$ в секундах для плотных квадратных матриц размерности $N \times M$ на Sandy Bridge.

Реализация \ $N = M$	6400	12800	19200	25600
Алгоритм 6	0.02436	0.092	0.222	0.3675
Intel MKL	0.0229	0.104	0.221	0.416
BLIS	0.0404	0.1642	0.43046	0.801
OpenBLAS	0.0343	0.161	0.6586	0.5871





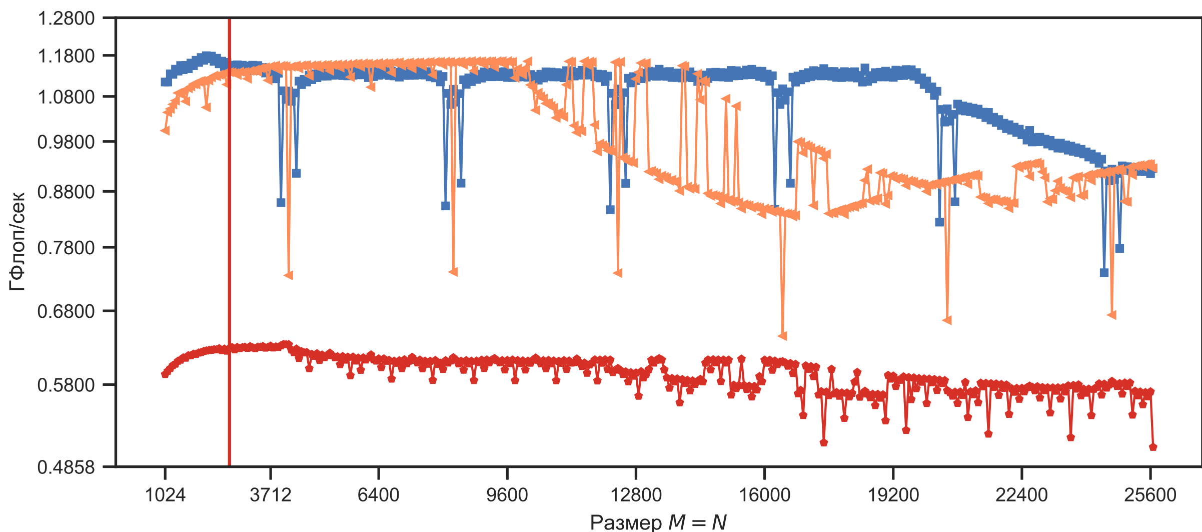


Рис. 4.19. $y = Ax + y$. Однопоточная производительность.

Отметим, что для размерностей матриц и векторов не кратных размерностям блоков, используемых для разбиения в алгоритмах 7 и 8, наблюдается колебание производительности в пределах 0.1 ГФлоп/сек. Это показывает, что для таких размерностей задачи формулы определения значений параметров реализаций алгоритма 7 и алгоритма 8 (см. раздел 2.3) могут использоваться только как приблизительная оценка. Можно сделать вывод о том, что подход для определения значений параметров M_c и N_c алгоритма 7, ориентированный на использование L_2(см. раздел 2.3), достигает лучшей однопоточной производительности по сравнению с подходом, ориентированным на использование L_1(см. раздел 2.3).

Отметим, что в случае Core реализации алгоритма 7 и алгоритма 8 достигают меньшей производительности для малых размеров матриц. Несмотря на это, в случае Broadwell и Sandy Bridge производительность для таких размерностей задачи больше. По сравнению с Core у Broadwell, K10 и Sandy Bridge имеется L_3, сокращающий количество обращений к основной памяти, вызванной промахами L_2. В случае алгоритма 7 и алгоритма 8 при малых размерах M и N элементы матрицы размерности $M \times N$ и векторов размерности M и N могут оставаться в L_3 во время всего выполнения алгоритмов. Рис. 4.16, рис. 4.17, рис. 4.18 и рис. 4.19 содержат вертикальные линии $M = N = 1982$ и $M = N = 1618$, обозначающие соответствующие размерности задачи для Broadwell и Sandy Bridge, соответственно. В случае K10, линия $M = N = 512$ не представлена на графиках. Так как L_3 физически индексируем, такое предположение является грубой оценкой (см. раздел 2.1). В частности, производительность уменьшается от 5 ГФлоп/сек до 3 ГФлоп/сек с ростом размерности задач, оцениваемых на Broadwell. В случае Sandy Bridge для размерностей задач меньших чем $M = N = 1618$ не видна разница производительностей из-за физической индексируемости L_3 и его меньшей чем у Broadwell размерности.

Несмотря на то, что у Core отсутствует L_3, производительность улучшается от 0.97 ГФлоп/сек до 1.03 ГФлоп/сек по мере того как размерность задачи увеличивается от 1024 до 6400 и L_2 начинает использоваться эффективнее. Если L_2 содержит все элементы $M_c \times N_c$ подматрицы матрицы A и подвектора размерности N_c, используемые итерацией цикла с индуктивной переменной i_c, то элементы рассматриваемой подматрицы могут быть замещены элементами этой же подматрицы, используемыми на следующей итерации данного цикла. В таком случае подвектор размерности N_c смог бы оставаться в L_2 без вытеснения. Однако, так как L_2 физически адресуем, элементы матрицы могут вытеснить элементы описываемого вектора. Так как все виртульные адреса отдельно рассматриваемой страницы памяти имеют разные физические адреса, и размер каждой страницы памяти обычно равен 4096 байт, $4096W_{L_2}/S_{DATA}$ элементов могут содержаться в L_2. Рис. 4.16 и рис. 4.18 содержат вертикальную линию $M = N = 1365$, обозначающую соответствующие размерности задач для Core. В случае K10 вертикальные линии $M = N = 1170$ и $M = N = 2730$ обозначают соответствующие размерности задач на рис. 4.17 и рис. 4.19, соответственно. Данное предположение является грубой оценкой, так как чтение элементов матрицы не выполняется с единичным шагом. Следовательно, во время выполнения новой итерации цикла с индуктивной переменной i_c элементы матрицы могут вытеснить элементы подвектора, располагающиеся в L_2. Размер L_2, значение параметра M_c и наличие L_3 определяют количество элементов, вытесняемых в основную память. Например, в случае K10 и $y = Ax + y$ производительность уменьшается от 1.1 ГФлоп/сек до 0.9 ГФлоп/сек по мере роста размерности задачи от 19200 до 25600. Производительность не уменьшается в случае Core, имеющего примерно в три раза больше памяти L_2. В случае K10 и $y = A^{\mathsf{T}}x + y$, используются большие значения M_c. Вследствие этого производительность уменьшается при увеличении размерности от 12800 до 25600.

Таблица 4.21. Значения C_{PAR_0}

ЦПУ \ Количество потоков	1	2	4	8	16	32	36
Core	763	3886	4649	7958	39774	189503	179263
Broadwell	455	3410	4779	6501	8399	85658	14850

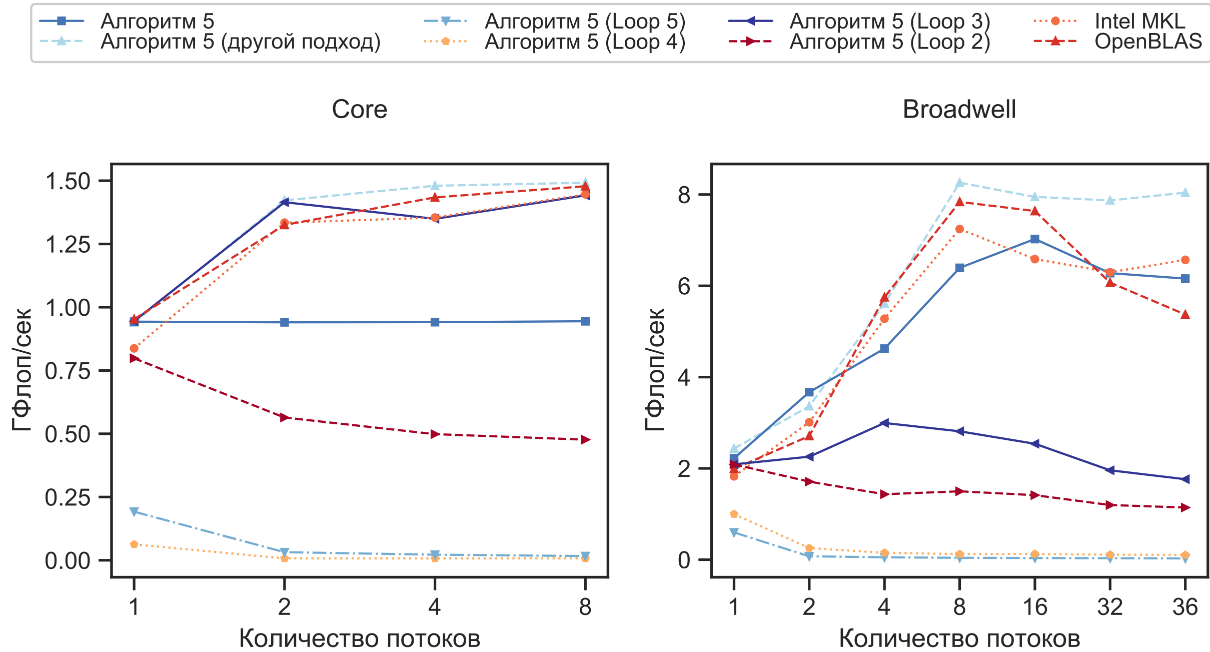


Рис. 4.20. $y = A^{\mathsf{T}}x + y$. Многопоточная производительность.

Таблица 4.22. Время вычисления MVM вида $y = A^{\mathsf{T}}x + y$ в секундах для плотных квадратных матриц размерности 25600×25600 и разного количества потоков на Broadwell.

Реализация \ Количество потоков	1	2	4	8	16	32	36
Алгоритм 5	0.54	0.39	0.23	0.158	0.165	0.17	0.163
Алгоритм 5 (Loop 2)	0.628	0.77	0.91	0.875	0.927	1.095	1.147
Алгоритм 5 (Loop 3)	0.628	0.58	0.44	0.466	0.5161	0.6693	0.744
Алгоритм 5 (Loop 4)	1.311	5.15	8.69	10.699	10.513	11.983	12.57
Алгоритм 5 (Loop 5)	2.193	17.8	25.09	32.002	36.585	41.304	47.925
Intel MKL	0.72	0.44	0.25	0.18	0.199	0.208	0.199
OpenBLAS	0.66	0.48	0.23	0.167	0.171	0.22	0.24
BLIS	0.63	0.74	0.65	0.74	0.73	0.696	0.68

4.3.2. Многопоточная производительность

Сравним многопоточную производительность реализаций алгоритмов 7 и 8 с многопоточной производительностью реализаций MVM, доступных в

библиотеках Intel MKL, BLIS и OpenBLAS для различного количества потоков, размерностей задач и плотных матриц. Рассматриваемые матрицы и векторы содержат элементы типа double. Эксперимент проводился на двух 4-ядерных Xeon E5450 и двух 18-ядерных Intel Xeon E5-2697 v4 (см. таблицу 4.17). Табл. 4.21 содержит значения C_{PAR_0}, вычисленные с использованием пакета EPCC [167].

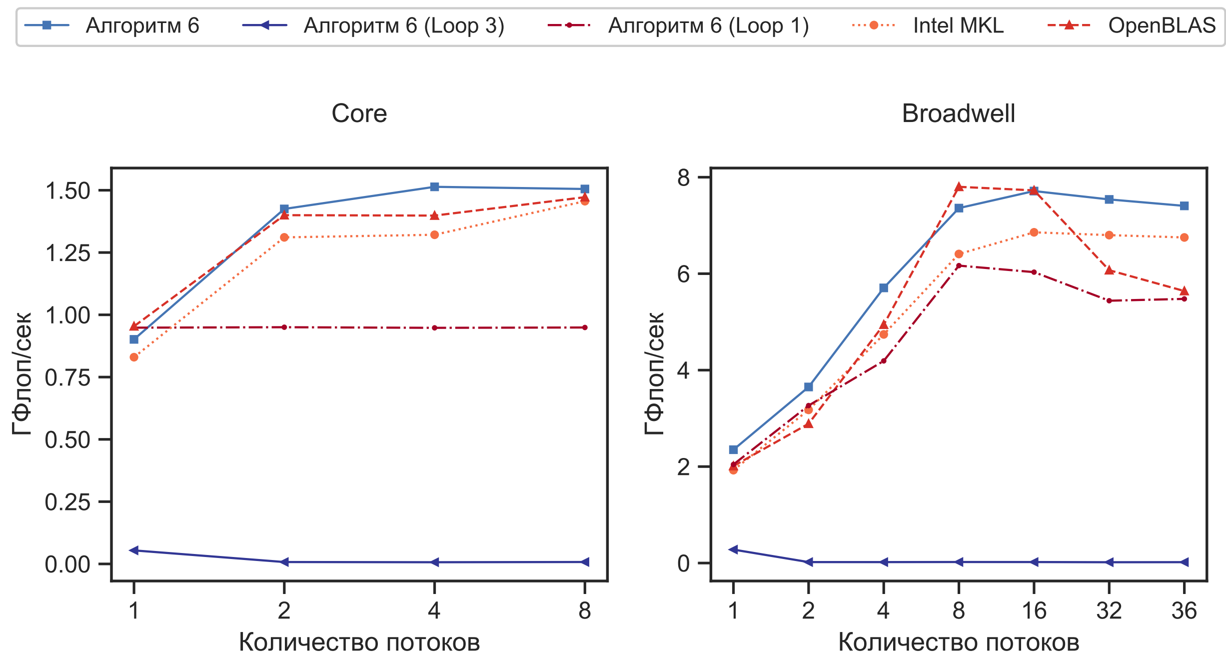


Рис. 4.21. $y = Ax + y$. Многопоточная производительность.

Таблица 4.23. Время вычисления MVM вида $y = Ax + y$ в секундах для плотных квадратных матриц размерности 25600×25600 и разного количества потоков на Broadwell.

Реализация \ Количество потоков	1	2	4	8	16	32	36
Алгоритм 6	0.56	0.36	0.23	0.18	0.169	0.173	0.176
Алгоритм 6 (Loop 1)	0.64	0.4	0.31	0.213	0.217	0.241	0.239
Алгоритм 6 (Loop 3)	4.698	65.89	66.91	60.79	63.09	77.1	71.15
Intel MKL	0.68	0.41	0.28	0.2	0.191	0.193	0.194
OpenBLAS	0.65	0.45	0.27	0.168	0.169	0.22	0.23
BLIS	0.64	0.68	0.74	0.8	0.79	0.7	0.695

Таблица 4.24. Циклы, для которых функция 3.1 возвращает наибольшие значения на Core ($M = N = 25600$).

Количество потоков / Случай MVM	1	2	4	8
$y = A^{\mathsf{T}}x + y$	Loop 1	Loop 1 (другой подход)		
$y = Ax + y$	Loop 1	Loop 2		

Таблица 4.25. Циклы, для которых функция 3.1 возвращает наибольшие значения на Broadwell ($M = N = 25600$).

Количество потоков / Случай MVM	1	2	4	8	16	32	36
$y = A^{\mathsf{T}}x + y$	Loop 1	Loop 1 (другой подход)					
$y = Ax + y$	Loop 2						

Рис. 4.20 и рис. 4.21 содержат многопоточную производительность реализаций $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$ для плотных квадратных матриц размерности 25600×25600, соответственно. Табл 4.22 и табл 4.23 содержат время вычисления в секундах для MVM вида $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно. Так как BLIS 0.2.2 поддерживает многопоточность только для третьего уровня интерфейса BLAS, фреймворк BLIS в данном сравнении не рассматривается. Табл. 4.24 и табл. 4.25 содержат циклы, выбранные для распараллеливания на основе функции 3.1, для Core и Broadwell, соответственно. В случае $y = A^{\mathsf{T}}x + y$ наибольшее значение функции 3.1 достигается для цикла с индуктивной переменной j_c, полученного альтернативным подходом к определению значений параметров M_c и N_c алгоритма 7 (см. раздел 3.3). Линия, отмеченная “Алгоритм 5 (другой подход)” на рис. 4.20, изображает производительность, полученную распараллеливанием описываемого цикла. По сравнению с результатами, полученными распараллеливанием других циклов алгоритма 7, используемый подход позволяет получить 1.075-кратное ускорение. В случае распараллеливания цикла с индуктивной переменной j_c, полученного не альтернативным подходом к определению значений параметров M_c и N_c алгоритма 7, достигается

меньшее ускорение. Причиной служит недостаточное количество итераций рассматриваемого цикла. Например, данный цикл имеет только одну итерацию в случае Core. Отмечается, что функция 3.1 предлагает неправильный цикл для распараллеливания в случае Broadwell и менее четырех потоков. Это объясняется неточной оценкой погрешности, возникающей в результате неэффективного использования L_2. В случае $y = Ax + y$ наибольшее значение функции 3.1 достигается для цикла с индуктивной переменной i_c. По сравнению с результатами, полученными распараллеливанием других циклов алгоритма 7, используемый подход позволяет получить 1.28-кратное ускорение. Можно сделать вывод о том, что во всех случаях реализации алгоритма 7 и алгоритма 8 превосходят оптимизированные библиотеки, достигая в среднем 1.05-кратного и 1.07-кратного ускорения по сравнению с OpenBLAS и Intel MKL, соответственно.

Таблица 4.26. Время вычисления MVM вида $y = A^{\mathsf{T}}x + y$ в секундах для плотных квадратных матриц размерности $N \times M$ и 36 потоков на Broadwell.

Реализация \ $N = M$	6400	12800	19200	25600
Алгоритм 5 (другой подход)	0.021	0.047	0.11	0.18
Алгоритм 5	0.044	0.088	0.17	0.22
Алгоритм 5 (Loop 2)	0.15	0.43	0.87	1.17
Алгоритм 5 (Loop 3)	0.19	0.49	0.71	1.1
Алгоритм 5 (Loop 4)	0.951	2.79	5.93	10.4
Алгоритм 5 (Loop 5)	3.34	11.47	25.79	42.89
Intel MKL	0.019	0.061	0.108	0.23
BLIS	0.039	0.21	0.34	0.76
OpenBLAS	0.019	0.057	0.12	0.238

Рис. 4.22 и рис. 4.23 содержат многопоточную производительность реализаций $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$ для плотных квадратных матриц. Размерность матриц изменялась от 1024 до 25600 с шагом 64. Количество потоков составляло 8 и 36 в случаях Core и Broadwell, соответственно. Табл. 4.26 и табл. 4.27 содержат время вычисления в секундах для MVM вида $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно, для плотных квадратных матриц размерности $N \times M$ и 36 потоков на Broadwell.

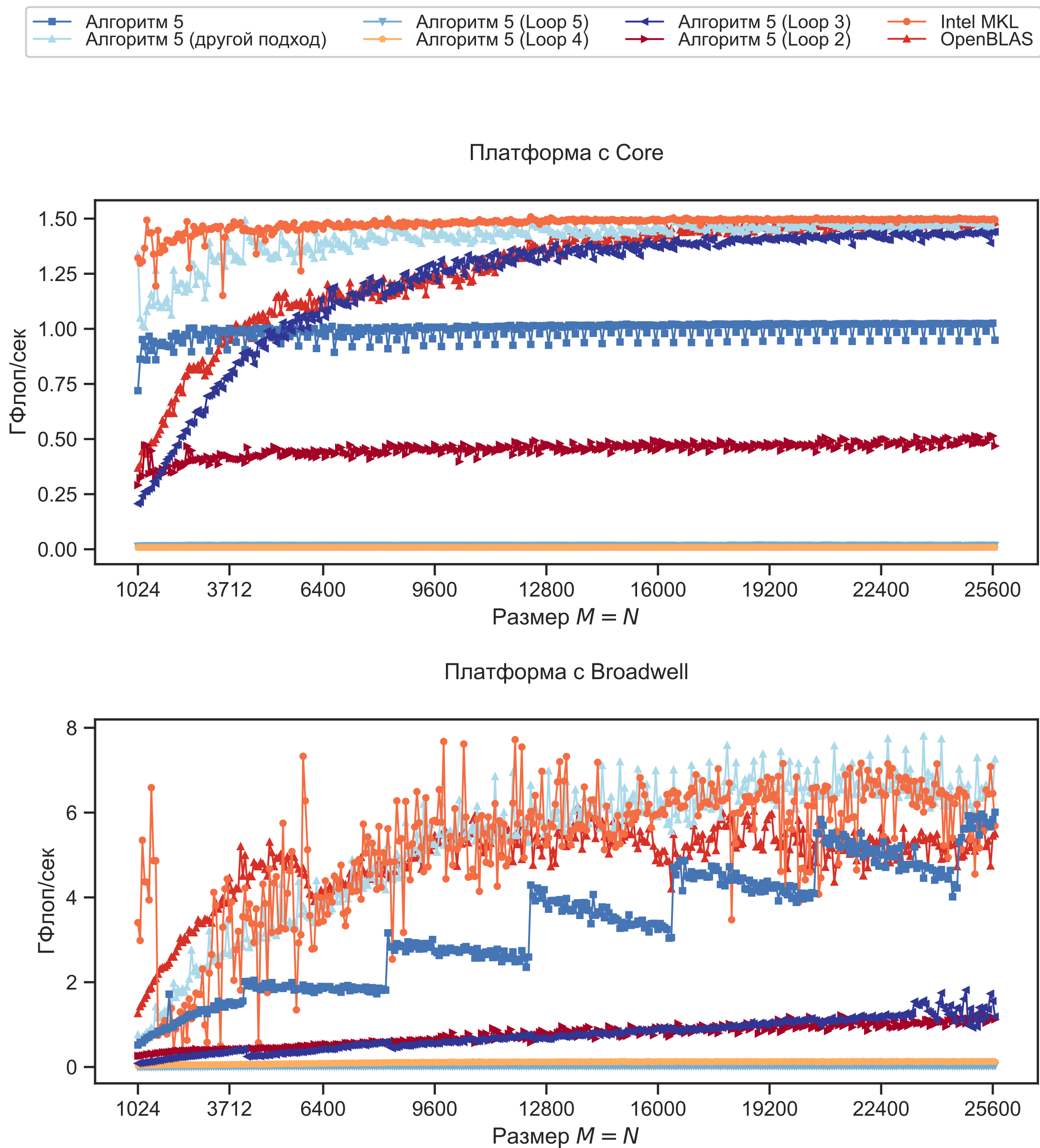


Рис. 4.22. $y = A^\top x + y$. Многопоточная производительность.

Таблица 4.27. Время вычисления MVM вида $y = Ax + y$ в секундах для плотных квадратных матриц размерности $N \times M$ и 36 потоков на Broadwell.

Реализация \ $N = M$	6400	12800	19200	25600
Алгоритм 6	0.018	0.045	0.104	0.18
Алгоритм 6 (Loop 1)	0.041	0.088	0.169	0.24
Алгоритм 6 (Loop 3)	4.75	18.32	40.88	71.15
BLIS	0.042	0.162	0.369	0.73
OpenBLAS	0.016	0.061	0.149	0.231
Intel MKL	0.025	0.048	0.104	0.176

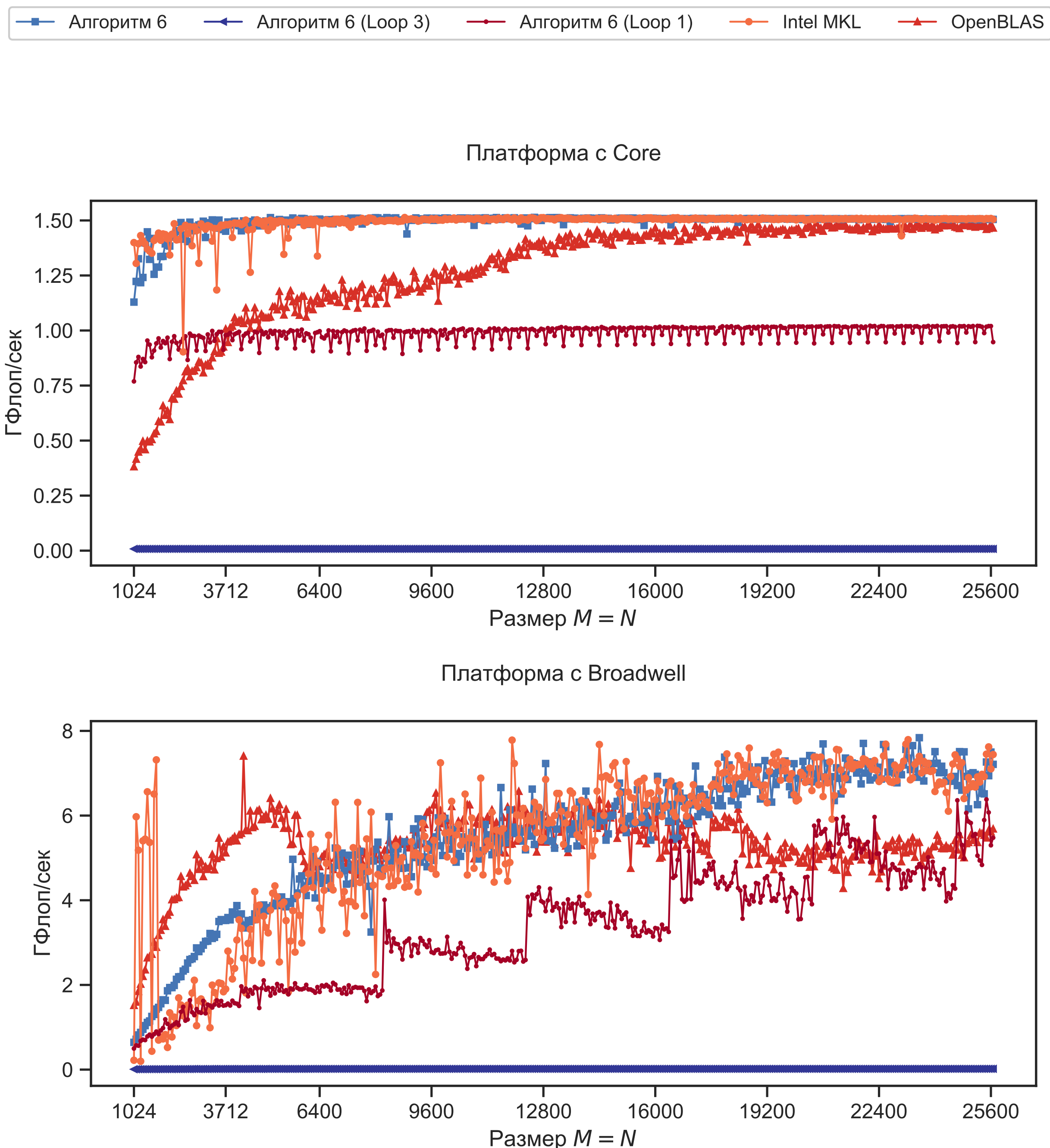


Рис. 4.23. $y = Ax + y$. Многопоточная производительность.

В случае $y = A^{\mathsf{T}}x + y$ наибольшее значение функции 3.1 достигается для цикла с индуктивной переменной j_c, полученного альтернативным подходом к определению значений параметров M_c и N_c алгоритма 7 (см. раздел 3.3). По сравнению с результатами, полученными распараллеливанием других циклов алгоритма 7, используемый подход позволяет получить 1.27-кратное ускорение. В случае $y = Ax + y$ наибольшее значение функции 3.1 достигается для цикла с индуктивной переменной i_c. По сравнению

с результатами, полученными распараллеливанием других циклов алгоритма 7, используемый подход позволяет получить 1.54-кратное ускорение. Можно сделать вывод о том, что во всех случаях реализации алгоритма 7 и алгоритма 8 в среднем не уступают производительности OpenBLAS, достигая 1-кратного ускорения по сравнению с ней. В случае с Intel MKL в среднем достигается 0.98-кратное ускорение.

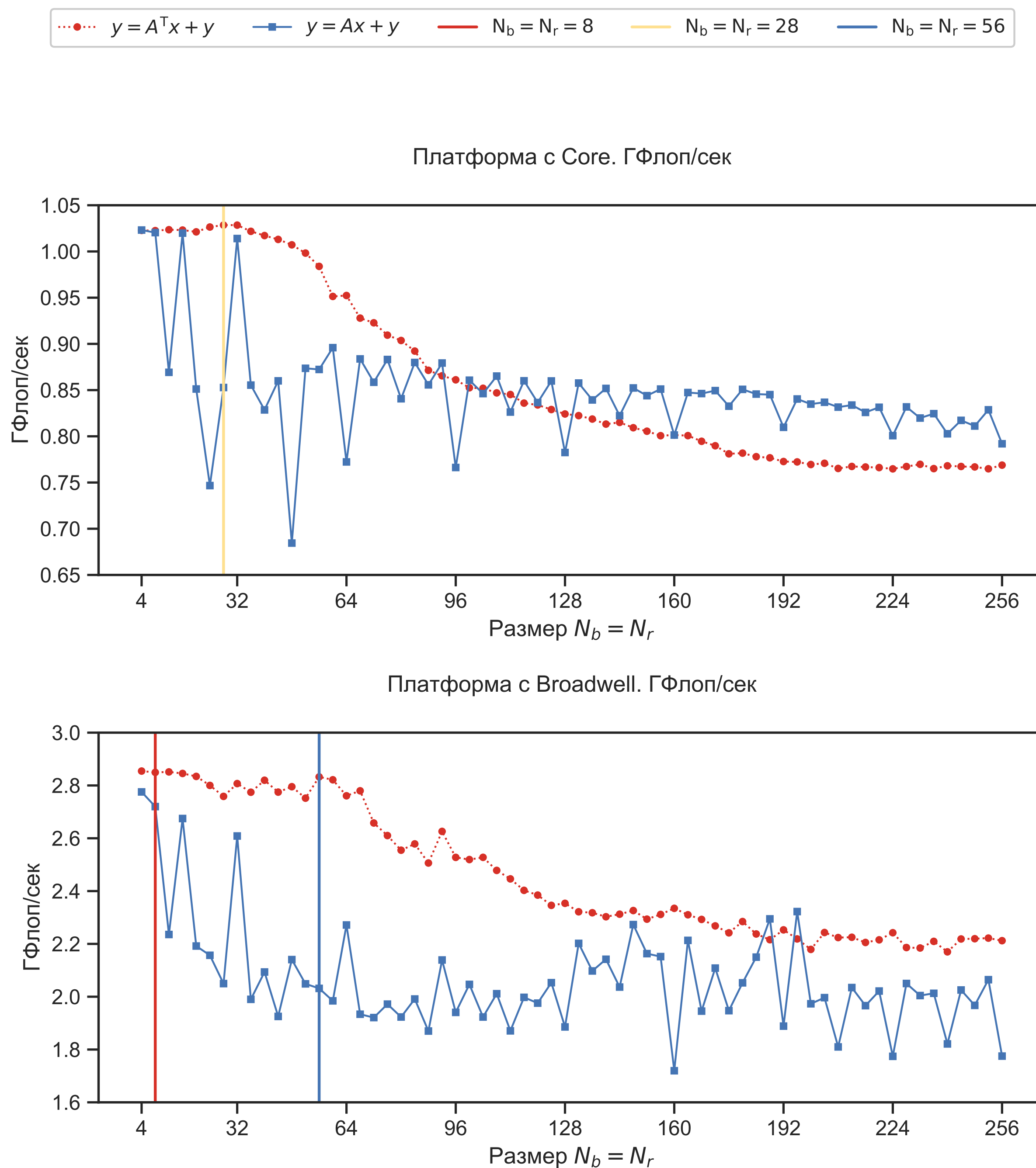


Рис. 4.24. Влияние значений N_b и N_r на однопоточную производительность.

4.3.3. Оценка значений параметров $\mathbf{N_b}$ и $\mathbf{N_r}$

Оценим точность вычисления значений параметров алгоритма 7 и алгоритма 8 формулами, выведенными в разделе 2.3 для плотных квадратных матриц размерности $M = N = 19152$ на Core и Broadwell. Так как такой размер матриц кратен размерам всех блоков алгоритма 7 и алгоритма 8, могут быть использованы предположения о значениях параметров (см. раздел 2.3).

Рис. 4.24 изображает значения однопоточной производительности для двух частных случаев MVM $y = A^\mathsf{T}x + y$ и $y = Ax + y$. Рассматриваются различные значения параметров N_b и N_r, изменяющиеся от 4 до 256 с шагом 4. В случае $y = A^\mathsf{T}x + y$ значение параметра N_r приравнивалось к значениям параметра N_{VEC}.

Для Core применяемые формулы определяют наилучшие значения $N_b = 28$ и $N_r = 2$ для $y = A^\mathsf{T}x + y$ и $y = Ax + y$, соответственно. В случае $y = Ax+y$ значение $N_r = 2$ не представлено на рис. 4.24 для соответствия с Broadwell, для которого требуется по крайней мере четыре элемента, чтобы заполнить векторный регистр. Если значения N_b и N_r больше чем 28 и 2, соответственно, то создается нехватка векторных регистров, что приводит к потери производительности от 1.03 ГФлоп/сек до 0.77 ГФлоп/сек в случае $y = A^\mathsf{T}x$ и от 1.03 ГФлоп/сек до 0.79 ГФлоп/сек в случае $y = Ax + y$, соответственно. В случае $y = A^\mathsf{T}x + y$ производительность возрастает от 1.02 ГФлоп/сек до 1.03 ГФлоп/сек по мере увеличения значений параметра N_b от 4 до 28, вследствие роста количества использований элементов вектора x и количества повторно используемых элементов вектора y.

В случае Broadwell и $y = Ax + y$ производительность уменьшается от 2.78 ГФлоп/сек до 1.78 ГФлоп/сек с увеличением значения параметра N_r от 4 до 256 и нехваткой векторных регистров. Используемая формула определяет неоптимальное значение $N_r = 8$, в котором достигается

97.8% наилучшей производительности. Причина заключается в издержках, создаваемых редукцией элементов вектора y, амортизируемых относительно N_c/N_r итераций. В случае Broadwell и $y = A^\mathsf{T}x + y$ производительность уменьшается от 2.85 ГФлоп/сек до 2.8 ГФлоп/сек с увеличением значений параметра N_b от 4 до 32. В случае рассматриваемой платформы Clang выполняет размотку цикла i_b алгоритма 7, чтобы использовать большее количество векторных регистров для хранения вектора x. Производительность увеличивается от 2.8 ГФлоп/сек до 2.83 ГФлоп/сек с увеличением значения параметра N_b от 32 до 56 и уменьшается до 2.2 ГФлоп/сек с дальнейшим ростом значений параметра N_b от 32 из-за ограниченного количества векторных регистров. Можно сделать вывод о том, что формулы предсказывают неоптимальное значение $N_b = 56$, для которого достигается 99.3% наилучшей производительности.

4.3.4. Оценка значений парамета D

Рис. 4.25 и рис. 4.26 содержат графики изменения однопоточной производительности и количества промахов L_1 для $y = A^\mathsf{T}x + y$ и $y = Ax + y$, соответственно. Рассматривались различные значения параметра D, изменяющиеся от 0 до 40 с шагом 1.

В случае Core и $y = A^\mathsf{T}x + y$ количество промахов L_1 уменьшается от 73030701 до 66918671 с увеличением значений параметра D от 0 до 2 и увеличивается до 224090891, если значения параметра D продолжают расти. Эти изменения влияют на производительность, возрастающую от 0.98 ГФлоп/сек до 1.03 ГФлоп/сек с увеличением значений параметра D от 0 до 2 и уменьшается до 1 ГФлоп/сек, если значения параметра D продолжают расти. Схожее поведение наблюдается для $y = Ax + y$ на Core и обоих случаев MVM, рассмотренных на Broadwell. В случае Core и $y = Ax + y$ наименьшее количество промахов L_1 и наилучшая производительность достигаются при $D = 2$. В случае Broadwell наименьшее количество промахов

L_1 достигается при $D = 3$ и $D = 2$ для $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно.

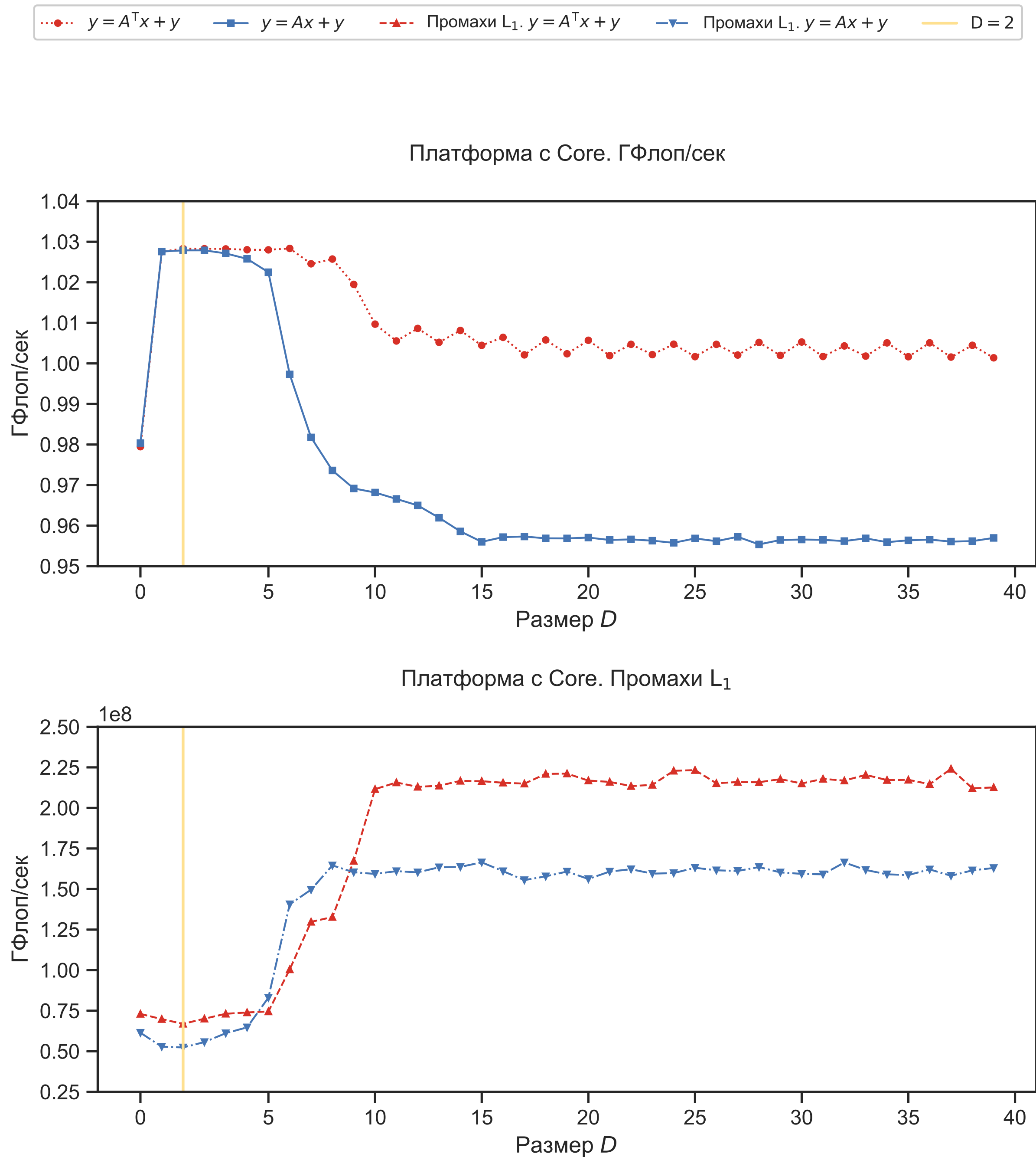


Рис. 4.25. Оценка влияния значений параметра D на производительность на Core.

Полученные с помощью формул значения параметра D для Core являются наилучшими для обоих случаев MVM. В случае Broadwell полученные значения D являются наилучшими только для $y = A^{\mathsf{T}}x + y$. Для $y = Ax + y$ вычисленное значение позволяет получить 98.28% наилучшей

производительности из-за задержки доступа к памяти, которое может отличаться от среднего значения, используемого в ГП по умолчанию (см. раздел 2.1).

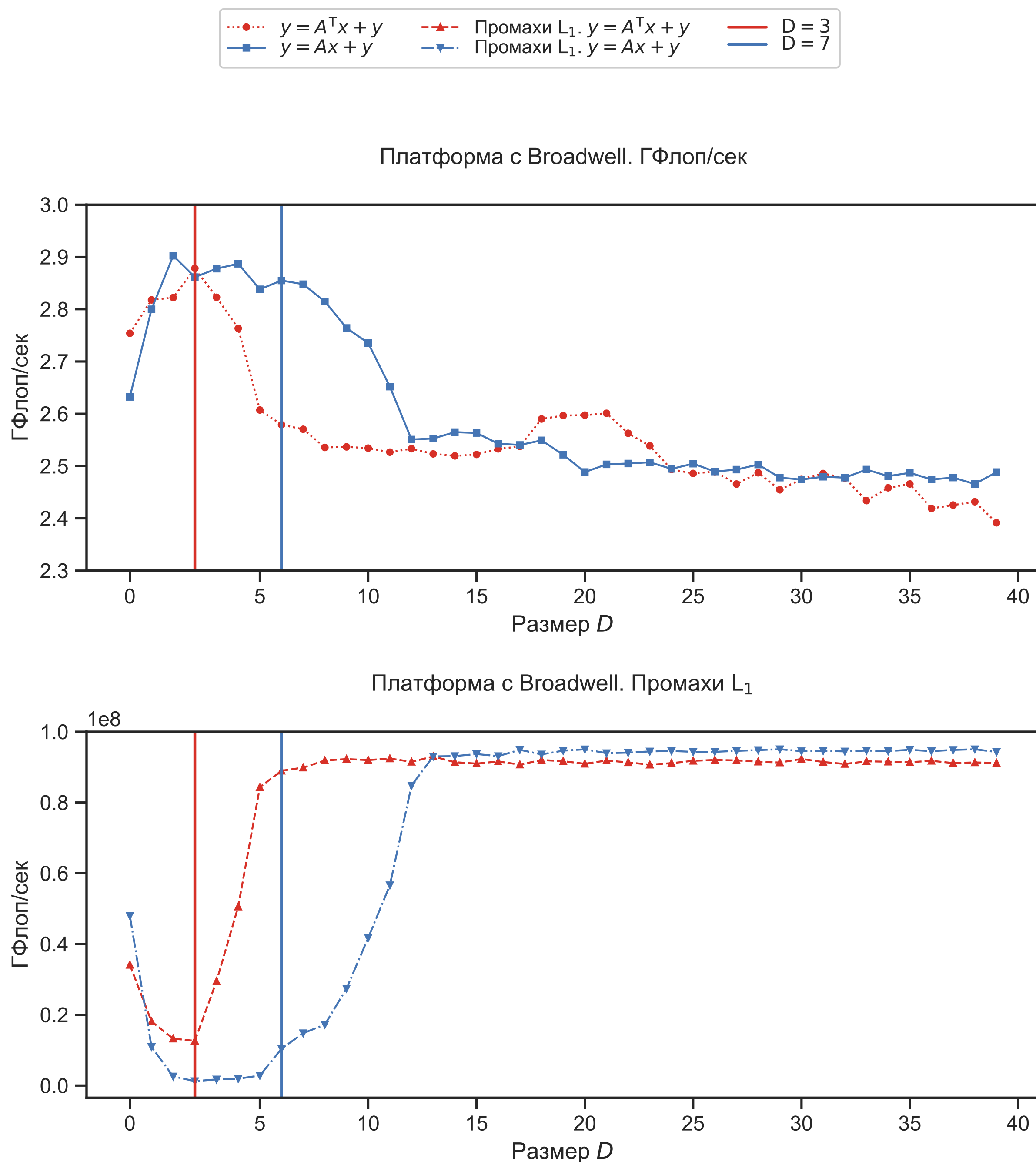


Рис. 4.26. Оценка влияния значений параметра D на производительность на Broadwell.

4.3.5. Оценка значений параметра M_c

Рис. 4.27 и рис. 4.28 содержат значение однопоточной производительности и количество промахов L_1 и L_2 для $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно. Оцениваемые значения параметра M_c изменялись от 1 до 80 с шагом 1.

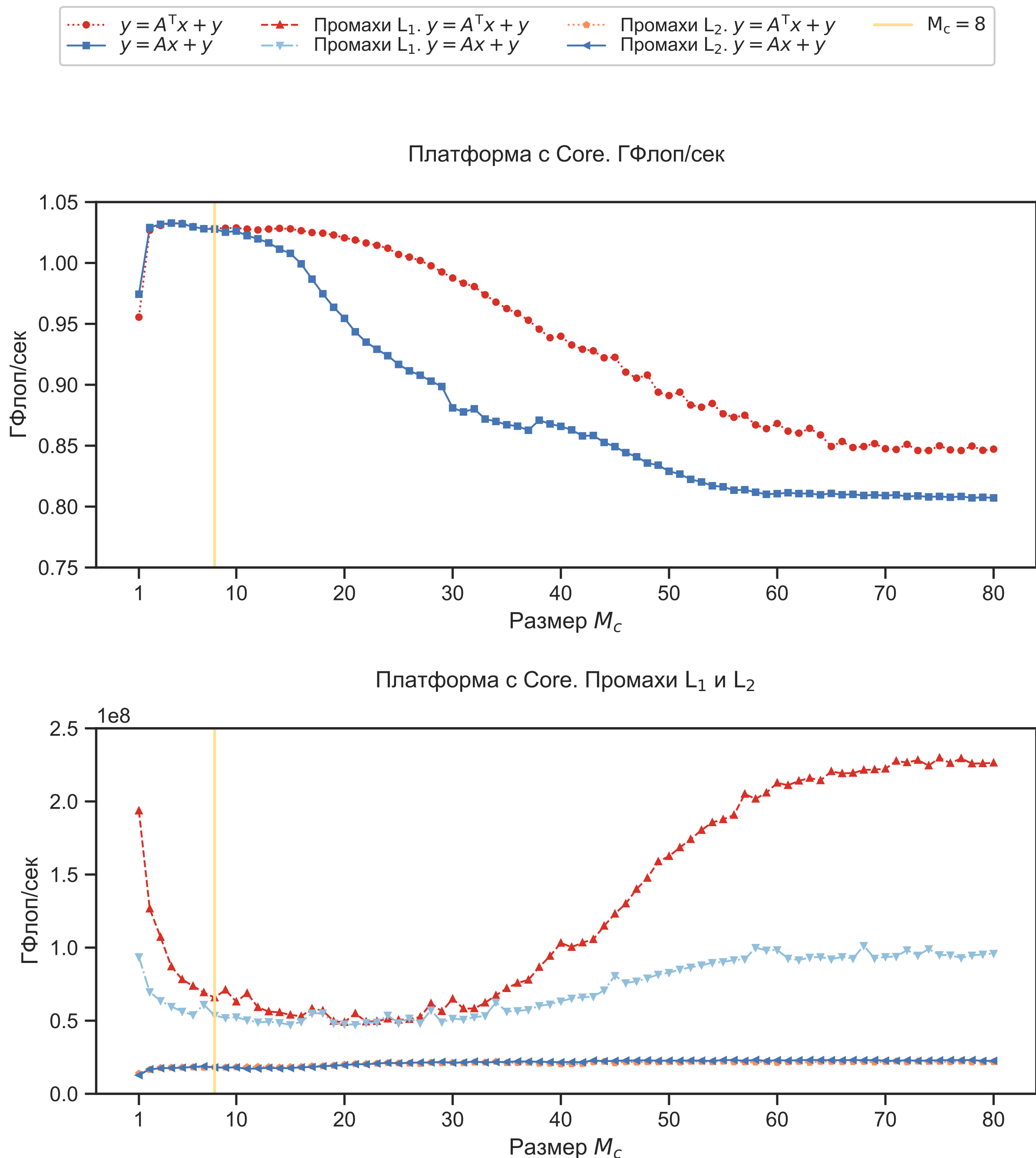


Рис. 4.27. Оценка влияния значений параметра M_c на производительность на Core.

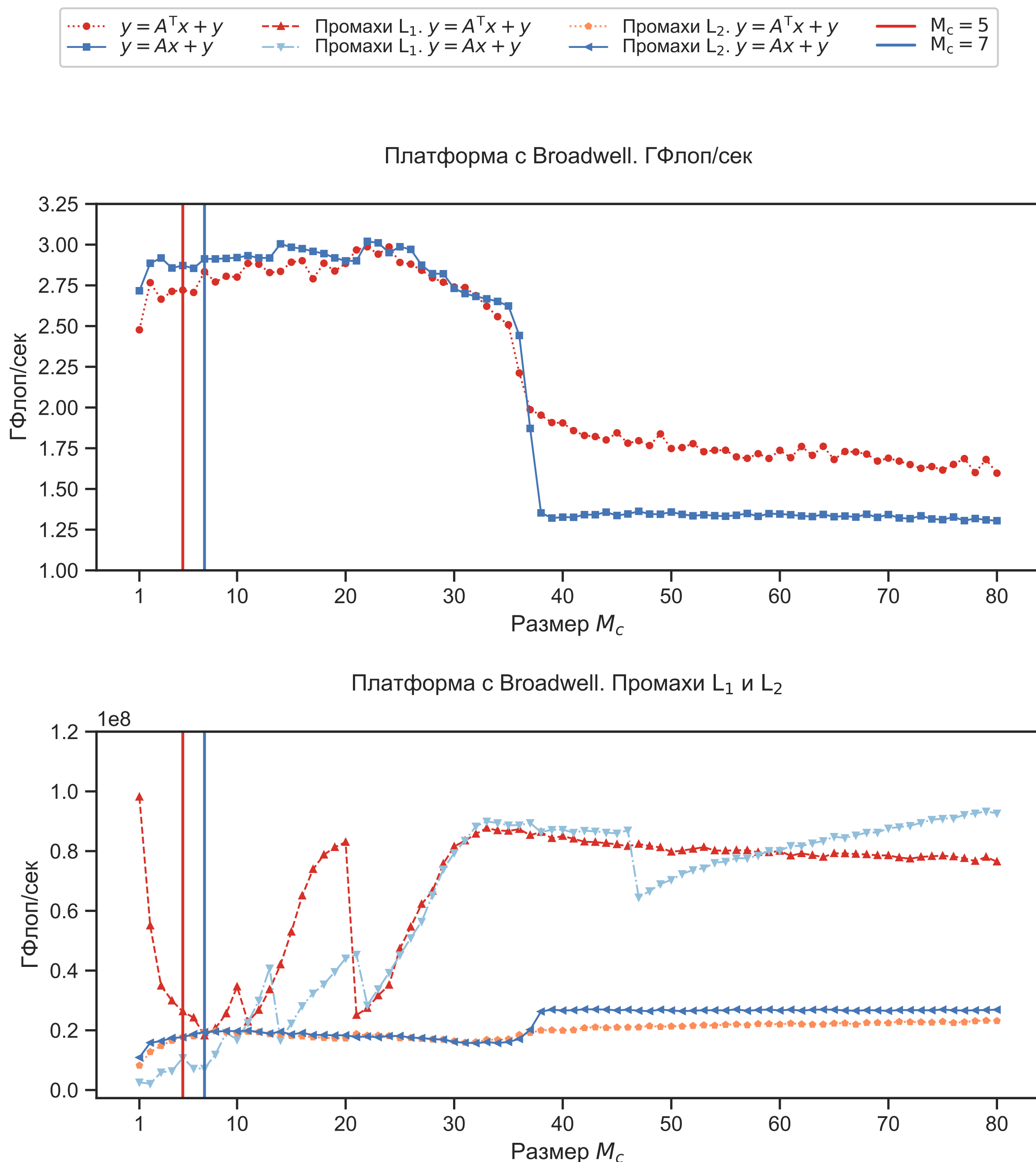


Рис. 4.28. Оценка влияния значений параметра M_c на производительность на Broadwell.

В случае Core и $y = A^{\mathsf{T}}x + y$ производительность возрастает от 0.96 ГФлоп/сек до 1.03 ГФлоп/сек с увеличением значений параметра M_c от 1 до 4 и уменьшается до 0.85 ГФлоп/сек с дальнейшим увеличением M_c. Рост производительности вызван количеством повторных использований элементов вектора y между двумя итерациями цикла с индуктивной пере-

менной i_b алгоритма 7 и количеством элементов вектора x, хранящихся в L_1. Вследствие этого количество промахов L_1 уменьшается от 193665807 до 49003837 с увеличением значений параметра M_c от 1 до 20. Увеличение значений параметра M_c приводит к вытеснению элементов матрицы A и вектора x из L_1 в L_2, что вызывает рост количества промахов L_1 от 49003837 до 226432208. Вытесненные элементы матрицы A и вектора x вытесняют элементы вектора y, расположенные в L_2 и используемые каждой итерацией цикла с индуктивной переменной i алгоритма 7. Вследствие этого промахи L_2 увеличиваются от 13845617 до 22117224 с увеличением значений параметра M_c от 1 до 54 и остаются примерно равными 22117224 с дальнейшим ростом M_c. В случае $y = Ax + y$ наблюдается схожая зависимость между изменением производительности и количеством промахов L_1 и L_2. Отличие состоит в том, что количество повторно используемых элементов y увеличивается с ростом значений параметра M_c. Можно сделать вывод о том, что используемые формулы позволяют получить неоптимальное значение параметра $M_c = 8$, для которого достигается 99.6% и 99.5% значений наилучшей производительности в случаях $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно. Причина в отсутствии L_3, который может сохранить вытесненные из L_2 элементы вектора y.

В случае Broadwell и $y = A^{\mathsf{T}}x + y$ производительность возрастает от 2.48 ГФлоп/сек до 2.99 ГФлоп/сек с увеличением значений параметра M_c от 1 до 22 и уменьшается до 1.6 ГФлоп/сек с дальнейшим ростом параметра M_c. В отличие от Core количество промахов L_2 увеличивается от 8228257 до 23162115 с ростом значений параметра M_c. Размер L_2 у целевой аппаратной платформы с Broadwell меньше чем у Core. Это приводит к росту промахов L_2 с увеличением значений параметра M_c. Количество промахов L_1 уменьшается от 98158369 до 18201677 с увеличением M_c от 1 до 7 и возрастает до 18201677 с дальнейшем увеличением значений параметра M_c. В отличие от Core количество промахов L_1 начинает возрастать

для меньших значений параметра M_c. Такое поведение вызвано тем, что значения параметров N_b и D больше чем у Core. Рост производительности вызван количеством повторных использований элементов вектора y между двумя итерациями цикла с индуктивной переменной i_b алгоритма 7. В отличие от целевой аппаратной платформы с Core уровень кэша L_3 у Broadwell хранит некоторые элементы вектора y, вытесненные из L_2. Похожее поведение наблюдается и для $y = Ax + y$. Однако, количество промахов L_1 увеличивается от 2592027 до 92640947 с увеличением значений параметра M_c от 1 до 80. Причина в том, что L_1 может хранить элементы вектора x между двумя последовательными итерациями цикла с индуктивной переменной i_c для $N_c = 4096$ и меньших значений параметра M_c. Можно сделать вывод о том, что используемые формулы вычисляют неоптимальные значения $M_c = 7$ и $M_c = 5$ для которых достигается 94.6% и 95.6% наилучшей производительности для случаев $y = A^\mathsf{T}x + y$ и $y = Ax + y$, соответственно.

Подход автора направлен на сокращение количества промахов L_2 с целью сокращения колебаний производительности и достижения лучших результатов для операндов MVM, которые могут полностью хранится в L_3. В качестве примера рассмотрим рис. 4.29, содержащий однопоточную производительность для $y = A^\mathsf{T}x + y$ и Broadwell. Рассматривались плотные квадратные матрицы. Матрицы и векторы содержали элементы типа double. Размеры матриц изменялись от 1024 до 25600 с шагом 64. Можно сделать вывод о том, что для $M_c = 22$ в среднем достигалось 1.04-кратное ускорение по сравнению с подходом автора. Однако в этом случае достигалась более высокая производительность для размерностей задачи меньшей чем $M = N = 1982$. В случае $M_c = 22$ отмечается ухудшение производительности и рост количества промахов L_2 для размерностей задачи кратных 2048.

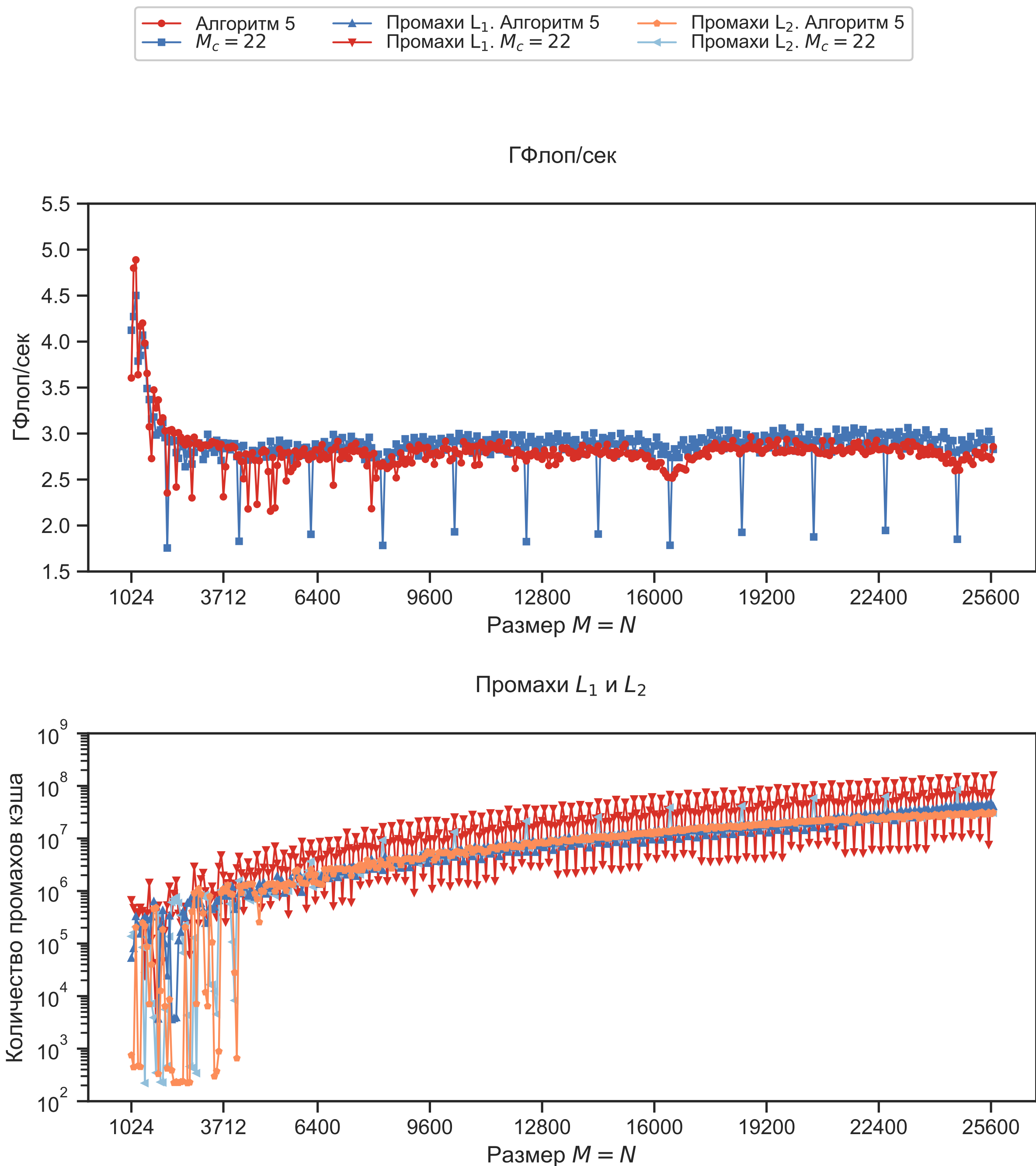


Рис. 4.29. Оценка влияния значений параметра M_c на производительность на Broadwell.

4.3.6. Оценка значений параметра N_c

Рис. 4.30 и рис. 4.31 содержат значение однопоточной производительности и количество промахов L_1 и L_2 для $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно. Оцениваемые значения параметра N_c изменялись от 64 до 19168 с шагом 32.

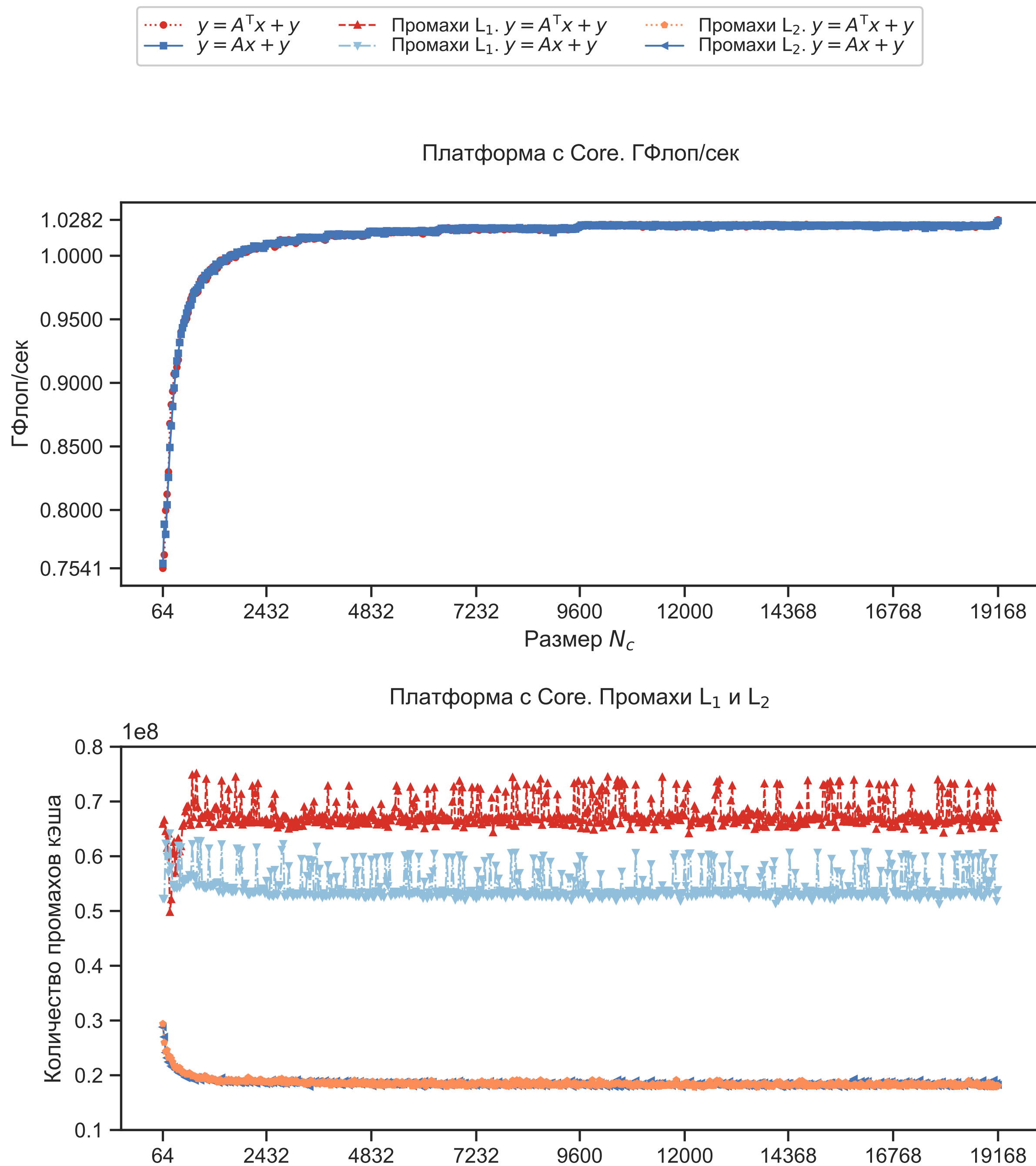


Рис. 4.30. Оценка влияния значений параметра N_c на производительность на Core.

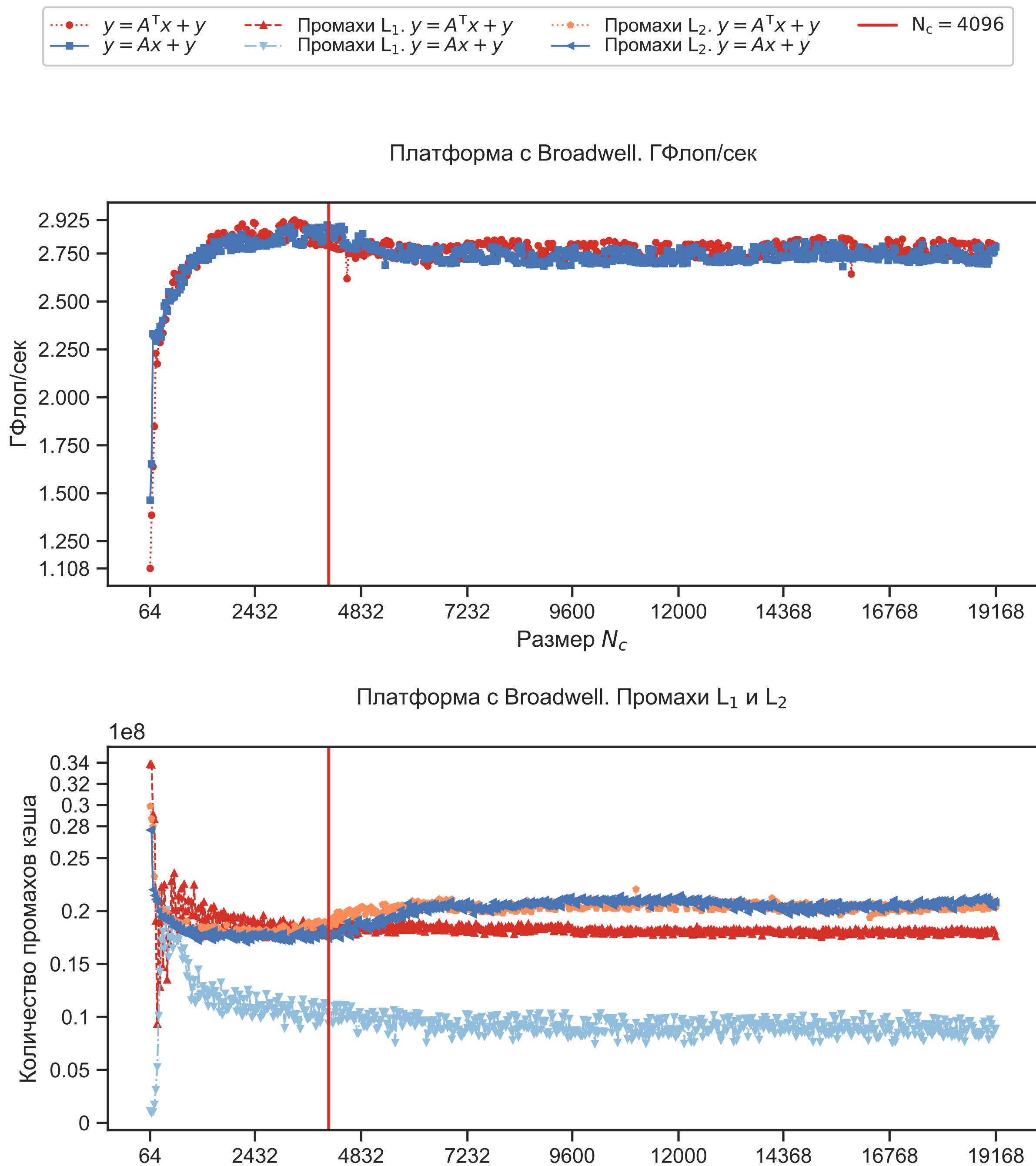


Рис. 4.31. Оценка влияния значений параметра N_c на производительность на Broadwell.

В случае Core и $y = A^{\mathsf{T}}x + y$ количество промахов L_1 остается примерно 65579898 для всех рассмотренных значений параметра N_c. Количество промахов L_2 уменьшается от 29409371 до 17844660 с увеличением значений параметра N_c от 64 до 1344 и остается примерно равным 17844660 для больших значений N_c. Отмечается постоянный рост производительности с увеличением количества повторных использований элементов вектора x и увеличением размера подвектора вектора y, располагающегося в L_2. Из-за

физической индексируемости L_2 некоторые элементы подвектора вектора y вытесняются из L_2, несмотря на значения M_c. Вследствие этого количество промахов L_2 перестает уменьшаться. Похожее поведение наблюдается в случае $y = Ax + y$. Отличие заключается в том, что количество повторных использований элементов y увеличивается, а количество повторных использований элементов x остается неизменным. В обоих случаях наилучшее значение N_c достигается для 32768. Это совпадает с результатом, вычисленным использованной функцией.

В случае Broadwell и $y = A^{\mathsf{T}}x + y$ производительность возрастает от 1.1 ГФлоп/сек до 2.92 ГФлоп/сек с увеличением значений параметра N_c от 64 до 3392 и уменьшается до 2.79 ГФлоп/сек с дальнейшим ростом значений параметра N_c. Как и в случае Core рост производительности вызван увеличением количества повторных использований элементов вектора x и увеличением размера подвектора вектора y, располагающегося в L_2. Это подтверждается уменьшением количества промахов L_2 от 48907412 до 17607323 с увеличением значений параметра N_c от 64 до 2624. У Broadwell размер кэш-уровня L_2 меньше по сравнению с размером такого же кэш-уровня Core. Вследствие этого количество промахов L_2 увеличивается до 20753599 с ростом значений параметра N_c, вызывая ухудшение производительности. Количество промахов L_1 уменьшается от 88488415 до 19105038 с увеличением значений параметра N_c от 64 до 256 и остается примерно на уровне 19105038. В случае Broadwell используемые значения параметров N_b и D больше чем у Core. Должны быть выполнены первые D итераций цикла с индуктивной переменной j_b алгоритма 7, чтобы можно было начать загружать элементы матрицы A из кэш-памяти. Так как элементы вектора x используются каждую итерацию цикла с индуктивной переменной j алгоритма 7 и эти элементы не загружаются в кэш-память вручную, значения параметра N_c обратно пропорционально количеству промахов L_1. Аналогичное наблюдение можно сделать в случае $y = Ax + y$. Отличие в

том, что количество промахов L_1 увеличивается от 1127089 до 18093749 с ростом значений параметра N_c от 64 до 352 и остается примерно равно 18093749. Увеличение количества промахов кэша вызвано неэффективностью предвыборки элементов для значений параметра N_c меньших чем $D4\mathbb{C}_{L_1}$ в случае Broadwell. Можно сделать вывод о том, что формулы определяют значение $N_c = 4096$, для которого достигается 98.3% и 98.6% наилучшей производительности в случаях $y = A^{\mathsf{T}}x + y$ и $y = Ax + y$, соответственно. Так как L_2 физически индексируем, предположения, на которых основываются формулы для вычисления наилучших значений параметров, являются грубой оценкой.

4.4. Оценка алгоритма сокращения времени выполнения сверток тензоров

Рассмотрим примеры ТС из библиотеки, представленной в работе [17]. Используемая библиотека содержит ТС разной размерности, применяемые в методах связанных кластеров [61] и квантовой химии [175]. Оценим однопоточную и многопоточную производительности ПС АОТО, современных компиляторов и фреймворков, позволяющих получить оптимизированную ТС (TCCG [17], TBLIS [16]). Рассматривались тензоры с типом элементов double.

Для экспериментов использовалась аппаратная платформа с процессором Intel Sandy Bridge (табл. 4.2). Табл. 4.3 содержит информацию о версии и дополнительных опциях используемого программного обеспечения. В случае компилятора ICC использовалась опция -qopt-matmul, чтобы обеспечить распознавание и замену МММ на вызов реализации МММ доступной в библиотеке Intel MKL (см. табл. 4.3). Обозначим ТС вида $\mathrm{C}_{\pi_C(IJ)} = \sum_P \mathrm{A}_{\pi_A(IP)} \cdot \mathrm{B}_{\pi_B(PJ)}$ как $\pi_C(IJ)$-$\pi_A(IP)$-$\pi_B(PJ)$. Тогда, например, $\mathrm{C}_{ijkl} = \sum_{m=0}^{n_m-1} \sum_{n=0}^{n_n-1} \mathrm{A}_{imjn} \cdot \mathrm{B}_{nlmk}$ может быть обозначена как ijkl-imjn-nlmk.

4.4.1. Однопоточная производительность

На рис. 4.32 представлена однопоточная производительность для сверток вида abcde-efbad-cf и abcd-eafc-bdfe. Табл. 4.28 содержит время вычисления в секундах для abcde-efbad-cf. Результаты всех примеров TC из рассматриваемой библиотеки приведены в табл. 4.29. В случае abcde-efbad-cf использовались размерности $n_a = n_b = 8$ и $n_d = n_e = 4$. Размерность n_c приравнивалась к n_f и изменялась от 4 до 1024 с шагом 4. В случае abcd-eafc-bdfe использовались равные размерности, изменявшиеся от 4 до 64 с шагом 4. Рис. 4.32 также содержит результаты для MMM[×, +], реализованной в библиотеке BLIS, для матриц эквивалентной размерности.

Таблица 4.28. Время вычисления TC вида abcde-efbad-cf в секундах, где $n_a = n_b = 8$ и $n_d = n_e = 4$.

Реализация \ $n_c = n_f$	256	512	768	1024
AOTO	0.006	0.023	0.053	0.089
BLIS	0.0057	0.022	0.048	0.078
TBLIS	0.006	0.024	0.054	0.087
TCCG	0.0079	0.026	0.055	0.09
polly	0.032	0.13	0.29	0.5
icc	0.085	0.35	0.82	1.87
gcc	0.063	0.34	0.84	2.001
clang	0.065	0.343	0.85	2.14

Эксперименты показали 84-кратное ускорение ПС AOTO по сравнению с компилятором ICC и 82-кратное ускорение по сравнению с GCC и CLang. Отметим, что распознавание и замена MMM на вызов реализации MMM доступной в библиотеке Intel MKL не повлияло на производительность компилятора ICC. Согласно полученным данным для тензоров, содержащих менее 32^2 элементов, целесообразно использовать другие подходы к сокращению времени выполнения MMA[×, +]. Это соответствует результатам, представленным в работе [52]. В случае фреймворков TCCG и TBLIS программная система AOTO достигает более 86.12% их производительности. Несмотря на использование подхода TBLIS, ПС AOTO не

всегда достигает производительности TBLIS (см. раздел 2.4). Как было показано ранее, генерируемые ПС AOTO части кода MMM, содержащие векторные инструкции, уступают в производительности коду, оптимизированному вручную, из-за проблем автоматической векторизации.

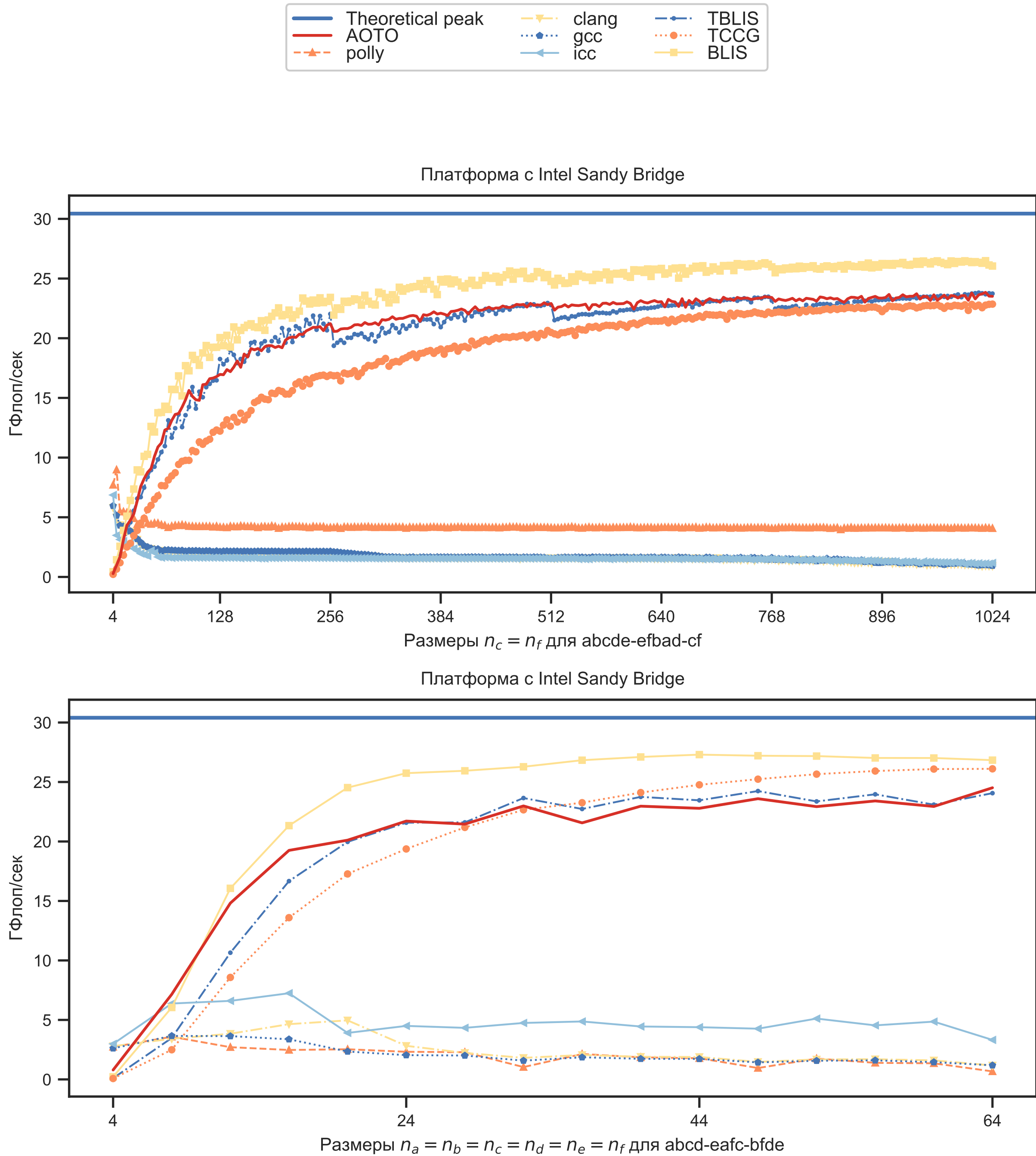


Рис. 4.32. TC для типа double.

Таблица 4.29. Весь набор тестов TC для типа double, * — наилучшее значения производительности для GCC и Clang

TC	Размеры	* (%)	icc (%)	polly (%)	AOTO (%)	tblis (%)	tccg (%)
ab-ac-cb	1024x1024-1024x1024-1024x1024	1.05	44.84	5.69	74.21	77.27	**85.8**
ab-acd-dbc	1024x1024-1024x32x32-32x1024x32	1.12	2.5	1.09	72.57	77.2	**81.8**
abc-acd-db	32x1024x32-32x32x1024-1024x1024	1.05	7.4	12.07	77.2	78.52	**82.9**
abc-ad-bdc	1024x32x32-1024x1024-32x1024x32	5	13.55	26.61	74.8	77.93	**82.4**
abc-adc-bd	32x1024x32-32x1024x32-1024x1024	5.53	8.36	32.93	77.76	**79.18**	78.6
ab-cad-dcb	1024x1024-32x1024x32-32x32x1024	1.02	17.07	0.99	75.07	77.14	**81.3**
abc-adec-ebd	32x1024x32-32x32x32x32-32x1024x32	1.05	17.76	1.05	**78.75**	78.07	74.6
abc-adc-db	32x1024x32-32x1024x32-1024x1024	0.92	25.1	30.79	76.48	**78.62**	78.5
abc-bda-dc	32x32x1024-32x1024x32-1024x1024	0.92	32.07	13.91	74.77	76.22	**82.5**
abcd-aebf-dfce	32x32x32x32-32x32x32x32-32x32x32x32	1.05	1.09	5.07	72.07	76.15	**78.7**
abcd-aebf-fdec	32x32x32x32-32x32x32x32-32x32x32x32	1.02	0.99	0.99	72.5	72.34	**79**
abcd-aecf-bfde	32x32x32x32-32x32x32x32-32x32x32x32	6.61	6.78	5.89	76.64	**78.06**	75.4
abcd-aecf-fbed	32x32x32x32-32x32x32x32-32x32x32x32	4.11	14.74	1.05	74.54	**78.29**	74.9
abcd-aedf-bfce	32x32x32x32-32x32x32x32-32x32x32x32	16.94	13.45	6.02	77.73	**77.83**	75.6
abcd-aedf-fbec	32x32x32x32-32x32x32x32-32x32x32x32	11.81	15.79	1.05	76.78	**78.12**	75.4
abcd-aefb-fdce	32x32x32x32-32x32x32x32-32x32x32x32	0.99	5	1.02	70.56	72.39	**78.2**
abcd-aefc-fbed	32x32x32x32-32x32x32x32-32x32x32x32	16.97	26.05	2.76	78.22	**78.39**	75.1
abc-dca-bd	32x1024x32-1024x32x32-1024x1024	1.02	15.66	6.25	76.41	78.06	**82**
abcd-dbea-ec	16x8x1024x8-8x8x1024x16-1024x1024	1.78	3.13	9.11	**77.89**	77.4	77.8
abcd-deca-be	16x1024x8x8-8x1024x8x16-1024x1024	1.97	6.74	5.36	77.17	77.11	**78.9**
abcd-ea-ebcd	1024x16x8x8-1024x1024-1024x16x8x8	3.68	10.92	18.29	74.11	77.17	**86.1**
abcd-eafb-fdec	32x32x32x32-32x32x32x32-32x32x32x32	1.02	5.26	0.89	72.04	71.98	**77.3**
abcd-eafc-bfde	32x32x32x32-32x32x32x32-32x32x32x32	5.95	14.74	3.42	67.5	**77.57**	73.9
abcd-eafd-fbec	32x32x32x32-32x32x32x32-32x32x32x32	12.89	26.61	0.92	**78.22**	78.19	72.9
abcd-ebad-ce	16x8x1024x8-1024x8x16x8-1024x1024	5.16	1.71	22.86	80.33	79.21	**82.2**
abcd-eb-aecd	16x1024x8x8-1024x1024-16x1024x8x8	4.21	2.7	26.25	76.88	**78.68**	78.4
abcd-ec-abed	16x8x1024x8-1024x1024-16x8x1024x8	5	1.25	30.59	**79.47**	79.08	77
abcde-ecbfa-fd	8x8x4x1024x4-4x4x8x1024x8-1024x1024	14.7	2.8	14.41	**78.52**	73.26	75.6
abcde-efbad-cf	8x8x1024x4x4-4x1024x8x8x4-1024x1024	3.29	0.95	13.39	**80.3**	78.39	75.6
abcde-efcad-bf	8x1024x8x4x4-4x1024x8x8x4-1024x1024	3.29	3.45	13.29	**78.88**	78.28	76.9
abcdef-dega-gfbc	16x16x8x8x8x8-8x8x1024x16-1024x8x16x8	2.76	1.78	5.1	74.7	76.55	**76.8**
abcdef-degb-gfac	16x16x8x8x8x8-8x8x1024x16-1024x8x16x8	2.7	1.74	5.3	74.61	76.48	**77.6**
abcdef-degc-gfab	16x16x8x8x8x8-8x8x1024x8-1024x8x16x16	4.84	2.66	3.95	70.76	72.27	**78.9**
abcdef-dfga-gebc	16x16x8x8x8x8-8x8x1024x16-1024x8x16x8	3.06	3.62	7.14	71.94	76.25	**76.6**
abcdef-dfgb-geac	16x16x8x8x8x8-8x8x1024x16-1024x8x16x8	2.99	4.05	7.07	72.83	76.28	**77.9**
abcdef-dfgc-geab	16x16x8x8x8x8-8x8x1024x8-1024x8x16x16	3.88	3.72	6.32	72.47	71.94	**77.6**
abcdef-efga-gdbc	16x16x8x8x8x8-8x8x1024x16-1024x8x16x8	1.02	3.52	4.47	72.07	76.25	**76.9**
abcdef-efgb-gdac	16x16x8x8x8x8-8x8x1024x16-1024x8x16x16	1.02	10.49	4.47	72.5	**76.44**	76.4
abcdef-efgc-gdab	16x16x8x8x8x8-8x8x1024x8-1024x8x16x16	1.02	3.65	4.74	71.55	71.38	**77.1**
abcdef-gdab-efgc	16x16x8x8x8x8-1024x8x16x16-8x8x1024x8	0.99	3.62	4.38	71.45	71.61	**76.8**
abcdef-gdac-efgb	16x16x8x8x8x8-1024x8x16x8-8x8x1024x16	0.99	10.49	3.49	72.43	76.02	**76.6**
abcdef-gdbc-efga	16x16x8x8x8x8-1024x8x16x8-8x8x1024x16	1.02	03.49	4.7	70.33	76.25	**77.1**
abcdef-geab-dfgc	16x16x8x8x8x8-1024x8x16x16-8x8x1024x8	3.78	3.72	6.12	72.37	71.61	**78.3**
abcdef-geac-dfgb	16x16x8x8x8x8-1024x8x16x8-8x8x1024x16	3.16	4.05	6.88	73.06	75.33	**78.3**
abcdef-gebc-dfga	16x16x8x8x8x8-1024x8x16x8-8x8x1024x16	2.99	3.62	6.81	71.97	76.15	**76.9**
abcdef-gfab-degc	16x16x8x8x8x8-1024x8x16x16-8x8x1024x8	4.47	2.7	3.88	70.79	72.6	**78.9**
abcdef-gfac-degb	16x16x8x8x8x8-1024x8x16x8-8x8x1024x16	2.76	1.74	4.87	74.9	77.04	**77.2**
abcdef-gfbc-dega	16x16x8x8x8x8-1024x8x16x8-8x8x1024x16	2.86	1.78	4.8	74.18	**76.68**	76.2

В некоторых случаях ПС AOTO достигает производительности TCCG и TBLIS. В случае TCCG это объясняется недостатками методов TTGT и

LoG, используемых этим фреймоворком (см. раздел 2.4). Производительность LoG зависит от размерности тензоров и их расположения в памяти. TTGT выполняет перестановку элементов тензоров с целью их представления в виде матриц и использования MMM. Для выполнения таких перестановок требуются дополнительное время и память. ПС AOTO достигает производительности TBLIS из-за отличия в применяемых эвристиках. В частности, TBLIS определяет операнд TC, последовательное обращение к которому может быть выполнено одновременно с последовательным обращением к вычисляемому тензору. Так, например, в случае abcde-fbade-cf таким операндом будет тензор fbade. Выбранный тензор сводится к операнду MMM, элементы которого используются чаще чем элементы второго операнда MMM (см. раздел 1). Если такого операнда не обнаружено, TBLIS выбирает первый из операндов TC. Вследствие этого в случае abcde-efbad-cf будет выбран тензор efbad. ПС AOTO использует такой же подход, но по умолчанию выбирает второй операнд TC вместо первого. Как показывает рис 4.32, выбор cf в случае abcde-efbad-cf позволяет ПС AOTO выполнить TC с меньшими издержками обращения к памяти и достичь однопоточную производительность TBLIS, несмотря на неоптимальность генерируемого кода (см. раздел 4.2.1).

4.4.2. Многопоточная производительность

Рассмотрим многопоточную производительность методов TBLIS, TTGT и ПС AOTO. Для выполнения эксперимента использовалась платформа с 4-ядерным процессором Intel Sandy Bridge (табл. 4.2). Оценивались фреймворок TBLIS, ПС AOTO и реализация метода TTGT доступная в фреймворке TCCG.

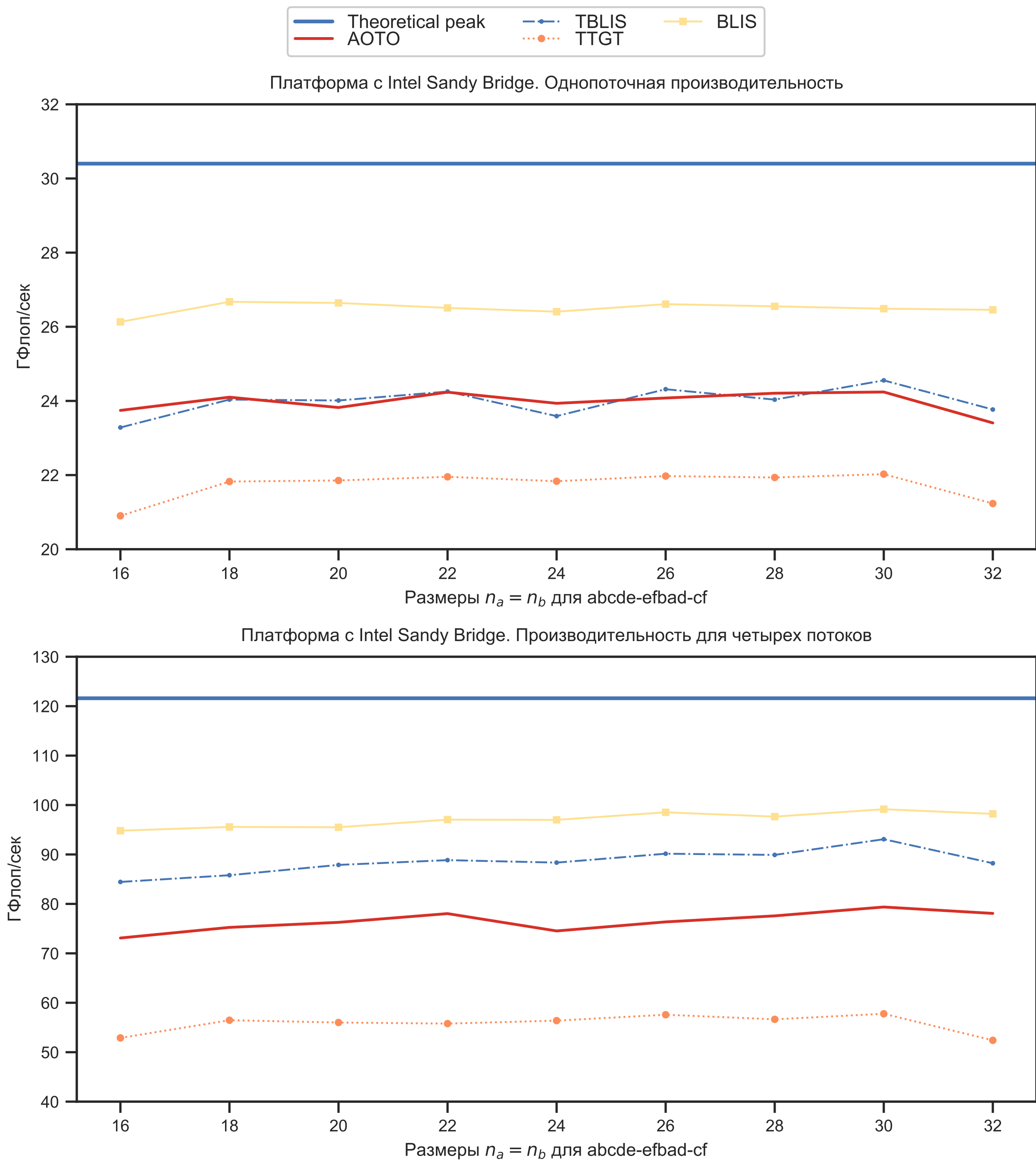


Рис. 4.33. TC для типа double.

Рис. 4.33 содержит графики однопоточной и многопоточной производительностей для сверток вида abcde-efbad-cf. Табл. 4.30 и табл. 4.31 содержат время вычисления в секундах для одного и четырех потоков, соответственно. Размерность n_a приравнивалась к n_b и изменялась от 16 до 32 с шагом 2. Размерности остальных измерений были фиксированы и равны $n_d = n_e = 2$ и $n_c = n_f = 1024$.

Таблица 4.30. Время вычисления TC вида abcde-efbad-cf в секундах для одного потока, где $n_d = n_e = 2$ и $n_c = n_f = 1024$.

Реализация \ $n_a = n_b$	16	18	20	22	24	26	28	30	32
AOTO	0.09	0.113	0.141	0.168	0.202	0.236	0.272	0.311	0.367
TTGT	0.103	0.125	0.154	0.185	0.22	0.258	0.299	0.343	0.405
TBLIS	0.092	0.113	0.139	0.167	0.205	0.233	0.274	0.307	0.361
BLIS	0.082	0.102	0.126	0.153	0.183	0.213	0.248	0.285	0.325

Таблица 4.31. Время вычисления TC вида abcde-efbad-cf в секундах для четырех потоков, где $n_d = n_e = 2$ и $n_c = n_f = 1024$.

Реализация \ $n_a = n_b$	16	18	20	22	24	26	28	30	32
AOTO	0.029	0.036	0.044	0.052	0.065	0.074	0.085	0.095	0.11
TTGT	0.041	0.048	0.059	0.073	0.086	0.098	0.116	0.131	0.164
TBLIS	0.025	0.032	0.038	0.046	0.055	0.063	0.073	0.081	0.097
BLIS	0.023	0.028	0.035	0.042	0.049	0.058	0.067	0.076	0.087

Оценим влияние издержек, создаваемых перестановками элементов тензоров, на производительность метода TTGT в зависимости от размерности тензоров и количества потоков. Метод TTGT использует библиотеку HPTT (High-Performance Tensor Transpose) [176] для перестановки элементов тензоров с целью сведения TC к MMM (см. раздел 1). Для оценки полученных издержек в TTGT использовалась реализация MMM доступная в библиотеке BLIS. Многопоточная и однопоточная производительности соответствующих MMM приведены на рис. 4.33, табл. 4.30 и табл. 4.31. Согласно полученным данным, TTGT достигает 81.81% и 57.53% однопоточной и многопоточной производительности BLIS для TC вида abcde-efbad-cf, соответственно. Можно сделать вывод о том, что с увеличением размерности тензоров и увеличением времени выполнения перестановок их элементов, производительность TTGT уменьшается до 79.61% и 53.04% однопоточной и многопоточной производительностей BLIS, соответственно. Это объясняется увеличением времени выполнения MMM, полученной сведением TC к MMM. Отметим, что несмотря на то, что библиотекой HPTT поддерживается многопоточное выполнение, в случае использова-

ния нескольких потоков, стоимость выполнения перестановки элементов тензоров увеличивается в случае использования четырех потоков.

Метод TBLIS использует части реализаций MMM библиотеки BLIS, содержащие векторные инструкции. В отличие от TTGT метод TBLIS не выполняет перестановки элементов тензоров с целью представления в виде матриц и применения реализации MMM (см раздел 1). Это позволяет TBLIS получить от 89.13% до 92.83% однопоточной производительности BLIS и от 89.69% до 93.82% многопоточной производительности BLIS. ПС AOTO достигает однопоточную производительность TBLIS в силу неоптимальности эвристики, применяемой TBLIS для доступа к элементам тензоров (см. раздел 4.4). В случае четырех потоков ПС AOTO достигает только 88.18% производительности TBLIS из-за проблем используемой автоматической векторизации (см раздел 4.2.1).

4.5. Выводы

В этой главе выполнена оценка эффективности ПС AOTO и представленных в предыдущих главах алгоритмов для сокращения времени выполнения TC, обобщенного MMM и MVM. Результаты этой главы опубликованы в работах [159–161].

В качестве примера приложения алгоритмов рассмотрено ускорение реализации решения задачи гравиметрии на основе предложенного автором алгоритма 8 вычисления MVM с реализациями на основе кода оптимизированных библиотек. Эксперименты показали, что в среднем результаты предложенного алгоритма отличаются не более чем на 1% и превосходят библиотеки OpenBLAS и BLIS.

Выполнено сравнение однопоточной и многопоточной производительностей реализации предложенных автором алгоритмов 7 и 8 с библиотеками, содержащими реализации MVM. В результате экспериментов уста-

новлено, что в среднем достигается однопоточная и многопоточная производительности библиотек Intel MKL, OpenBLAS и BLIS. Вследствие этого целесообразна модификация алгоритма 9 для получения реализаций MVM, описанных алгоритмами 7 и 8 (см. раздел 2.4).

Выполнено сравнение однопоточной и многопоточной производительностей ПС АОТО с библиотеками, содержащими оптимизированную реализацию MMM, и современными компиляторами. В случае однопоточной производительности в результате экспериментов установлено, что ПС АОТО достигает 1.63-кратного ускорения по сравнению с компилятором ICC [63]; 20-кратного ускорения по сравнению с фронтендом для компиляции Clang [64] и компиляторами GCC [65], IBM XLC [66]; 83.33% производительности рассмотренных библиотек Intel MKL, ARMPL, OpenBLAS и BLIS. Установлено, что ПС АОТО достигает 86.18% многопоточной производительности рассмотренных библиотек Intel MKL, OpenBLAS и BLIS. Как было показано в разделе 4.2.1, устранение проблем векторизации, выполняемой библиотекой LLVM Core в ПС АОТО, позволяет достичь производительности библиотек Intel MKL, OpenBLAS и BLIS.

Несмотря на то, что в случае MMM описанная реализация ПС АОТО не позволяет достичь производительности рассмотренных библиотек, предложенный метод может быть использован для создания временных реализаций MMM для новых целевых аппаратных платформ без доступа к ним. Полученная реализация MMM, совместно с реализациями MVM на основе алгоритмов 7 и 8, может использоваться в качестве основы для реализации всех операций третьего и второго уровней интерфейса BLAS, соответственно (см. раздел 1). Для получения временных реализаций операций первого уровня BLAS в ПС АОТО используются оптимизации, применяемые библиотекой LLVM Core и фреймворком Polly по умолчанию. Вследствие этого создание бэкендов, поддерживающих генерацию ПС АОТО машинного кода для архитектур отечественных процессоров, и устранение проблем

векторизации, выполняемой библиотекой LLVM Core в ПС AOTO, может облегчить создание высокопроизводительных реализаций BLAS для отечественных целевых аппаратных платформ.

Выполнено сравнение однопоточной производительности ПС AOTO с современными компиляторами для решения задач из класса APP, в случае которых неприменимы реализации интерфейса BLAS. Эксперименты показали, что ПС AOTO достигает 85% верхней теоретической границы производительности. Для всех рассматриваемых задач ПС AOTO достигла 1.56-кратного ускорения по сравнению с компилятором ICC.

Выполнено сравнение однопоточной производительности ПС AOTO для случая TC, оптимизация которых с помощью операций BLAS затруднительна из-за возникающих издержек [16, 17]. Для сравнения рассматривались фреймворки, позволяющие получить оптимизированную реализацию TC, и современные компиляторы. Эксперименты показали, что ПС AOTO достигает 80-кратного ускорения по сравнению с рассмотренным фронтендом для компиляции Clang и компиляторами GCC, ICC; 86.12% однопоточной производительности рассмотренных фреймворков TCCG и TBLIS.

Заключение

Сформулируем основные преимущества программной системы AOTO.

1. Использует формулы, позволяющие автоматически получить значения параметров алгоритмов выполнения тензорных операций в зависимости от характеристик многоядерных процессоров общего назначения.

2. Программная система AOTO автоматически оптимизирует время выполнения свертки тензоров, не описываемой интерфейсом BLAS. Интерфейс BLAS неприменим для реализации новых методов оптимизации времени выполнения свертки тензоров, реализованных в программной системе AOTO, а также в фреймворках TCCG и TBLIS, поддерживающих малое количество архитектур.

3. Программная система AOTO автоматически оптимизирует время выполнения обобщения матричного произведения на замкнутые полукольца с элементами из множества вещественных чисел с целью оптимизации решений общей задачи о путях для соответствующего типа полуколец. Указанные обобщения не описываются интерфейсом BLAS и не содержатся во всех известных его реализациях. Новый интерфейс GraphBLAS, описывающий обобщения матричного произведения, реализован для малого количества архитектур.

4. Программная система AOTO применяется для получения высокопроизводительных реализаций матричного и матрично-векторного произведений для различных архитектур и типов данных. Программная система AOTO позволяет получить временные реализации для архитектур с отсутствующими специализированными библиотеками. Например, в случае процессоров от AMD не существует специализированной библиотеки, реализующей интерфейс BLAS. В отличие от ARMPL и Intel MKL библиотеки, применяемые для процессоров AMD, не содержат реализации матричного произведения для целочисленных типов данных, не описываемых интерфейсом BLAS, но широко используемых в машинном обучении.

5. Программная система АОТО значительно превосходит по производительности компиляторы общего назначения Clang и GCC.

Основные результаты диссертационной работы.

1. Разработана новая модель гипотетического процессора, которая позволяет сократить время выполнения матрично-векторных операций и их обобщений на замкнутые полукольца с элементами из множества вещественных чисел.

2. Разработаны новые алгоритмы выполнения тензорных операций константной сложности относительно размерности тензоров, уменьшающие время выполнения таких операций. Выведены формулы, позволяющие получить значения параметров алгоритмов выполнения тензорных операций в зависимости от характеристик многоядерных процессоров общего назначения для архитектур x86-64, x86, ppc64le, aarch64.

3. Разработана программная система АОТО для автоматической оптимизации времени выполнения тензорных операций и их автоматического распараллеливания при компиляции программ для многоядерных процессоров общего назначения. Получена оценка производительности многопоточной программы, представленной группой полностью вложенных циклов. Автоматическая оптимизация времени выполнения обобщения матричного произведения внедрена в основной код Polly проекта LLVM.

4. С помощью экспериментов при решении обратной задачи гравиметрии, общей задачи о путях, оптимизации матрично-векторных операций и тензорных сверток подтверждена применимость программной системы АОТО для оптимизации времени выполнения тензорных операций. Показано, что программная система АОТО сопоставима по производительности скомпилированного кода с кодом библиотек Intel MKL, OpenBLAS, BLIS, реализующих матричные и матрично-векторные произведения; с фреймворками TCCG и TBLIS, реализующими свертки тензоров; со специали-

зированным компилятором ICC и существенно превосходит компиляторы общего назначения Clang и GCC.

Основные результаты диссертационной работы являются новыми и не содержатся в ранее опубликованных научных работах других авторов, приведенных в разделе 1. Рассмотрим основные отличия:

1) ни одна из известных моделей гипотетических процессоров не позволяет смоделировать выполнение операций предвыборки данных в кэш-память и операций из замкнутых полуколец;

2) все известные алгоритмы сокращения времени выполнения тензорных операций основаны на ручной настройке и автонастройке, требующих доступа к целевой аппаратной платформе и значительных временных затрат на оптимизацию;

3) все известные компиляторы не позволяют автоматически получать высокопроизводительные реализации тензорных операций.

Полученные методы могут использоваться для оптимизации программ в процессе компиляции; для упрощения создания новых реализаций BLAS; уменьшения времени выполнения автонастройки, оценивающей время выполнения нескольких реализаций одной и той же программы.

Основным направлением дальнейших исследований предполагается создание модели гипотетического процессора для описания ГПУ и получения значений параметров алгоритмов. Другим направлением дальнейших исследований может быть разработка методов автоматического моделирования выполнения алгоритмов с целью автоматического вывода формул, позволяющих получить значения параметров алгоритмов в зависимости от характеристик многоядерных процессоров общего назначения.

Автор выражает искреннюю благодарность своему научному руководителю — доктору физико-математических наук, ведущему научному сотруднику ИММ УрО РАН Елене Николаевне Акимовой.

Литература

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления // Санкт-Петербург, БХВ-Петербург. 2002. 599 с.
2. Ортега Дж. Параллельные вычисления // Москва, Мир. 1991. 366 с.
3. Akimova E.N., Belousov D.V . Parallel algorithms for solving linear systems with block-tridiagonal matrices on multi-core CPU with GPU // Journal of Computational Science. 2012. Vol. 3, no. 6. P. 445—449. DOI: 10.1016/j.jocs.2012.08.004.
4. Akimova E.N., Misilov V.E. Efficient numerical algorithm for solving the gravimetry problem of finding a lateral density in a layer: Parallel implementation // Mathematical Methods in the Applied Sciences. 2020. Vol. 43, no. 13. P. 7774—7787. DOI: 10.1002/mma.6206.
5. Gergel V.P. A Global Optimization Algorithm for Multivariate Functions with Lipschitzian First Derivatives // Journal of Global Optimization. 1997. Vol. 10. P. 257—281. DOI: 10.1023/A:1008290629896.
6. Sokolinsky L.B. Organization of Parallel Query Processing in Multiprocessor Database Machines with Hierarchical Architecture // Programming and Computer Software. 2001. Vol. 27. P. 297—308. DOI: 10.1023/A:1012706401123.
7. Abramov S., Glück R. Principles of Inverse Computation and the Universal Resolving Algorithm // Lecture Notes in Computer Science. 2002. Vol. 10. P. 269—295. DOI: 10.1007/3-540-36377-7_13.
8. Ammaev S.G., Gervich L.R., Steinberg B.Y. Combining Parallelization with Overlaps and Optimization of Cache Memory Usage // Lecture Notes in Computer Science. 2017. Vol. 10421. P. 257—264. DOI: 10.1007/978-3-319-62932-2_24.
9. Пан К.С., Цымблер М.Л. Разработка параллельной СУБД на основе последовательной СУБД PostgreSQL с открытым исходным кодом //

Вестник ЮУрГУ. Серия: Математическое моделирование и программирование. 2012. Т. 18 (277). С. 112—120.

10. Sedukhin S.G., Paprzycki M. Generalizing Matrix Multiplication for Efficient Computations on Modern Computers // Lecture Notes in Computer Science. 2012. Vol. 7203. P. 225—234. DOI: 10.1007/978-3-642-31464-3_23.
11. Graph BLAS Forum. URL: `http://graphblas.org/index.php?title=Graph_BLAS_Forum` (дата обращения: 5.09.2020).
12. Akihito T., Sedukhin S. Parallel Blocked Algorithm for Solving the Algebraic Path Problem on a Matrix Processor // Lecture Notes in Computer Science. 2005. Vol. 3726. P. 786—795. DOI: 10.1007/11557654_89.
13. Lawson C.L., Hanson R.J., Kincaid D.R., Krogh F.T. Basic Linear Algebra Subprograms for Fortran Usage // ACM Transactions on Mathematical Software. 1979. Vol. 5, no. 3. P. 308—323. DOI: 10.1145/355841.355847.
14. Dongarra J.J., Du Croz J., Hammarling S., Hanson R.J. An Extended Set of FORTRAN Basic Linear Algebra Subprograms // ACM Transactions on Mathematical Software. 1988. Vol. 14, no. 1. P. 1—17. DOI: 10.1145/42288.42291.
15. Dongarra J.J., Du Croz J., Hammarling S., Duff I.S. A Set of Level 3 Basic Linear Algebra Subprograms // ACM Transactions on Mathematical Software. 1990. Vol. 16, no. 1. P. 1—17. DOI: 10.1145/77626.79170.
16. Matthews D. High-Performance Tensor Contraction without BLAS // SIAM Journal on Scientific Computing. 2018. Vol. 40, no. 1. P. C1—C24. DOI: 110.1137/16m108968x.
17. Springer P., Bientinesi P. Design of a High-Performance GEMM-like Tensor–Tensor Multiplication // ACM Transactions on Mathematical Software. 2018. Vol. 44, no. 3. P. 28:1—28:29. DOI: 10.1145/3157733.
18. Jacob B., Kligys S., Chen B., Zhu M. et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference // 2018 IEEE/CVF Conference on Computer Vision and Pattern Recog-

nition (Salt Lake City, UT, USA, June 18—23, 2018). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2018. P. 2704—2713. DOI: 10.1109/CVPR.2018.00286.

19. Векуа И.Н. Основы тензорного анализа и теории ковариантов // Москва, Наука. 1978. 298 с.
20. Победря Б.Е. Деформационная теория пластичности анизотропных сред // Прикладная математика и механика. 1984. Т. 48, № 1. С. 29—37.
21. Тыртышников Е.Е. Тензорные аппроксимации матриц, порожденных асимптотически гладкими функциями // Математический сборник. 2003. Т. 194, № 6. С. 147—160. DOI: 10.4213/sm747.
22. Oseledets I.V. Tensor-Train decomposition // SIAM Journal on Scientific Computing. 2011. Vol. 33, no. 5. P. 2295—2317. DOI: 10.1137/090752286.
23. Gates M., Luszczek P., Abdelfattah A., Kurzak J. et al. C++ API for BLAS and LAPACK. SLATE Working Notes. 2017. URL: `https://www.icl.utk.edu/files/publications/2017/icl-utk-1031-2017.pdf` (дата обращения: 5.09.2020).
24. Kågström B., Ling P., Van Loan C. GEMM-Based Level 3 BLAS: High--Performance Model Implementations and Performance Evaluation Benchmark // ACM Transactions on Mathematical Software. 1995. Vol. 24, no. 3. P. 268—302. DOI: 10.1145/292395.292412.
25. Frison G. Algorithms and Methods for High-Performance Model Predictive Control // Ph.D. thesis. 2016. URL: `https://backend.orbit.dtu.dk/ws/portalfiles/portal/124371046/phd402_Frison_G.pdf` (дата обращения: 5.09.2020).
26. Goto K., Van De Geijn R. High-Performance Implementation of the Level-3 BLAS // ACM Transactions on Mathematical Software. 2008. Vol. 35, no. 1. P. 4:1—4:14. DOI: 10.1145/1377603.1377607.
27. Goto K., Geijn R.A. van de. Anatomy of High-Performance Matrix Multiplication // ACM Transactions on Mathematical Software. 2008. Vol. 34,

no. 3. P. 12:1—12:25. DOI: 10.1145/1356052.1356053.

28. Xianyi Z., Qian W., Yunquan Z. Model-driven level 3 BLAS performance optimization on Loongson 3A processor // 2012 IEEE 18th International Conference on Parallel and Distributed Systems (Singapore, December 17—19, 2012). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2012. P. 684—691. DOI: 10.1109/ICPADS.2012.97.
29. Van Zee F.G., van de Geijn R.A. BLIS: A Framework for Rapidly Instantiating BLAS Functionality // ACM Transactions on Mathematical Software. 2015. Vol. 41, no. 3. P. 14:1—14:33. DOI: 10.1145/2764454.
30. Intel. Intel Math Kernel Library (Intel MKL). URL: https://software.intel.com/ru-ru/intel-mkl/ (дата обращения: 5.09.2020).
31. IBM. IBM Engineering and Scientific Subroutine Library Intel Math Kernel Library (Intel MKL). URL: https://www.ibm.com/support/knowledgecenter/SSFHY8_6.1/reference/essl_reference_pdf.pdf (дата обращения: 5.09.2020).
32. ARM. ARM Performance Libraries Reference Manual. URL: https://ds.arm.com/media/uploads/arm_performance_libraries_reference_manual_arm-ecm-0493690.pdf (дата обращения: 5.09.2020).
33. Gunnels J.A., Henry G.M., van de Geijn R.A. A Family of High-Performance Matrix Multiplication Algorithms // Lecture Notes in Computer Science. 2001. Vol. 2073. P. 51—60. DOI: 10.1007/3-540-45545-0_15.
34. Agarwal R.C., Gustavson F.G., Zubair M. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms // IBM Journal of Research and Development. 1994. Vol. 38, no. 5. P. 563—576. DOI: 10.1147/rd.385.0563.
35. Whaley R., Petitet A., Dongarra J. Automated empirical optimizations of software and the ATLAS project // Parallel Computing. 2001. Vol. 27. P. 3—35. DOI: 10.1016/S0167-8191(00)00087-9.

36. Hassan S.A., Mahmoud M.M.M., Hemeida A.M., Saber M.A. Effective Implementation of Matrix–Vector Multiplication on Intel's AVX multicore Processor // Computer Languages, Systems & Structures. 2018. Vol. 51. P. 158—175. DOI: 10.1016/j.cl.2017.06.003.

37. Liang J., Zhang Y. Optimization of GEMV on Intel AVX processor // International Journal of Database Theory and Application. 2016. Vol. 9, no. 2. P. 47—60. DOI: 10.14257/ijdta.2016.9.2.06.

38. Юрушкин М.В. Двойное блочное размещение данных в оперативной памяти при решении задачи умножения матриц // Программная инженерия. 2016. Т. 7, № 3. С. 132—139.

39. Frison G., Quirynen R., Zanelli A., Diehl M. et al. Hardware Tailored Linear Algebra for Implicit Integrators in Embedded NMPC // IFAC-PapersOnLine. 2017. Vol. 50, no. 1. P. 14392—14398. DOI: 10.1016/j.ifacol.2017.08.2026.

40. Whaley R.C., Dongarra J.J. Automatically Tuned Linear Algebra Software // The 1998 ACM/IEEE Conference on Supercomputing (Orlando, FL, USA, USA, November 7—13, 1998). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 1998. P. 1—27. DOI: 10.1109/SC.1998.10004.

41. Bilmes J., Asanovic K., Chin C., Demmel J. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology // The 11th International Conference on Supercomputing (Vienna, Austria, July 7—11, 1997). New York, NY, USA, ACM, 1997. P. 340—347. DOI: 10.1145/2591635.2591656.

42. Spampinato D.G., Markus P. A Basic Linear Algebra Compiler // Annual IEEE/ACM International Symposium on Code Generation and Optimization (Orlando, FL, USA, February 15—19, 2014). New York, NY, USA, ACM, 2014. P. 23—32. DOI: 10.1145/2544137.2544155.

43. Belter G., Jessup E.R., Karlin I., Siek J.G. Automating the Generation of Composed Linear Algebra Kernels // The Conference on High Performance

Computing Networking, Storage and Analysis (Portland, Oregon, November 14—20, 2009). New York, NY, USA, ACM, 2009. P. 59:1—59:12. DOI: 10.1145/1654059.1654119.

44. Wang Q., Zhang X., Zhang Y., Yi Q. AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs // The International Conference on High Performance Computing, Networking, Storage and Analysis (Denver, Colorado, USA, November 17—22, 2013). New York, NY, USA, ACM, 2013. P. 25:1—25:12. DOI: 10.1145/2503210.2503219.

45. Knijnenburg P.M.W., Kisuki T., O'Boyle M.F.P. Iterative Compilation // Lecture Notes in Computer Science. 2002. Vol. 2268. P. 171—187. DOI: 10.1007/3-540-45874-3_10.

46. Yotov K., Xiaoming L., Gang R., Garzaran M.J.S. et al. Is Search Really Necessary to Generate High-Performance BLAS? // Proceedings of the IEEE. 2005. Vol. 93, no. 2. P. 358—386. DOI: 10.1109/JPROC.2004.840444.

47. Low T.M., Igual F.D., Smith T.M., Quintana-Orti E.S. Analytical Modeling Is Enough for High-Performance BLIS // ACM Transactions on Mathematical Software. 2016. Vol. 43, no. 2. P. 12:1—12:18. DOI: 10.1145/2925987.

48. Demmel J. Communication Avoiding Algorithms // 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (Salt Lake City, UT, USA, November 10—16, 2012). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2012. P. 1942—2000. DOI: 10.1109/SC.Companion.2012.351.

49. Ballard G., Carson E., Demmel J., Hoemmen M. et al. Communication lower bounds and optimal algorithms for numerical linear algebra // Acta Numerica. 2014. Vol. 23. P. 1—155. DOI: 10.1017/S0962492914000038.

50. Frigo M., Leiserson C.E., Prokop H., Ramachandran S. Cache-Oblivious Algorithms // The 40th Annual Symposium on Foundations of Computer Science (New York City, NY, USA, October 17—19, 1999). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 1999. P. 285—297. DOI:

10.1109/SFFCS.1999.814600.

51. Yotov K., Roeder T., Pingali K., Gunnels J. et al. An Experimental Comparison of Cache-Oblivious and Cache-Conscious Programs // Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (San Diego, California, USA, June 9—11, 2007). New York, NY, USA, ACM, 2007. P. 93—104. DOI: 10.1145/1248377.1248394.
52. Heinecke A., Henry G., Hutchinson M., Pabst H. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation // SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Salt Lake City, UT, USA, November 13—18, 2016). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2016. P. 981—991. DOI: 10.1109/SC.2016.83.
53. Su X., Liao X., Xue J. Automatic Generation of Fast BLAS3-GEMM: A Portable Compiler Approach // The 2017 International Symposium on Code Generation and Optimization (Austin, TX, USA, February 4—8, 2017). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2017. P. 122—133. DOI: 10.1109/CGO.2017.7863734.
54. Hirata S. Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories // The Journal of Physical Chemistry A. 2003. Vol. 107, no. 46. P. 9887—9897. DOI: 10.1021/jp034596z.
55. Solomonik E., Matthews D., Hammond J. R., Stanton J.F. et al. A massively parallel tensor contraction framework for coupled-cluster computations // Journal of Parallel and Distributed Computing. 2014. Vol. 74, no. 12. P. 3176—3190. DOI: 10.1016/j.jpdc.2014.06.002.
56. Epifanovsky E., Wormit M., Kuś T., Landau A. et al. New implementation of high-level correlated methods using a general block tensor library for high--performance electronic structure calculations // Journal of Computational Chemistry. 2013. Vol. 34. P. 2293—2309. DOI: 10.1002/jcc.23377.

57. Calvin J.A., Lewis C.A., Valeev E.F. Scalable Task-Based Algorithm for Multiplication of Block-Rank-Sparse Matrices // The 5th Workshop on Irregular Applications: Architectures and Algorithms (Austin, Texas, USA, November 15, 2015). New York, NY, USA, ACM, 2015. P. 4:1—4:8. DOI: 10.1145/2833179.2833186.
58. Calvin J., Valeev E. Task-Based Algorithm for Matrix Multiplication: A Step Towards Block-Sparse Tensor Computing // The 5th Workshop on Irregular Applications: Architectures and Algorithms (Austin, Texas, USA, November 15, 2015). 2015. P. 1—9. URL: `https://arxiv.org/pdf/1504.05046.pdf` (дата обращения: 5.09.2020).
59. Napoli E.D. Towards an efficient use of the BLAS library for multilinear tensor contractions // Applied Mathematics and Computation. 2014. Vol. 235. P. 454—468. DOI: 10.1016/j.amc.2014.02.051.
60. Aprà E., Klemm M., Kowalski. Efficient Implementation of Many-Body Quantum Chemical Methods on the Intel E; Xeon PhiTM Coprocessor // The International Conference for High Performance Computing, Networking, Storage and Analysis (New Orleans, Louisiana, United States, November 16—21, 2014). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2014. P. 674—684. DOI: 10.1109/SC.2014.60.
61. Stock K., Henretty T., Murugandi I., Sadayappan P. et al. Model-Driven SIMD Code Generation for a Multi-resolution Tensor Kernel // The 2011 IEEE International Parallel & Distributed Processing Symposium (Anchorage, AK, USA, May 16—20, 2011). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2011. P. 1058—1067. DOI: 10.1109/IPDPS.2011.101.
62. Ma W., Krishnamoorthy S., Villa O., Kowalski K. GPU-Based Implementations of the Noniterative Regularized-CCSD(T) Corrections: Applications to Strongly Correlated Systems // Journal of Chemical Theory and Computation. 2011. Vol. 7, no. 5. P. 1316—1327. DOI: 10.1021/ct1007247.

63. Intel. Intel C++ Compiler 16.0 Update 4 User and Reference Guide. Intel. URL: `https://software.intel.com/en-us/intel-cplusplus-compiler-16.0-user-and-reference-guide-pdf` (дата обращения: 5.09.2020).
64. Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's Thesis. 2002. URL: `http://llvm.org/pubs/2002-12-LattnerMSThesis-book.pdf` (дата обращения: 5.09.2020).
65. Gough B.J., Stallman R.M. An Introduction to GCC: For the GNU Compilers GCC and G++ // Bristol, UK, Network Theory. 2004. 144 p.
66. IBM. XL C/C++: Compiler Reference - IBM. IBM. URL: `http://www-01.ibm.com/support/docview.wss?uid=swg27024742&aid=1` (дата обращения: 5.09.2020).
67. Cong J., Xiao B. Minimizing Computation in Convolutional Neural Networks // Lecture Notes in Computer Science. 2014. Vol. 8681. P. 281—290. DOI: 10.1007/978-3-319-11179-7_36.
68. Gordon M., Duh K., Andrews N. Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning // The 5th Workshop on Representation Learning for NLP (Seattle, USA, July 9, 2020). Stroudsburg, Pennsylvania, USA, ACL, 2020. P. 143—155. DOI: 10.18653/v1/2020.repl4nlp-1.18.
69. Srinivas S., Subramanya A., Babu R. V. Training Sparse Neural Networks // 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) (Honolulu, HI, USA, July 21—26, 2017). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2017. P. 455—462. DOI: 10.1109/CVPRW.2017.61.
70. Olivares-Amaya R., Watson M., Edgar R., Vogt L. et al. Accelerating Correlated Quantum Chemistry Calculations Using Graphical Processing Units and a Mixed Precision Matrix Multiplication Library // Journal of Chemical Theory and Computation. 2010. Vol. 6, no. 1. P. 135—144. DOI: 10.1021/ct900543q.

71. Cooper B., Girdlestone S., Burovskiy P., Gaydadjiev G. et al. Quantum Chemistry in Dataflow: Density-Fitting MP2 // Journal of Chemical Theory and Computation. 2017. Vol. 13, no. 11. P. 5265—5272. DOI: 10.1021/acs.jctc.7b00649.
72. Kurashige Y. Multireference electron correlation methods with density matrix renormalisation group reference functions // Molecular Physics. 2014. Vol. 112, no. 11. P. 1485—1494. DOI: 10.1080/00268976.2013.843730.
73. Yu J., Lukefahr A., Palframan D., Dasika G. et al. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism // The 44th Annual International Symposium on Computer Architecture (Toronto, ON, Canada, June 24—28, 2017). New York, NY, USA, ACM, 2017. P. 548—560. DOI: 10.1145/3079856.3080215.
74. Yao Z., Cao S., Xiao W., Zhang C. et al. Balanced Sparsity for Efficient DNN Inference on GPU // Proceedings of the AAAI Conference on Artificial Intelligence. 2019. Vol. 33, no. 1. P. 5676—5683. DOI: 10.1609/aaai.v33i01.33015676.
75. Zhu M., Zhang T., Gu Z., Xie Y. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-Wise Sparse Neural Networks on Modern GPUs // The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (Columbus, OH, USA, October 12—16, 2019). New York, NY, USA, ACM, 2019. P. 359—371. DOI: 10.1145/3352460.3358269.
76. Yang X., Parthasarathy S., Sadayappan P. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining // Proceedings of the VLDB Endowment. 2011. Vol. 4, no. 4. P. 231—242. DOI: 10.14778/1938545.1938548.
77. Kang U., Tsourakakis C. E., Faloutsos C. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations // 2009 Ninth IEEE International Conference on Data Mining (Miami, FL, USA, December 6—9, 2009). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2009.

P. 229—238. DOI: 10.1109/ICDM.2009.14.

78. Besta M., Marending F., Solomonik E., Hoefler T. SlimSell: A Vectorizable Graph Representation for Breadth-First Search // 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (Orlando, FL, USA, 29 May—2 June, 2017). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2017. P. 32—41. DOI: 10.1109/IPDPS.2017.93.
79. Kaushik D., Gropp W., Minkoff M., Smith B. Improving the Performance of Tensor Matrix Vector Multiplication in Cumulative Reaction Probability Based Quantum Chemistry Codes // Lecture Notes in Computer Science. 2008. Vol. 5374. P. 120—130. DOI: 10.1007/978-3-540-89894-8_14.
80. Tóbiás R., Stachó L., Tasi G. First-order chemical reaction networks I: theoretical considerations // Journal of Mathematical Chemistry. 2016. Vol. 54. P. 1863—1878. DOI: 10.1007/s10910-016-0655-2.
81. Kendrick B.K. Non-adiabatic quantum reactive scattering in hyperspherical coordinates // The Journal of Chemical Physics. 2018. Vol. 148. P. 044116–1—044116–29. DOI: 10.1063/1.5014989.
82. Chernykh I., Kulikov I., Glinsky B., Vshivkov V. et al. Advanced Vectorization of PPML Method for Intel® Xeon® Scalable Processors // Communications in Computer and Information Science. 2019. P. 465—471. DOI: 10.1007/978-3-030-05807-4_39.
83. Yi Q., Wang Q., Cui H. Specializing Compiler Optimizations through Programmable Composition for Dense Matrix Computations // The 47th Annual IEEE/ACM International Symposium on Microarchitecture (Cambridge, UK, December 13—17, 2014). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2014. P. 596—608. DOI: 10.1109/MICRO.2014.14.
84. Sidnev A. Hardware-Specific Selection the Most Fast-Running Software Components // Lecture Notes in Computer Science. 2016. Vol. 10049. P. 354—364. DOI: 10.1007/978-3-319-49956-7_28.

85. Falch T.L., Elster A.C. Machine Learning Based Auto-Tuning for Enhanced OpenCL Performance Portability // 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (Hyderabad, India, May 25—29, 2015). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2015. P. 1231—1240. DOI: 10.1109/IPDPSW.2015.85.
86. Nugteren C. CLBlast: A Tuned OpenCL BLAS Library // The International Workshop on OpenCL (Oxford, United Kingdom, May 14—16, 2018). New York, NY, USA, ACM, 2018. P. 1—10. DOI: 10.1145/3204919.3204924.
87. Park E., Cavazos J., Pouchet L., Bastoul C. et al. Predictive Modeling in a Polyhedral Optimization Space // International Symposium on Code Generation and Optimization (CGO 2011) (Chamonix, France, April 2—6, 2011). Vol. 41. Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2013. P. 119—129. DOI: 10.1109/CGO.2011.5764680.
88. Ashouri A.H., Killian W., Cavazos J., Palermo G. et al. A Survey on Compiler Autotuning Using Machine Learning // ACM Computing Surveys. 2018. Vol. 51, no. 5. P. 96:1—96:42. DOI: 10.1145/3197978.
89. Fog A. Instruction Tables. Technical University of Denmark. 2020. URL: https://www.agner.org/optimize/instruction_tables.pdf (дата обращения: 5.09.2020).
90. ARM. Cortex-A57 Software Optimization Guide. URL: http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/Cortex_A57_Software_Optimization_Guide_external.pdf (дата обращения: 5.09.2020).
91. Abadi M., Barham P., Chen J., Chen Z. et al. TensorFlow: A System for Large-Scale Machine Learning // The 12th USENIX Conference on Operating Systems Design and Implementation (Savannah, GA, USA, November 2—4, 2016). Berkeley, USENIX Association, 2016. P. 265—283.
92. Kossaifi J., Khanna A., Lipton Z., Furlanello T. et al. Tensor Contraction Layers for Parsimonious Deep Nets // 2017 IEEE Conference on Computer

Vision and Pattern Recognition Workshops (CVPRW) (Honolulu, HI, USA, July 21—26, 2017). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2017. P. 1940—1946. DOI: 10.1109/CVPRW.2017.243.

93. Ji Y., Wang Q., Li X., Liu J. A survey on Tensor techniques and applications in machine learning // IEEE Access. 2019. Vol. 7. P. 162950—162990. DOI: 10.1109/ACCESS.2019.2949814.

94. Pekurovsky D. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions // SIAM Journal on Scientific Computing. 2012. Vol. 34, no. 4. P. C192—C209. DOI: 10.1137/11082748X.

95. Bartlett R., Musial M. Coupled-Cluster Theory in Quantum Chemistry // Reviews of Modern Physics. 2007. Vol. 79, no. 1. P. 291—352. DOI: 10.1103/RevModPhys.79.291.

96. Harrison R.J., Beylkin G., Bischoff F.A., Calvin J.A. et al. MADNESS: A Multiresolution, Adaptive Numerical Environment for Scientific Simulation // SIAM Journal on Scientific Computing. 2016. Vol. 38, no. 5. P. S123—S142. DOI: 10.1137/15M1026171.

97. Deville M.O., Fischer P.F., Mund E.H. High-Order Methods for Incompressible Fluid Flow // Cambridge University Press. 2002. 528 p. DOI: 10.1017/CBO9780511546792.

98. Tufo H.M., Fischer P.F. Terascale Spectral Element Algorithms and Implementations // The 1999 ACM/IEEE Conference on Supercomputing (Portland, Oregon, USA, November 13—19, 1999). New York, NY, USA, ACM, 1999. P. 1—14. DOI: 10.1145/331532.331599.

99. Rink N., Susungi A., Castrillón J., Stiller J. et al. CFDlang: High-level code generation for high-order methods in fluid dynamics // The Real World Domain Specific Languages Workshop 2018 (Vienna, Austria, February 24—24, 2018). New York, NY, USA, ACM, 2018. P. 1—10. DOI: 10.1145/3183895.3183900.

100. Moxey D., Cantwell C.D., Bao Y., Cassinelli A. et al. Nektar++: Enhancing the capability and application of high-fidelity spectral/hp element methods // Computer Physics Communications. 2020. Vol. 249. P. 1—18. DOI: 10.1016/j.cpc.2019.107110.
101. Frigo M., Johnson S.G. The Design and Implementation of FFTW3 // Proceedings of the IEEE. 2005. Vol. 93, no. 2. P. 216—231. DOI: 10.1109/JPROC.2004.840301.
102. Park C.-S., Ko S.-J. The Hopping Discrete Fourier Transform [sp Tips&Tricks] // Signal Processing Magazine, IEEE. 2014. Vol. 31. P. 135—139. DOI: 10.1109/MSP.2013.2292891.
103. Wang S., Kim S.-H., Liu Y., Ryu H.-K. et al. Super-Resolution Algorithm Based on Discrete Fourier Transform // Lecture Notes in Computer Science. 2010. Vol. 6216. P. 368—375. DOI: 10.1007/978-3-642-14932-0_46.
104. Beaudoin N., Beauchemin S.S. An accurate discrete Fourier transform for image processing // Object recognition supported by user interaction for service robots (Quebec City, Quebec, August 11—15, 2002). Vol. 3. Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2002. P. 981—991. DOI: 10.1109/SC.2016.83.
105. Drake J., Foster I., Michalakes J., Toonen B. et al. Design and Performance of a Scalable Parallel Community Climate Model // Parallel Computing. 1995. Vol. 21, no. 10. P. 1571—1591. DOI: 10.1016/0167-8191(96)80001-9.
106. Logg A. Automating the Finite Element Method // Archives of Computational Methods in Engineering. 2007. Vol. 14. P. 93–138. DOI: 10.1007/s11831-007-9003-9.
107. Lenzen M., Sun Y.-Y., Faturay F., Ting Y.-P. et al. The carbon footprint of global tourism // Nature Climate Change. 2018. Vol. 8. P. 522—528. DOI: 10.1038/s41558-018-0141-x.
108. van der Walt S., Colbert S.C., Varoquaux G. The NumPy Array: A Structure for Efficient Numerical Computation // Computing in Science Engi-

neering. 2011. Vol. 13, no. 2. P. 22—30. DOI: 10.1109/MCSE.2011.37.
109. Guennebaud G. Eigen v3. URL: http://eigen.tuxfamily.org (дата обращения: 5.09.2020).
110. Bader B.W., Kolda T.G. Algorithm 862: MATLAB Tensor Classes for Fast Algorithm Prototyping // ACM Transactions on Mathematical Software. 2006. Vol. 32, no. 4. P. 635—653. DOI: 10.1145/1186785.1186794.
111. Гареев Р.А. Методы оптимизации обобщенных тензорных сверток // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2020. Т. 9, № 2. С. 19—39. DOI: 10.14529/cmse200202.
112. Byna S., Chen Y., Sun X. Taxonomy of Data Prefetching for Multicore Processors // Journal of Computer Science and Technology. 2009. Vol. 24, no. 3. P. 405–417. DOI: 10.1007/s11390-009-9233-4.
113. Lee J., Kim H., Vuduc R. When Prefetching Works, When It Doesn't, and Why // ACM Transactions on Architecture and Code Optimization. 2012. Vol. 9, no. 1. P. 2:1—2:29. DOI: 10.1145/2133382.2133384.
114. Aho A.V., Hopcroft J.E. The Design and Analysis of Computer Algorithms // Boston, Massachusetts, USA, Addison-Wesley. 1974. 470 p.
115. Abdali S.K., Saunders B.D. Transitive closure and related semiring properties via eliminants // Theoretical Computer Science. 1985. Vol. 40. P. 257—274. DOI: 10.1016/0304-3975(85)90170-7.
116. Sedukhin S., Miyazaki T., Kuroda K. Orbital Systolic Algorithms and Array Processors for Solution of the Algebraic Path Problem // IEICE Transactions on Information and Systems. 2010. Vol. 93-D, no. 3. P. 534—541. DOI: 10.1587/transinf.E93.D.534.
117. Amirteimoori A. An extended shortest path problem: A data envelopment analysis approach // Applied Mathematics Letters. 2012. Vol. 25, no. 11. P. 1839—1843. DOI: 10.1016/j.aml.2012.02.042.
118. Jahed R., Amirteimoori A., Azizi H. Performance measurement of decision-making units under uncertainty conditions: An approach based on dou-

ble frontier analysis // Measurement. 2015. Vol. 69. P. 264—279. DOI: 10.1016/j.measurement.2015.03.014.

119. Ghahraman A., Prior D. A learning ladder toward efficiency: Proposing network-based stepwise benchmark selection // Omega. 2016. Vol. 63. P. 83—93. DOI: 10.1016/j.omega.2015.10.004.

120. Haymond R.E., Thornton J.R., Warner D.D. A Shortest Path Algorithm in Robotics and Its Implementation on the FPS T-20 Hypercube // Annals of Operations Research. 1988. Vol. 14. P. 305—320. DOI: 10.1007/BF02186485.

121. Briggs A., Detweiler C., Scharstein D., Vandenberg-Rodes A. Expected Shortest Paths for Landmark-Based Robot Navigation // Springer Tracts in Advanced Robotics. 2004. Vol. 7. P. 381—398. DOI: 10.1007/978-3-540-45058-0_23.

122. Xue Y., Sun J.Q. Solving the Path Planning Problem in Mobile Robotics with the Multi-Objective Evolutionary Algorithm // Applied Sciences. 2018. Vol. 8, no. 9. P. 1—21. DOI: 10.3390/app8091425.

123. Dong M., Wu C., Hou F. Shortest path based simulated annealing algorithm for dynamic facility layout problem under dynamic business environment // Expert Systems with Applications. 2009. Vol. 36, no. 8. P. 11221—11232. DOI: 10.1016/j.eswa.2009.02.091.

124. Besbes M., Zolghadri M., Affonso R.C., Masmoudi F. et al. A methodology for solving facility layout problem considering barriers: genetic algorithm coupled with A* search // Journal of Intelligent Manufacturing. 2020. Vol. 31. P. 615—640. DOI: 10.1007/s10845-019-01468-x.

125. Kheirkhah A.S., Messi Bidgoli M. A graph-theoretic-based meta-heuristic algorithm for solving cooperative dynamic facility layout problems // International Journal of Process Management and Benchmarking. 2018. Vol. 8, no. 3. P. 340—366. DOI: 10.1504/IJPMB.2018.10012846.

126. Setiawan E., Gunawan G., Maryati I., Santoso J. et al. Shortest Path Problem for Public Transportation Using GPS and Map Service // Proce-

dia - Social and Behavioral Sciences. 2012. Vol. 57. P. 426—431. DOI: 10.1016/j.sbspro.2012.09.1207.

127. Jigang W., Jin S., Ji H., Srikanthan T. Algorithm for Time-dependent Shortest Safe Path on Transportation Networks // Procedia Computer Science. 2011. Vol. 4. P. 958—966. DOI: 10.1016/j.procs.2011.04.101.

128. Idri A., Oukarfi M., Boulmakoul A., Zeitouni K. et al. A new time-dependent shortest path algorithm for multimodal transportation network // Procedia Computer Science. 2017. Vol. 109. P. 692—697. DOI: 10.1016/j.procs.2017.05.379.

129. Peyer S., Rautenbach D., Vygen J. A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing // Journal of Discrete Algorithms. 2009. Vol. 7, no. 4. P. 377—390. DOI: 10.1016/j.jda.2007.08.003.

130. Zheng S.Q., Lim J.S., Iyengar S. Routing using implicit connection graphs [VLSI design] // The 9th International Conference on VLSI Design (Bangalore, India, India, January 3–6, 1996). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 1996. P. 49—52. DOI: 10.1109/ICVD.1996.489453.

131. Sen S. VLSI Routing in Multiple Layers using Grid based Routing Algorithms // International Journal of Computer Applications. 2014. Vol. 93, no. 16. P. 41—45. DOI: 10.5120/16303-6210.

132. MapQuest. URL: `https://www.mapquest.com/` (дата обращения: 5.09.2020).

133. Google Maps. URL: `https://www.google.ru/maps` (дата обращения: 5.09.2020).

134. Rodríguez-Puente R., Lazo-Cortés M. Algorithm for shortest path search in Geographic Information Systems by using reduced graphs // SpringerPlus. 2013. Vol. 2, no. 291. P. 1—13. DOI: 10.1186/2193-1801-2-291.

135. Kai N., Yao-ting Z., Yue-peng M. Shortest Path Analysis Based on Dijkstra's Algorithm in Emergency Response System // Indonesian Journal of Electrical Engineering and Computer Science. 2014. Vol. 12. P. 3476—3482.

DOI: 10.11591/TELKOMNIKA.V12I5.3236.

136. Manliguez C., Diche Z.J., Jimenez M., Agrazamendez M. et al. GIS-based Evacuation Routing using Capacity Aware Shortest Path Evacuation Routing Algorithm and Analytic Hierarchy Process for Flood Prone Communities // The 3rd International Conference on Geographical Information Systems Theory, Applications and Management (Porto, Portugal, April 27—28, 2017). Setúbal, Portugal, SciTePress, 2017. P. 237—243. DOI: 10.5220/0006327402370243.

137. Toss J., Comba J., Raffin B. Parallel Shortest Path Algorithm for Voronoi Diagrams with Generalized Distance Functions // 2014 27th SIBGRAPI Conference on Graphics, Patterns and Images (Rio de Janeiro, Brazil, August 26—30, 2014). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2014. P. 212—219. DOI: 10.1109/SIBGRAPI.2014.1.

138. Bae S., Chwa K.-Y. Shortest Paths and Voronoi Diagrams with Transportation Networks Under General Distances // Lecture Notes in Computer Science. 2005. Vol. 3827. P. 1007—1018. DOI: 10.1007/11602613_100.

139. Toss J., Comba J., Raffin B. Parallel Voronoi Computation for Physics-Based Simulations // Computing in Science & Engineering. 2016. Vol. 18, no. 3. P. 88—94. DOI: 10.1109/MCSE.2016.52.

140. Toan T.Q., Sorokin A.A., Trang V.T. H. Using modification of visibility-graph in solving the problem of finding shortest path for robot // 2017 International Siberian Conference on Control and Communications (SIBCON) (Astana, Kazakhstan, June 29—30, 2017). 2017. P. 1—6. DOI: 10.1109/SIBCON.2017.7998564.

141. Sun L., Liu X., Leng M. An Effective Algorithm of Shortest Path Planning in a Static Environment // IFIP International Federation for Information Processing. 2006. Vol. 207. P. 257—262. DOI: 10.1007/0-387-34403-9_35.

142. Ganganath N., Cheng C., Fernando T., Iu H.H.C. et al. Shortest Path Planning for Energy-Constrained Mobile Platforms Navigating on Uneven

Terrains // IEEE Transactions on Industrial Informatics. 2018. Vol. 14, no. 9. P. 4264—4272. DOI: 10.1109/TII.2018.2844370.

143. Lehmann D.J. Algebraic structures for transitive closure // Theoretical Computer Science. 1977. Vol. 4, no. 1. P. 59—76. DOI: 10.1016/0304-3975(77)90056-1.

144. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms, Third Edition // Cambridge, Massachusetts, USA, The MIT Press. 2009. 1292 p.

145. Floyd R.W. Algorithm 97: Shortest Path // Communications of the ACM. 1962. Vol. 5, no. 6. P. 344—348. DOI: 10.1145/367766.368168.

146. Warshall S. A Theorem on Boolean Matrices // Journal of the ACM. 1962. Vol. 9, no. 1. P. 11—12. DOI: 10.1145/321105.321107.

147. Miller R., Stout Q.F. Geometric algorithms for digitized pictures on a mesh-connected computer // IEEE transactions on pattern analysis and machine intelligence. 1985. Vol. PAMI-7, no. 2. P. 216—228. DOI: 10.1109/TPAMI.1985.4767645.

148. Maggs B.M., Plotkin S.A. Minimum-cost spanning tree as a path-finding problem // Information Processing Letters. 1988. Vol. 26, no. 6. P. 291—293. DOI: 10.1016/0020-0190(88)90185-8.

149. Venkataraman G., Sahni S., Mukhopadhyaya S. A Blocked All-Pairs Shortest-Paths Algorithm // ACM Journal of Experimental Algorithmics. 2000. Vol. 8. P. 419—432. DOI: 10.1145/996546.996553.

150. Penner M., Prasanna V.K. Cache-friendly implementations of transitive closure // 2001 International Conference on Parallel Architectures and Compilation Techniques (Barcelona, Spain, Spain, September 8—12, 2001). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2001. P. 185—196. DOI: 10.1109/PACT.2001.953299.

151. Park J., Penner M., Prasanna V.K. Optimizing graph algorithms for improved cache performance // IEEE Transactions on Parallel and Distributed

Systems. 2004. Vol. 15, no. 9. P. 769—782. DOI: 10.1109/TPDS.2004.44.

152. Mowry T.C., Lam M.S., Gupta A. Design and Evaluation of a Compiler Algorithm for Prefetching // SIGPLAN Notices. 1992. Vol. 27, no. 9. P. 62—73. DOI: 10.1145/143371.143488.

153. IBM. IBM Cell Broadband Engine Software Development Kit V3.0 for Multicore Acceleration. URL: `https://arcb.csc.ncsu.edu/~mueller/cluster/ps3/SDK3.0/docs/accessibility/sdkpt/cbet_2cplus_vecttyp.html` (дата обращения: 5.09.2020).

154. Bhattacharjee A., Lustig D., Martonosi M. Architectural and Operating System Support for Virtual Memory. Synthesis Lectures on Computer Architecture // San Rafael, California, USA, Morgan & Claypool Publishers. 2017. 175 p. DOI: 10.2200/S00795ED1V01Y201708CAC042.

155. Lim H., Yew P. A Compiler-Directed Cache Coherence Scheme Using Data Prefetching // 11th International Parallel Processing Symposium (Geneva, Switzerland, April 1—5, 1997). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 1997. P. 643—649. DOI: 10.1109/IPPS.1997.580970.

156. Feautrier P., Lengauer C. Polyhedron Model // Encyclopedia of Parallel Computing. 2011. P. 1581—1592. DOI: 10.1007/978-0-387-09766-4_502.

157. Loechner V., Wilde D.K. Parameterized Polyhedra and Their Vertices // International Journal of Parallel Programming. 1997. Vol. 25, no. 6. P. 525—549. DOI: 10.1023/A:1025117523902.

158. Verdoolaege S. Presburger formulas and polyhedral compilation. 2016. URL: `https://lirias.kuleuven.be/retrieve/361209` (дата обращения: 5.09.2020).

159. Gareev R., Grosser T., Kruse M. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach // ACM Transactions on Architecture and Code Optimization (TACO). 2018. Vol. 15, no. 3. P. 34:1—34:27. DOI: 10.1145/3235029.

160. Акимова Е.Н., Гареев Р.А. Аналитическое моделирование матрично-векторного произведения на многоядерных процессорах // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2020. Т. 9, № 1. С. 69—82. DOI: 10.14529/cmse200105.

161. Akimova E.N., Gareev R.A., Misilov V.E. Analytical Modeling of Matrix-Vector Multiplication on Multicore Processors: Solving Inverse Gravimetry Problem // 2019 International Multi-Conference on Engineering, Computer and Information Sciences (Novovsibirsk, Russia, October 25—27, 2019). Boston, Massachusetts, USA, IEEE Xplore Digital Library, 2019. P. 0823—0827. DOI: 10.1109/SIBIRCON48586.2019.8958103.

162. Bondhugula U., Baskaran M., Krishnamoorthy S., Ramanujam J. et al. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model // Lecture Notes in Computer Science. 2008. Vol. 4959. P. 132—146. DOI: 10.1007/978-3-540-78791-4_9.

163. Bondhugula U., Hartono A., Ramanujam J., Sadayappan P. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer // The 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (Tucson, AZ, USA, June 7—13, 2008). New York, NY, USA, ACM, 2008. P. 101—113. DOI: 10.1145/1375581.1375595.

164. Verdoolaege S. isl: An Integer Set Library for the Polyhedral Model // Lecture Notes in Computer Science. 2010. Vol. 6327. P. 299—302. DOI: 10.1007/978-3-642-15582-6_49.

165. Grosser T., Größlinger A., Lengauer C. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation // Parallel Processing Letters. 2012. Vol. 22, no. 3. DOI: 10.1142/S0129626412500107.

166. Smith T.M., Geijn R. van de, Smelyanskiy M., Hammond J.R. et al. Anatomy of High-Performance Many-Threaded Matrix Multiplication // The 2014 IEEE 28th International Parallel and Distributed Processing Symposium (Phoenix, AZ, USA, May 19—23, 2014). Boston, Massachusetts, USA, IEEE

Xplore Digital Library, 2014. P. 1049—1059. DOI: 10.1109/IPDPS.2014.110.

167. Bull J.M. Measuring Synchronisation and Scheduling Overheads in OpenMP // The 1st European Workshop on OpenMP (Lund, Sweden, September 30—October 1, 1999). 1999. P. 99—105. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.8780&rep=rep1&type=pdf (дата обращения: 5.09.2020).
168. Akimova E.N., Gareev R.A. Algorithm of Automatic Parallelization of Generalized Matrix Multiplication // CEUR Workshop Proceedings. 2017. Vol. 2005. P. 1—10.
169. Нумеров Б.В. Интерпретация гравитационных наблюдений в случае одной контактной поверхности // Доклады Академии наук СССР. Т. 21. 1930. С. 569—574.
170. Vasin V.V., Eremin I.I. Operators and iterative processes of Fejér type: theory and applications // Berlin, Germany, Walter de Gruyter. Vol. 53. 2009. 155 p.
171. Pouchet L.-N. PolyBench/C the Polyhedral Benchmark suite. URL: http://web.cse.ohio-state.edu/~pouchet/software/polybench/ (дата обращения: 5.09.2020).
172. Lehn M. GEMM: From Pure C to SSE Optimized Micro Kernels. URL: http://apfel.mathematik.uni-ulm.de/~lehn/sghpc/gemm/index.html (дата обращения: 5.09.2020).
173. Banerjee K., Georganas E., Kalamkar D., Ziv B. et al. Optimizing Deep Learning RNN Topologies on Intel Architecture // Supercomputing Frontiers and Innovations. 2019. Vol. 6, no. 3. DOI: 10.13140/RG.2.2.18211.50729.
174. Ngxande M., Moorosi N. Development of Beowulf Cluster to Perform Large Datasets Simulations in Educational Institutions // International Journal of Computer Applications. 2014. Vol. 15, no. 15. P. 29–35. DOI: 10.5120/17450-8341.

175. Baumgartner G., Auer A., Bernholdt D.E., Bibireata A. et al. Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models // Proceedings of the IEEE. 2005. Vol. 93, no. 2. P. 276—292. DOI: 10.1109/JPROC.2004.840311.

176. Springer P., Su T., Bientinesi P. HPTT: A High-Performance Tensor Transposition C++ Library // The 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (Barcelona, Spain, June 18, 2017). New York, NY, USA, ACM, 2017. P. 56—62. DOI: 10.1145/3091966.3091968.

Приложение 1. Основные обозначения

$\oplus$	—	операция сложения из замкнутого полукольца $\{S, \oplus, \otimes, 0, 1\}$;
$\otimes$	—	операция умножения из замкнутого полукольца $\{S, \oplus, \otimes, 0, 1\}$;
ГП	—	Гипотетический Процессор;
ПС АОТО	—	Программная Система Автоматической Оптимизации Тензорных Операций;
ARMPL	—	Arm Performance Libraries, библиотека, реализующая интерфейс BLAS;
BLAS	—	Basic Linear Algebra Subprograms, стандарт интерфейса библиотек;
BLIS	—	BLAS-like Library Instantiation Software, библиотека, реализующая интерфейс BLAS;
Clang	—	фронтенд для компиляции;
FMA	—	Fused Multiply-Add, инструкция смешанного умножения и сложения;
GCC	—	GNU Compiler Collection, коллекция компиляторов;
GETT	—	GEMM-like Tensor-Tensor multiplication, метод выполнения свертки тензоров;
IBM's ESSL	—	Engineering and Scientific Subroutine Library, библиотека, реализующая интерфейс BLAS;
ICC	—	Intel C++ Compiler, компилятор;
Intel MKL	—	Intel Math Kernel Library, библиотека, реализующая интерфейс BLAS;
LLVM	—	Low Level Virtual Machine, набор средств для для создания компиляторов;
MMM	—	Matrix-Matrix Multiplication, матричное произведение;
MVM	—	Matrix-Vector Multiplication, матрично-векторное произведение;
OpenBLAS	—	библиотека, реализующая интерфейс BLAS;
OpenMP	—	Open Multi–Processing, технология параллельных вычислений для многоядерных архитектур;
TBLIS	—	Tensor-Based Library Instantiation Software, фреймоврк для оптимзаций сверток тензоров;
TC	—	Tensor Contraction, свертка тензоров;
TCCG	—	Tensor Contraction Code Generator, фреймоврк для оптимзаций сверток тензоров;
VFMA	—	Vector Fused Multiply-Add, векторная инструкция смешанного умножения и сложения;
VMMA	—	Vector Multiply-and-Add инструкция, выполняющая набор невекторных умножений и сложений, составляющих MMA.