МИНОБРНАУКИ РОССИИ
ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО УЧРЕЖДЕНИЯ «ФЕДЕРАЛЬНЫЙ НАУЧНЫЙ ЦЕНТР НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЙ ИНСТИТУТ СИСТЕМНЫХ ИССЛЕДОВАНИЙ РОССИЙСКОЙ АКАДЕМИИ НАУК»
Межведомственный суперкомпьютерный центр Российской академии наук
(МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН)

УТВЕРЖДАЮ
Директор МСЦ РАН –
руководитель ЦКП
вычислительными ресурсами
МСЦ РАН – филиала ФГУ ФНЦ
НИИСИ РАН

Б.М. Шабанов
«26» 12 2019 г.

Методическая документация

«Руководство разработчика по использованию набора инструкций AVX-512»

Руководство рассмотрено и одобрено на заседании Ученого совета МСЦ РАН –
секции Ученого совета ФГУ ФНЦ НИИСИ РАН
«25» 12 2019 г., протокол № 6

Москва 2019

Оглавление

Введение

Суперкомпьютерные технологии в наши дни находят все более широкое применение в различных областях науки, промышленности и бизнеса. Применение имитационного моделирования с использованием суперкомпьютерных расчетов позволяет проводить анализ различных ситуаций и сценариев взаимодействия объектов окружающего мира и получать результаты, недоступные без использования данных средств. При этом постоянно возрастает объем данных, участвующих в суперкомпьютерных расчетах. Размер расчетных сеток в сотни миллионов ячеек уже является обычным для запусков на суперкомпьютерах петафлопсного диапазона производительности, и постепенно появляется потребность в использовании сеток, содержащих миллиарды ячеек [1, 2]. По некоторым оптимистическим прогнозам к 2024 году возможно появление первого экзафлопсного суперкомпьютера, способного выполнять 10^{18} операций с плавающей точкой в секунду [3]. Одновременно с наращиванием вычислительной мощности суперкомпьютеров возникают вопросы об эффективности их применения. В частности ведутся работы по повышению эффективности систем обмена данными между вычислительными узлами [5], по развитию технологий управления расчетными сетками и равномерного распределения вычислений на кластере [6], активно развиваются инструменты языков программирования, направленные на облегчение создания высокопроизводительного параллельного кода [7].

Самым низкоуровневым направлением создания высокопроизводительного параллельного исполняемого кода является векторизация вычислений, позволяющая напрямую задействовать аппаратные возможности вычислителей [9, 10].

В данном документе рассматриваются особенности векторизации программного кода с использованием набора инструкций AVX-512 [11]. Данный набор инструкций имеет ряд уникальных особенностей, позволяющих создавать высокоэффективный параллельный код [12].

Глава 1. Обзор микропроцессоров с поддержкой инструкций AVX-512

В данном разделе рассматриваются основные линейки микропроцессоров Intel, в которых поддержан набор инструкций AVX-512: Intel Xeon Phi Knights Landing, Intel Xeon Skylake SP, Intel Xeon Cascade Lake SP. Микропроцессоры Intel Xeon Phi Knights Mill являются тупиковой ветвью развития микропроцессоров Intel, однако их краткое упоминание также приведено в данном разделе для полноты повествования.

1.1 Intel Xeon Phi Knights Landing

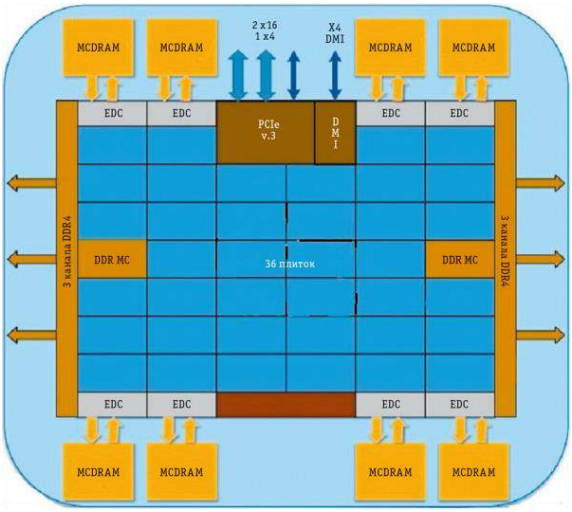

Рис. 1.1. Схема микропроцессора Intel Xeon Phi Knights Landing.

Чипы второго поколения ускорителей Intel Xeon Phi серии x200 (Knights Landing) [13, 14] способны заменить центральные процессоры x86-64, они по сути являются микропроцессорами. Это означает возможность выполнения без перекомпиляции всех имеющихся программ.

Микропроцессоры Intel Xeon Phi Knights Landing относятся к устройствам класса MIC и обладают рядом архитектурных особенностей. Микропроцессоры построены на микросхемах, содержащих до 36 процессорных «плиток» (tile), связанных межсоединением по топологии двумерной решетки (рис. 1.1). Каждая «плитка» содержит два процессорных ядра, специально адаптированных для выполнения приложений систем высокопроизводительных вычислений (HPC). Это ядра Intel Atom Silvermont с двумя VPU – AVX-512 для работы с числами в формате с плавающей запятой двойной точности (DP). В ядрах Atom в Knights Landing много усовершенствований по сравнению с версией для первого поколения процессоров – Intel Xeon Phi Knights Corner (которые являлись ускорителями). Например, добавлена внеочередная обработка команд и модернизирована AVX-архитектура, а благодаря увеличению числа ядер, обеспечивающих выполнение по 16 операций над числами в формате DP на ядро за такт, производительность увеличилась вдвое.

Как и у Knights Corner, каждое ядро KNL имеет кэши команд и данных по 32 KB с дополнительным разделяемым ядрами плитки кэшем второго уровня емкостью 1 MB. Во всей микросхеме обеспечивается когерентность кэша второго уровня для всех ядер с общей емкостью до 36 MB. Каждое ядро предполагает одновременное использование

четырех нитей, или тредов (одновременно выполняемые потоки команд, HyperThreading в процессорах Intel x86-64).

Микросхемы Knights Landing изготовлены по технологии 14 нм и работают на частотах 1.3–1.5 GHz. Пиковую производительность Knights Landing при работе с числами в формате с плавающей запятой можно посчитать, умножив тактовую частоту при работе с AVX на число ядер и на 32 команды, выполняемые за такт, что дает 3 TFLOPS. Кроме базовой тактовой частоты, у Knights Landing возможна и ускоренная – до 1.7 GHz.

В состав микропроцессора входят восемь модулей «ближней памяти» MCDRAM (Multi-Channel DRAM) общей емкостью 16 GB и пропускной способностью 400 GB/s, имеющих доступ к плитке через восемь контроллеров. В Knights Landing есть еще два контроллера для обращения к «дальней памяти» DDR4 2400 емкостью до 384 GB и пропускной способностью 90 GB/s. «Ближняя память» может работать в трех разных режимах: как кэш дальней памяти; в составе единого адресного пространства с дальней памятью (гладкий режим); в комбинированном режиме, когда часть MCDRAM используется как кэш, а часть – в едином адресном пространстве с DDR4. Уникальным для микропроцессора является не только такое двухуровневое построение памяти, но и поддержка интерфейса с межсоединением Intel Omni-Path.

1.2 Intel Xeon Phi Knights Mill

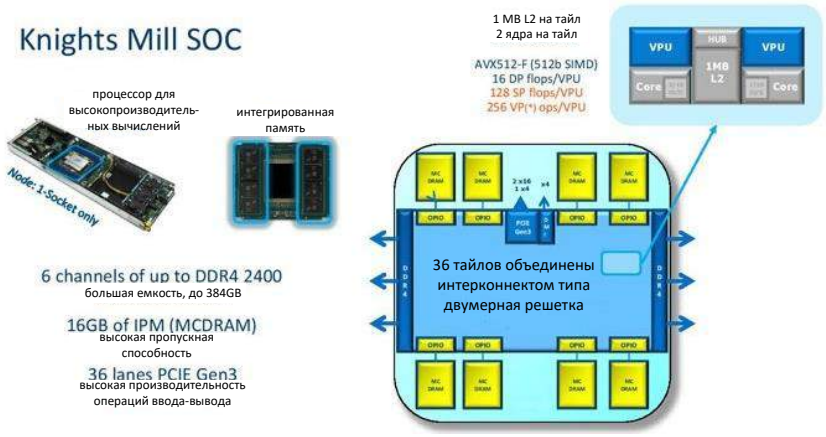

Рис. 1.2. Схема микропроцессора Intel Xeon Phi Knights Mill.

Микропроцессоры Intel Xeon Phi Knights Mill [15, 16, 17] были предназначены для ускорения задач глубокого обучения. Всего опубликованы спецификации трех моделей: Xeon Phi Knights Mill 7235, 7285 и 7295. Их базовые частоты составляют соответственно 1.3, 1.3 и 1.5 GHz, повышенные – 1.4, 1.4 и 1.6 GHz.

Модель Xeon Phi KNM 7235 включает 64 ядра и 32 MB кэш-памяти второго уровня, 7285 – 68 ядер и 34 MB, 7295 – 72 ядра и 36 MB. Все чипы получили интегрированные шестиканальные контроллеры памяти: DDR4-2133 – у модели Xeon Phi 7235, DDR4-2400 – у остальных.

Конфигурация, состав и архитектура Knights Mill практически полностью копирует строение Knights Landing, но у них сделан акцент на вычисления с одинарной точностью (этот параметр вырос в два раза) и с переменной точностью (рост производительности в 4 раза), а также добавлен набор новых инструкция AVX-512 (рис. 1.2). Иными словами, процессоры/ускорители Knights Mill разрабатывались для ниши платформ для машинного

обучения и должны были составить конкуренцию актуальным ускорителям NVIDIA и AMD. Производительность вычислений с двойной точностью при этом упала в два раза, но этого следовало ожидать, поскольку Intel просто изменили блоки FP64 в пользу FP32 и VP.

Микропроцессоры Intel Xeon Phi Knights Mill распространения не получили, они являются завершением линейки Intel Xeon Phi.

1.3 Intel Xeon Skylake SP

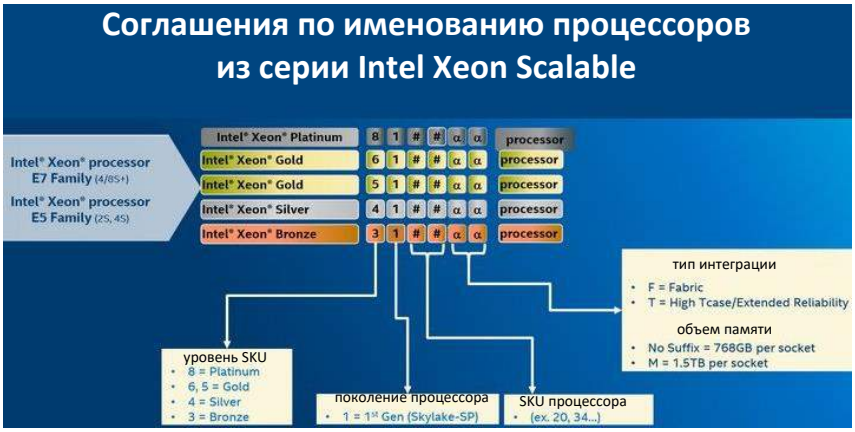

Рис. 1.3. Схема наименований микропроцессоров Intel Xeon Skylake SP.

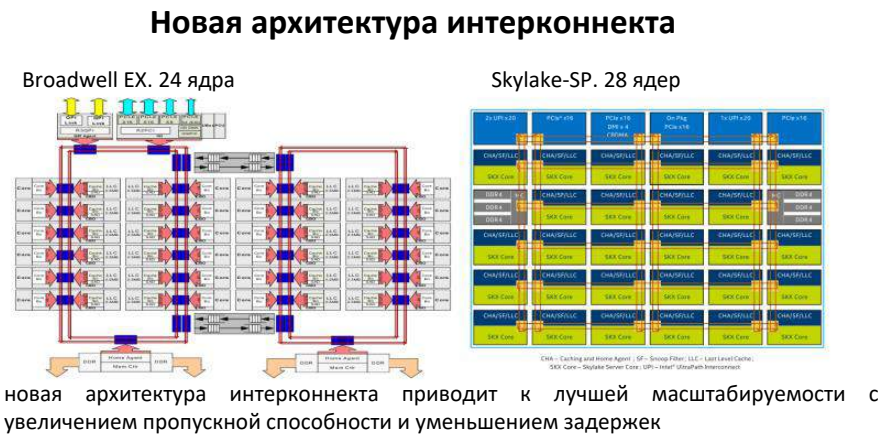

Рис. 1.4. Архитектура соединений между ядрами Intel Xeon Skylake SP.

Серверные процессоры Intel Xeon Skylake SP [18, 19, 20, 21], рассчитанные на использование в многопроцессорных системах, делятся на линейки Xeon Platinum, Xeon Gold, Xeon Silver и Xeon Bronze, вместо Xeon E7 и Xeon E5.

Что касается названий самих моделей процессоров, то первая цифра в них означает принадлежность к линейке (8 – Platinum, 6 и 5 – Gold, 4 – Silver и 3 – Bronze), вторая – поколение (1 – Skylake SP), а третья и четвёртая — модельный номер процессора. Если

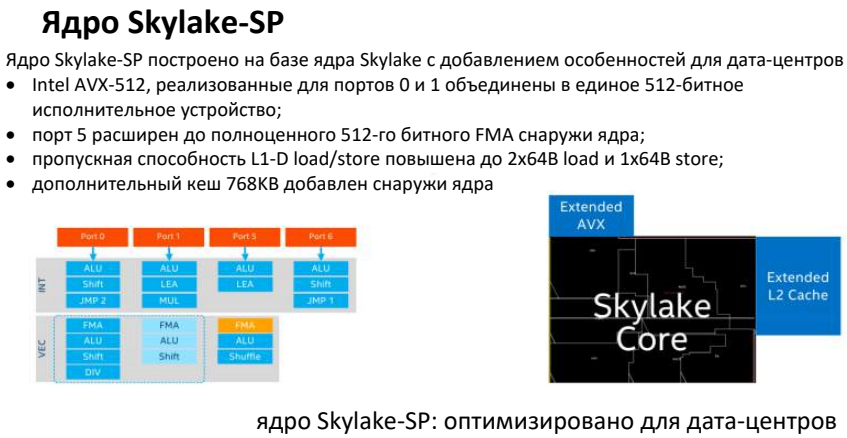

Рис. 1.5. Схема ядра Intel Xeon Skylake SP.

Рис. 1.6. Улучшения микроархитектуры Intel Xeon Skylake SP.

после четырёх цифр буквы отсутствуют, то процессор поддерживает до 768 GB оперативной памяти, а буква "M"означает поддержку до 1.5 TB памяти. Буква "T"указывает на использования специальной упаковки и увеличенный срок службы, буква "F"обозначает наличие встроенной шины OmniPath (рис. 1.3).

Старшая линейка Xeon Platinum включает сразу шестнадцать моделей большая часть из которых имеет от 24 до 28 физических ядер (рис. 1.4). Данные процессоры могут использоваться в системах, обладающих сразу восемью процессорными разъёмами. Эти процессоры поддерживают HyperThreading, имеют 48 линий PCI-Express и шесть каналов памяти DDR4-2666 и по два FMA с Intel AVX-512 на ядро.

Линейка Xeon Gold разделена на две группы: Gold 6100 и Gold 5100, представителей которых отличает друг от друга количество подключений UPI (три и два, соответственно), поддерживаемая частота памяти DDR4 (2666 против 2400) и количество FMA с AVX-512 на ядро (два и один, соответственно). И те и те поддерживают системы с двумя или четырьмя процессорами. Всего линейка Xeon Gold включает 32 процессора, имеющих от 4 до 22 ядер с поддержкой HyperThreading.

Линейки Xeon Silver и Xeon Bronze в сумме насчитывают десять процессоров, с коли-

Рис. 1.7. Поддержка AVX-512 в микропроцессорах Intel Xeon Skylake SP.

чеством ядер от 4 до 12. Эти новинки имеют один FMA с AVX-512 на ядро. Процессоры Silver поддерживаются DDR4-2400 и HyperThreading. В свою очередь модели Bronze используют DDR4-2133, не поддерживают HyperThreading, и имеют более медленный UPI.

Базовая структура каждого ядра Xeon Skylake-SP осталась ровной той же, что и у обычных Skylake. Однако присутствует пара изменений. Во-первых, объём L2-кеша достигает 1 MB путём прибавки 768 KB, которые фактически находятся не в самом ядре, а рядом. L3-кеш имеет «некруглый» объём 1.375 MB на ядро. Кеш неинклюзивный — L2 заполняется напрямую из RAM, и лишь затем ненужные линии вытесняются в L3. Общие для нескольких ядер данные хранятся в L3. При этом есть один неявный нюанс – объём L3 зависит не только от числа ядер. У некоторых моделей есть отдельное упоминание о его размере. Например, L3-кешем на 24.75 MB могут оснащаться модели с 8 и 12 ядрами. Ещё более существенный разрыв виден у вариантов на 6 и 12 ядер, которые, тем не менее, имеют кеш на 19.25 MB. Но в общем и целом основной упор сделан именно на работу с кешем L2.

Во-вторых, к ядру дополнительно «прилеплен» блок AVX-512. Собственно, Port0 и 1 на пару могут выполнять одну такую инструкцию, плюс Port5 сам по себе умеет работать с одной AVX-512/FMA (рис. 1.5 и рис. 1.6). Из инструкций AVX-512 поддержаны подмножества AVX-512 F, AVX-512 VL, AVX-512 BW, AVX-512 DQ, AVX-512 CD (рис. 1.7). Кроме того, базовая и турбочастота каждого ядра разнится и зависит от того, какой тип инструкций сейчас исполняется – с тяжёлыми AVX2/AVX-512 или без них.

1.4 Intel Xeon Cascade Lake SP

Общая схема наименований и серии Platinum, Gold, Silver, Bronze в линейке Intel Xeon Cascade Lake SP [22, 23, 24, 25, 26] остались прежними, но "суффиксов"стало больше. Уже имеющиеся L и M всё так же указывают на поддержку увеличенного объёма памяти – до 4.5 и 2 TB вместо базовых 1.5 TB соответственно. Варианты T для процессоров с расширенным сроком поддержки, готовых к работе в более жёстких условиях, тоже сохранились. Новыми для Xeon стали версии Y, N, V и S – все они являются вариациями процессоров с поддержкой Speed Select. Наиболее общий и универсальный вариант – это Y. Версии N и S заранее оптимизированы для работы с сетевыми приложениями и с базами данных соответственно. Версия V нацелена на плотную виртуализацию. Версий F с интегрированным модулем Intel Omni-Path теперь нет (хотя и те, что были, всё равно

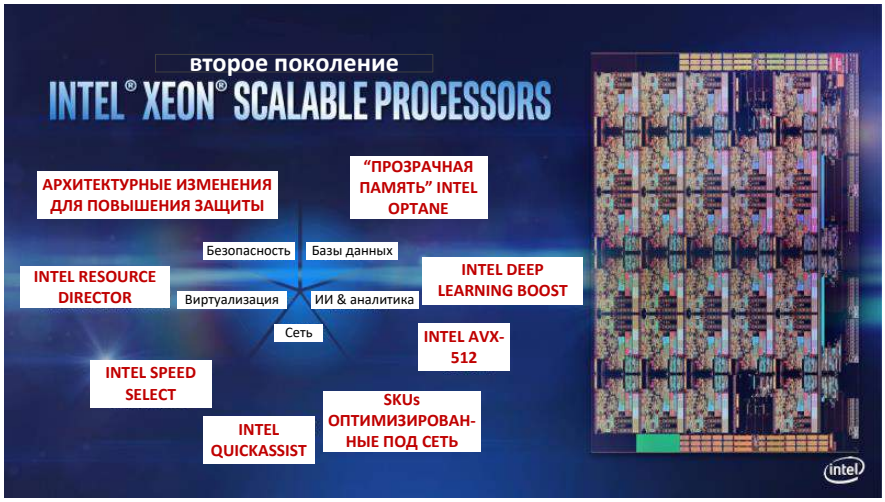

Рис. 1.8. Второе поколение Intel Scalable Processors.

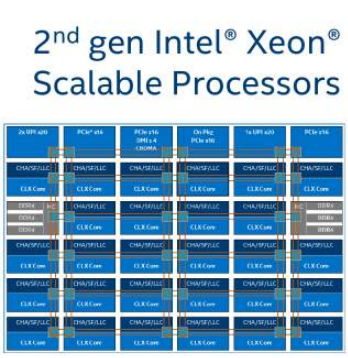

Схема 28-ядерного процессора

Features	2nd gen Intel® Xeon® Scalable Processors
Ядра и потоки на процессор	[8200] Up to 28 cores and 56 threads **[9200] Up to 56 cores and 112 threads**
Иерархия памяти	1MB dedicated L2 Cache per Core Up to 38.5 MB (non-inclusive) Shared L3 Cache
Межпроцессорное соединение	Up to 3x Intel® UPI @ 10.4 GT/s Up to 8 socket glueless connectivity
PCIe Lanes	Up to 48 Lanes PCIe 3.0 (2.5, 5, 8 GT/s)
Память	Up to 6 channels **at 2933 MT/s** per processor Up to 2 DPC RDIMMs, LRDIMMs, 3DS LRDIMMs, **supporting up to 16Gb DDR4 devices** **Intel® Optane™ DC Persistent Memory Module support for up to 4.5TB system memory per processor**
Векторные вычисления	Intel® AVX-512 with up to 16 DP, 32 SP, **and 128 INT8 MACs w/ Intel® DL Boost** per cycle per core
Устранение уязвимостей	**Variants 2, 3, 3a, 4, and L1TF**
Турбо режим	**Boost across Stack**

Рис. 1.9. Изменения во втором поколении архитектуры Scalable.

нельзя было просто так купить).

К традиционным уже ограничениям на число сокетов, каналов UPI, одновременно выполняемых AVX/FMA-инструкций и частот памяти для процессоров Bronze и Silver добавилось ещё одно – они не умеют работать с Intel Optane DC Persistent Memory. Вообще говоря, на уровне общей архитектуры и техпроцесса Cascade Lake SP мало чем отличаются от прошлых Skylake SP. Их следует рассматривать как очередной этап оптимизации Skylake в целом или, если хотите, работу над ошибками и над реализацией тех задумок, которые изначально должны были быть в процессорах первого поколения, но по тем или иным причинам до стадии производства не дошли (рис. 1.8).

В целом различных микроапдейтов не один десяток, но из более-менее общих и заметных внешнему наблюдателю выделяются два. Во-первых, в среднем частоты стали выше при сохранении прежнего TDP, то есть вычисления стали «дешевле». В среднем заявленный прирост для популярной серии Gold составляет около трети, но он не совсем уж равномерен по классам и задачам, хотя та или иная прибавка есть везде. Во-вторых, появилась поддержка DDR4-2933 с 16-Гбит чипами, то есть с типовым объёмом модуля

64-256 GB. Но для конфигураций с двумя DIMM на канал частота всё равно снижается до привычных 2666 MT/s.

В остальном базовые характеристики и набор возможностей остались прежними. Число ядер и объёмы кешей не поменялись: до 28 и по 1 MB L2 на ядро + до 38.5 MB общего L3. Число и тип линий PCI-E тоже такие же, какие и были – 48 линий версии 3.0. Масштабируемость не изменилась: до 3 линий UPI на 10,4 GT/s и до 8 (бесшовно) сокетов в системе (рис. 1.9).

В Cascade Lake появились первые заплатки против уязвимостей, в частности Spectre, Spectre NG, а также против L1TF (Foreshadow).

Глава 2. Классификация и описание инструкций AVX-512

В данном разделе приведена классификация и описание основных инструкций AVX-512, поддерживаемых в линейках микропроцессоров Intel Xeon Phi KNL, Intel Xeon Skyalake SP, Intel Xeon Cascade Lake SP. При описании функционала AVX-512 основным источником являлась последняя доступная на момент написания данного документа версия [11].

2.1 Окружение

Набор инструкций AVX-512 является естественным расширением наборов AVX и AVX2. Инструкции из этого набора используют для своей работы 32 zmm регистра, имеющих размер 512 бит (zmm0 – zmm31, эти регистры являются расширением регистров ymm0 – ymm31), а также 8 специальных масочных регистров, позволяющих обрабатывать элементы векторов по-отдельности (k0 – k7) (рис. 2.1).

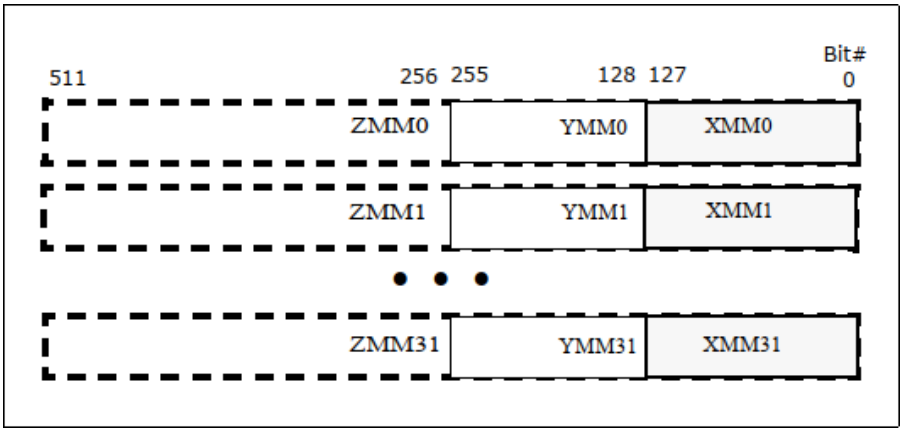

Рис. 2.1. Регистры zmm являются расширениями регистров ymm.

2.2 Классификация набора инструкций AVX-512

Набор инструкций AVX-512 делится на несколько подмножеств. В данном руководстве упоминаются только те подмножества, которые поддержаны в поколении микропроцессоров Phi Knights Landing, Skylake SP и Cascade Lake SP (рис. 2.2).

AVX-512 F – Foundation – Основные инструкции работающие с 32-битными и 64-битными данными. Сюда включаются поэлементные операции над векторами, операции с масками, слияния векторов по маске, сравнение векторов, операции перестановок элементов векторов, операции конвертации и другие.

AVX-512 CD – Conflict Detection – Набор операций, предназначенный для разрешения конфликтов при векторизации циклов. Основная инструкция из этого набора VPCONFLICT позволяет сравнить каждый элемент первого вектора с каждым элементом второго вектора. Данная инструкция применяется для динамической проверки диапазонов адресов на предмет конфликтов.

AVX-512 ER – Exponential and Reciprocal – Набор инструкций для поэлементного вычисления с повышенной точностью функций 2^x, $1/x$, $1/\sqrt{(x)}$.

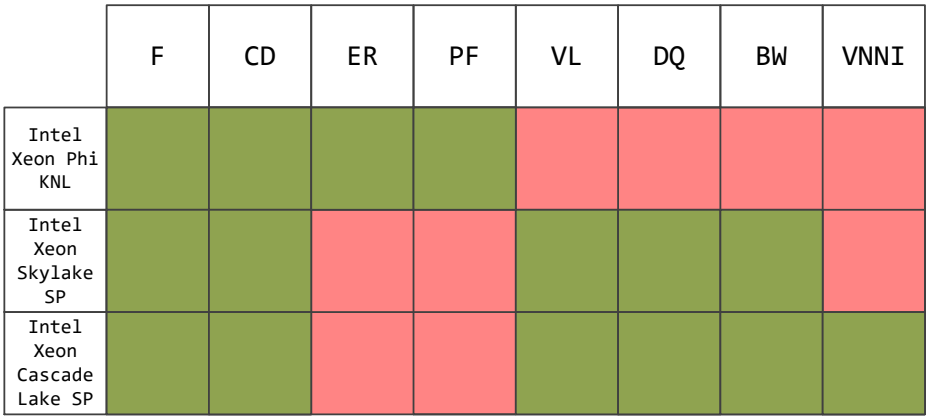

Рис. 2.2. Схема поддержки поднаборов AVX-512 в микропроцессорах Intel разных поколений.

AVX-512 PF – Prefetch – Инструкции предварительной подкачки данных для операций чтения и записи по адресам с произвольными смещениями (gather/scatter).

AVX-512 VL – Vector Length – Расширение многих инструкций на элементы данных размера 128 и 256 бит.

AVX-512 DQ – Doubleword and Quadword – Содержит дополнительные инструкции для работы с элементами данных размера 32 и 64 бита.

AVX-512 BW – Byte and Word – Расширение набора инструкции для работы с данными размера 8 и 16 бит.

AVX-512 VNNI – Vector Neural Network Instructions – Дополнительные инструкции, введенные для оптимизации алгоритмов машинного обучения.

2.3 Описание основных типов инструкций AVX-512

В данном разделе приводится классификация инструкций из набора AVX-512 по логике их работы. Данная классификация не связана с разбиением их по подмножествам.

2.3.1 Операции с масками

Примеры операций: KMOVB, KMOVW, KMOVD, KMOVQ, KNOTB, KNOTW, KNOTD, KNOTQ, KANDB, KANDW, KANDD, KANDQ, KANDNB, KANDNW, KANDND, KANDNQ, KORB, KORW, KORD, KORQ, KXNORB, KXNORW, KXNORD, KXNORQ, KXORB, KXORW, KXORD, KXORQ, KTESTB, KTESTW, KTESTD, KTESTQ, KORTESTB, KORTESTW, KORTESTD, KORTESTQ, KADDB, KADDW, KADDD, KADDQ, KSHIFTLB, KSHIFTLW, KSHIFTLD, KSHIFTLQ, KSHIFTRB, KSHIFTRW, KSHIFTRD, KSHIFTRQ, KUNPCKBW, KUNPCKWD, KUNPCKDQ.

Для поддержки выборочного применения операций над упакованными данными к конкретным элементам векторов используются маски. Большинство инструкций из набора AVX-512 могут использовать специальные регистры масок. Всего таких регистров 8 (k0 – k7). Длина каждой маски составляет 64 бита. Маски k1 – k7 используются в командах для осуществления условной операции над элементами упакованных данных (если соответствующий бит выставлен в 1, то операция осуществляется) или для слияния элементов данных в регистр назначения. Маска k0 является константой, все ее биты выставлены в 1. Также маски могут использоваться для выборочного чтения из памяти и записи в

память элементов векторов, для аккумулирования результатов логических операций над элементами векторов.

Для комбинирования различных масок AVX-512 поддерживает набор операций для работы с масками элементов данных различных размеров (byte, word, doubleword, quadword). В число этих операций входят пересылки, побитовые логические операции, сложение масок, операции сдвигов и операции UNPACK (слияние двух масок с чередованием битов).

2.3.2 Упакованные операции с одним операндом zmm и результатом zmm

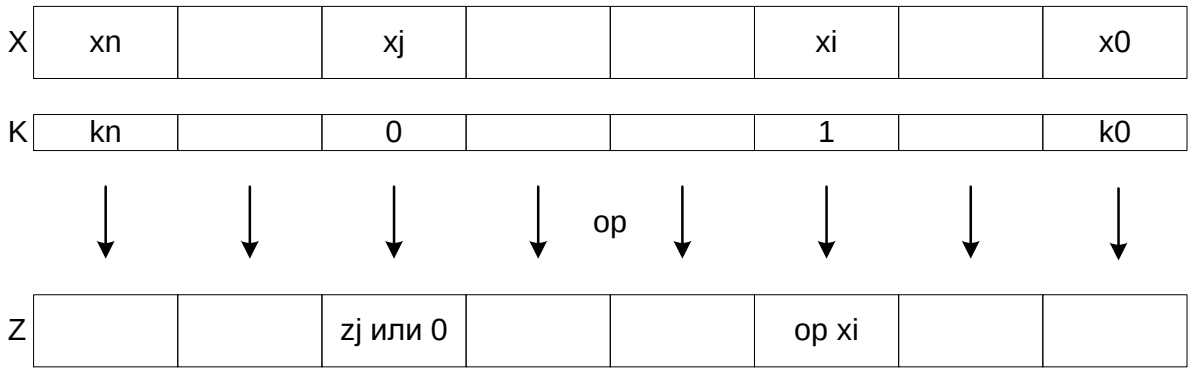

Рис. 2.3. Схема работы упакованной операции с одним операндом ZMM (X) и результатом ZMM (Z) с использованием маски K.

Примеры инструкций: VPABSD, VPABSQ, VSQRTPD, VSQRTPS, VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ, VPROLD, VPROLQ, VPRORD, VPRORQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VPABSB, VPABSW, VPLZCNTD, VPLZCNTQ, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VREDUCEPD, VREDUCEPS, VEXP2PD, VEXP2PS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS, VPSLLDQ, VPSLLW, VPSRAW, VPSRLDQ, VPSRLW, VRNDSCALEPD, VRNDSCALEPS.

Операция из данной группы получает на вход вектор и применяет к каждому его элементу конкретную функцию, получая результат того же размера. Также операция получает на вход маску. Результат применения функции к элементу входного вектора записывается в выходной вектор только если соответствующий бит маски выставлен в 1. В противном случае возможно обнуление или игнорирование соответствующего элемента выходного вектора (рис. 2.3).

В качестве функций, применяемых к элементам векторов можно выделить получение абсолютного значения, извлечение корня, операции сдвигов и вращений, получение обратных величин и их квадратов, подсчет количества ведущий нулей, извлечение экспоненты и мантиссы, редукцию, вычисление экспоненты, округление.

2.3.3 Упакованные операции с двумя операндами zmm и одним результатом zmm

Примеры инструкций: VADDPD, VADDPS, VDIVPD, VDIVPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VPADDD, VPADDQ, VPANDD, VPANDQ, VPANDND, VPANDNQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPORD, VPORQ, VPSUBD, VPSUBQ, VPXORD, VPXORQ, VSUBPD, VSUBPS, VPROLVD, VPROLVQ, VPRORVD, VPRORVQ, VPSLLVD,

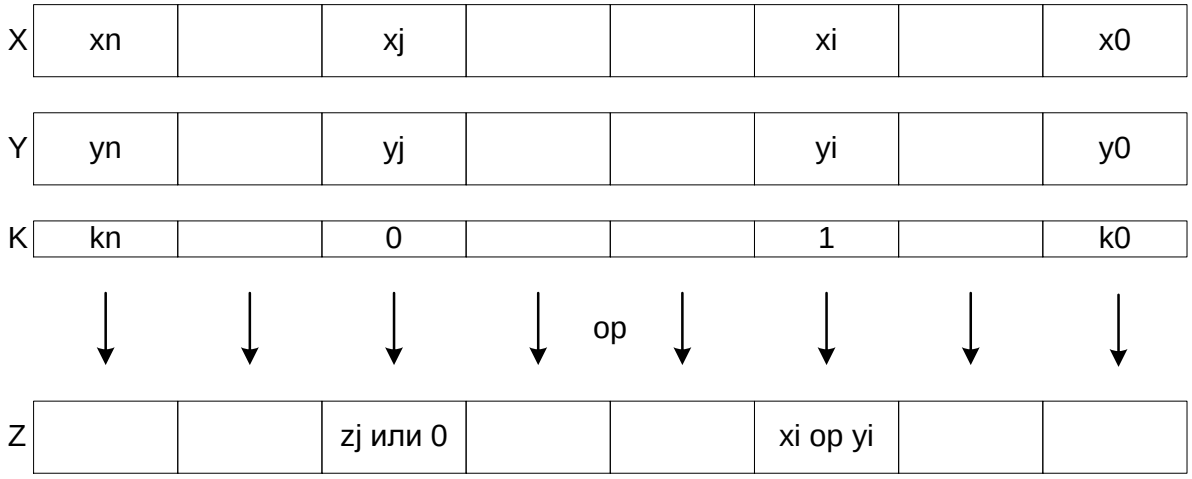

Рис. 2.4. Схема работы упакованной операции с двумя операндами ZMM (X, Y) и результатом ZMM (Z) с использованием маски K.

VPSLLVQ, VPSRAVD, VPSRAVQ, VPSRLVD, VPSRLVQ, VXORPD, VXORPS, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPMULDQ, VPMULLD, VPMULUDQ, VANDPD, VANDPS, VANDNPD, VANDNPS, VORPD, VORPS, VPSLLVW, VPSRAVW, VPSRLVW, VPMULLQ, VRANGEPD, VRANGEPS, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW, VPMULLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPSUBUSB, VPSUBUSW, VFIXUPIMMPD, VFIXUPIMMPS, VSCALEFPD, VSCALEFPS.

Операции из данной группы схожи с операциями из предыдущего пункта, однако вместо одного входного вектора здесь имеется два. Функция получения результата, соответственно, вместо одного аргумента принимает два (рис. 2.4). Для записи результатов функции в выходной вектор также используется маска. Из числа функций над элементами векторов можно отметить операции сложения, вычитания, умножения, деления, получение максимального, минимального и среднего значения, логические побитовые операции, операции сдвигов и вращений переменное количество разрядов.

2.3.4 Упакованные операции с двумя операндами zmm и результатом маской

Примеры операций: VCMPPD, VCMPPS, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPCMPB, VPCMPUB, VPCMPW, VPCMPUW, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VPCMPD, VPCMPUD, VPCMPQ, VPCMPUQ, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW.

Операция из данной группы получает на вход два вектора и применяет к каждой паре элементов логическую функцию. Инструкция использует две маски. Одна маска используется для записи результатов, а вторая регулирует, какие результаты должны быть записаны в эту результирующую маску (рис. 2.5). В данную группу входят операции сравнения и теста (сравнение с нулем результата операции логического умножения) упакованных данных.

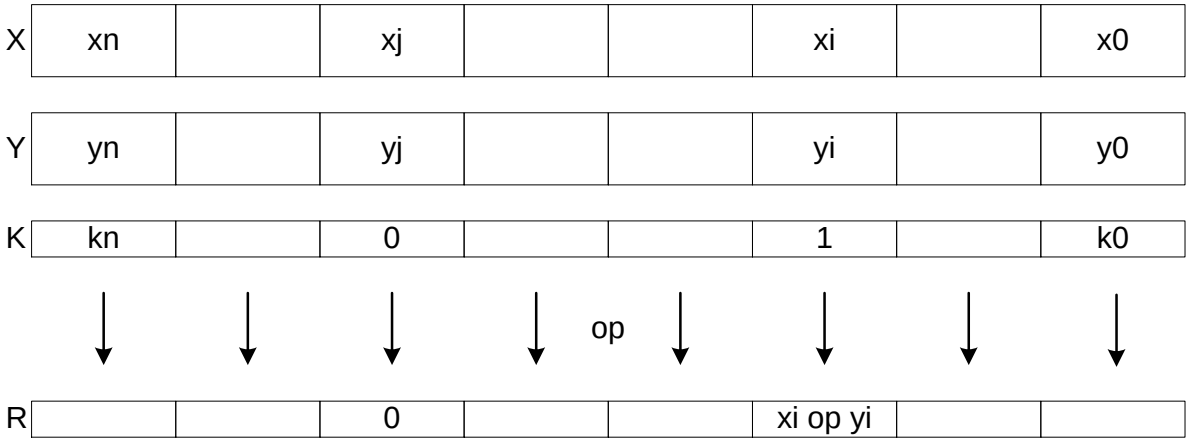

Рис. 2.5. Схема работы операции с двумя операндами ZMM (X, Y) и результатом маской R с использованием маски K

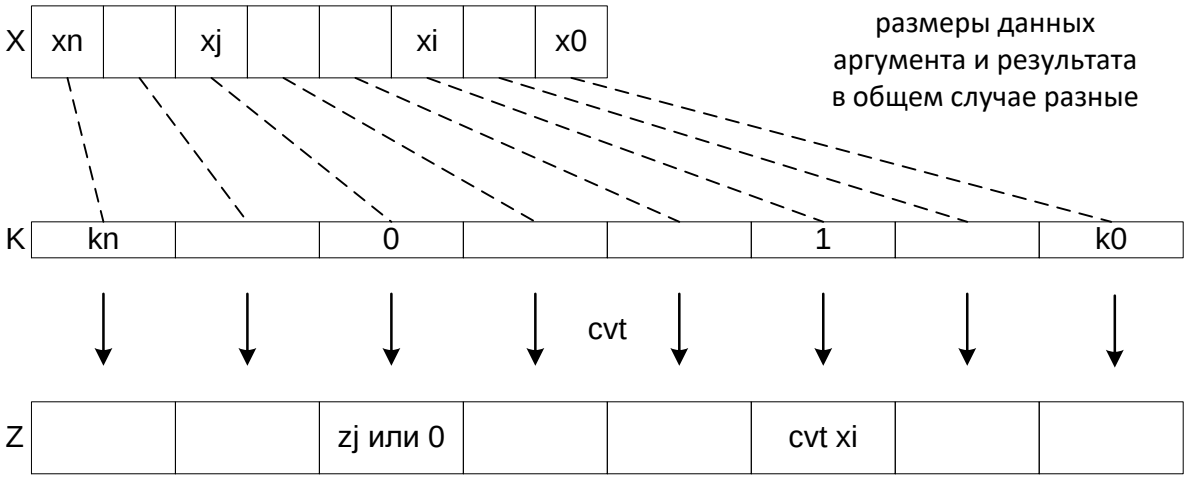

Рис. 2.6. Схема работы операции производящий конвертацию каждого элемента операнда X в тип элемента аргумента Z с использованием маски K.

2.3.5 Операции конвертации

Примеры операций: VCVTDQ2PD, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPS2DQ, VCVTPS2PD, VCVTTPD2DQ, VCVTTPS2DQ, VCVTPD2UDQ, VCVTPH2PS, VCVTPS2PH, VCVTPS2UDQ, VCVTTPD2UDQ, VCVTTPS2UDQ, VCVTUDQ2PD, VCVTUDQ2PS, VCVTPD2QQ, VCVTPD2UQQ, VCVTPS2QQ, VCVTPS2UQQ, VCVTQQ2PD, VCVTQQ2PS, VCVTTPD2QQ, VCVTTPD2UQQ, VCVTTPS2QQ, VCVTTPS2UQQ, VCVTUQQ2PD, VCVTUQQ2PS, VPACKSSWB, VPACKSSDW, VPACKUSDW, VPACKUSWB.

Операции данной группы разделим на две части. Во-первых это операции, которые выполняют конвертирование элементов одного входного вектора (преобразование данных из одного формата в другой). При этом возможны изменения размера данных. Результат записывается в выходной вектор по маске (рис. 2.6). Среди операций конвертации можно выделить операции с использованием вещественных значений половинной точности (16-битные значения) и конвертацию между вещественными и целочисленными значениями.

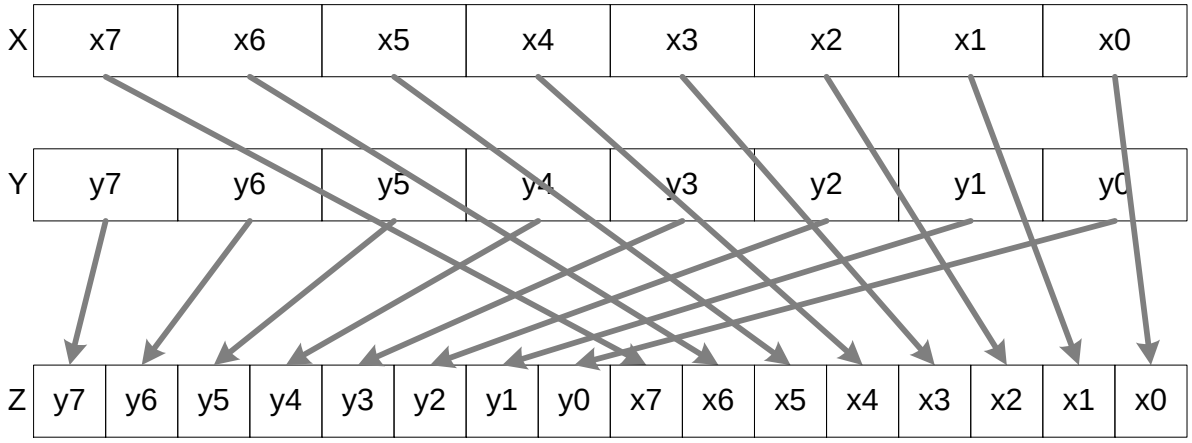

Рис. 2.7. Схема работы операции PACK, выполняющей упаковку элементов двух векторов в один вектор с конвертацией элементов аргументов в тип половинного размера.

Во вторую группу поместим операции PACK, которые осуществляют конвертацию частей аргументов до половинного размера и упаковывают результаты в один выходной регистр. Для простоты использование маски на схеме ниже опущено (рис. 2.7). Таких операций всего четыре: конвертация из doubleword в word и из word в byte (для каждого вида конвертации предусмотрена работа со знаковыми и беззнаковыми значениями).

2.3.6 Упакованные комбинированные операции

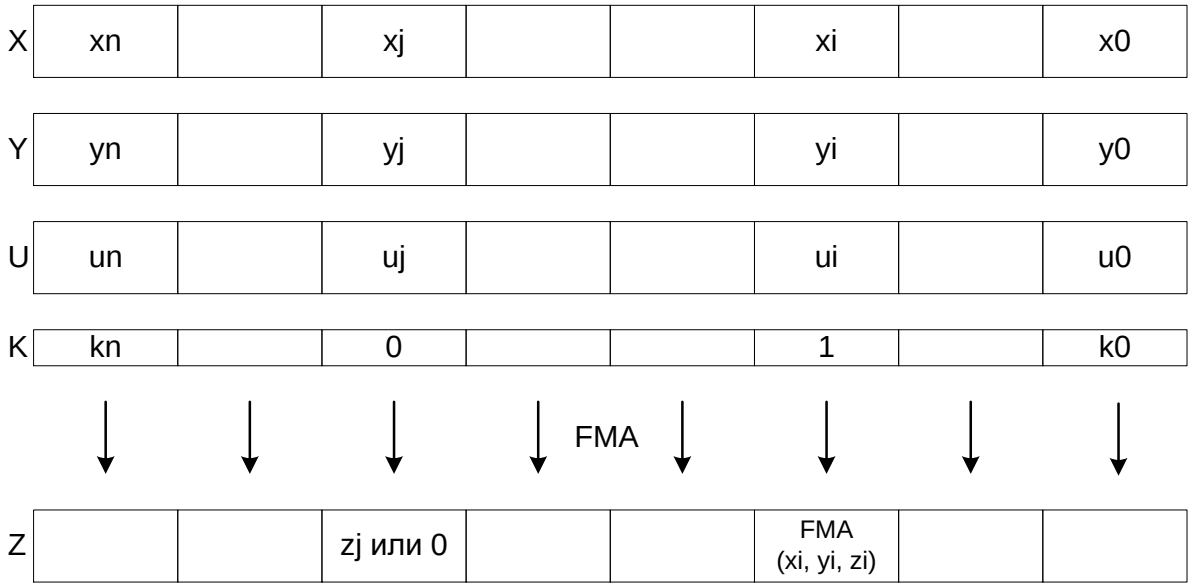

Рис. 2.8. Схема работы комбинированной операции, выполняющей умножение и сложение/вычитание для каждой тройки соответствующих элементов аргументов X, Y, U с использованием маски K и записью результата в вектор Z.

Примеры операций: VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSUB213PD,

VFMADDSUB231PD, VFMADDSUB132PD, VFMADDSUB213PS, VFMADDSUB231PS, VFMADDSUB132PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS, VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS.

Комбинированные трехаргументные операции выполняют две последовательные арифметические операции. Первая из этих операций использует два из поданных аргументов, а вторая – третий аргумент и результат первой операции (рис. 2.8). Такой подход можно встретить, например, в архитектуре «Эльбрус», однако в упакованных комбинированных операциях пошли дальше. Из особенностей реализации Intel можно выделить три существенных улучшения. Кроме очевидных первых двух (во-первых, команды векторные, а во-вторых, результат записывается по маске) можно заметить, что команды MADDSUB*, MSUBADD* объединяют в себе выполнение двух разных цепочек над элементами вектора.

2.3.7 Операции перестановок

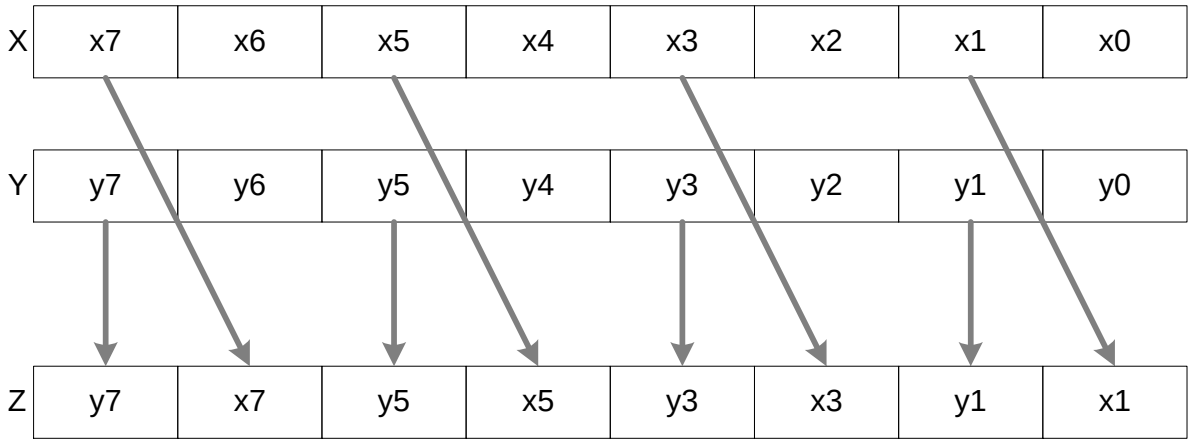

Рис. 2.9. Схема работы операции UNPACK.

Примеры операций: VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS, VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD , VPSHUFD, VSHUFPD, VSHUFPS, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2, VPSHUFB, VPSHUFHW, VPSHUFLW, VALIGND, VALIGNQ, VPALIGNR, VBLENDMPD, VBLENDMPS, VPBLENDMD, VPBLENDMQ, VPBLENDMB, VPBLENDMW, VPERMD, VPERMI2D, VPERMI2Q, VPERMI2PS, VPERMI2PD, VPERMILPD, VPERMILPS, VPERMPD, VPERMPS, VPERMQ, VPERMT2D, VPERMT2Q, VPERMT2PS, VPERMT2PD, VPERMW, VPERMI2W, VPERMT2W.

Операции перестановок представляют обширное и довольно слабо структурируемое множество команд. Также одного и того же эффекта можно добиться с помощью применения различных операций перестановок. С другой стороны определенные схемы перестановок элементов могут быть реализованы только с помощью последовательности команд.

Операции UNPACK принимают на вход два операнда упакованных данных, выбирают из них верхние или нижние части определенного размера и размещают в результирующем операнде с чередованием (рис. 2.9).

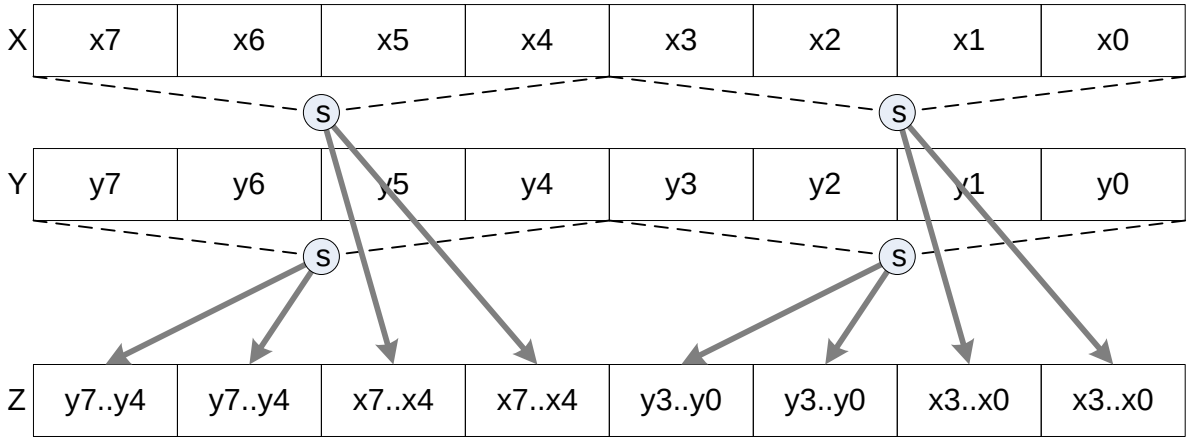

Рис. 2.10. Схема работы одного из типов операции SHUFFLE.

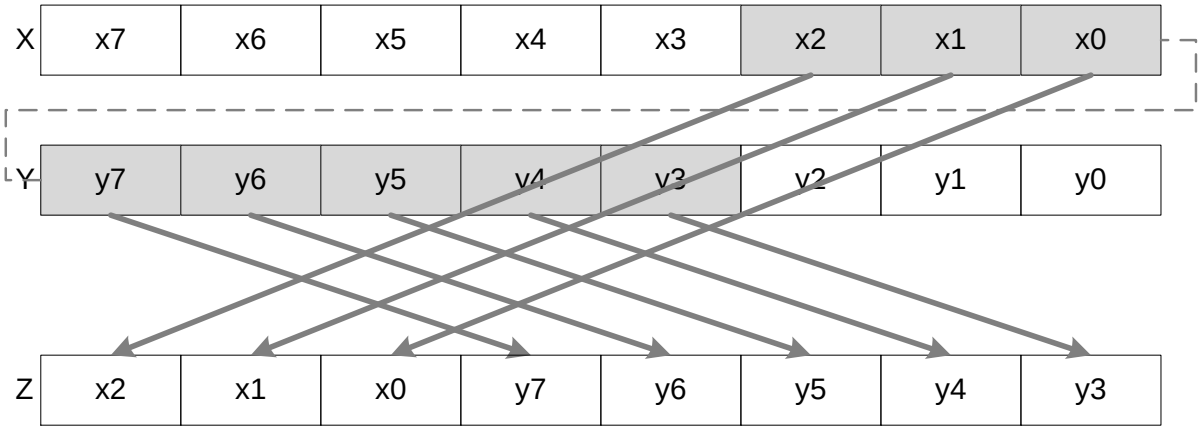

Рис. 2.11. Схема работы операции ALIGN, выполняющей выравниванием вектора по произвольному индексу элемента.

Операции SHUFFLE осуществляют перемешивание частей одного или двух операндов (порядок перемешивания закодирован в дополнительном операнде команды), а затем в случае двух операндов производят слияние результатов с чередованием (возможны варианты один через один и два через два) (рис. 2.10).

Операции ALIGN выполняют конкатенацию двух аргументов, после чего сдвигают полученный аргумент вправо на количество бит согласно грануляции команды и записывают нижнюю часть результата в выходной регистр (рис. 2.11).

Операции BLEND выполняют слияние частей двух операндов, опираясь на значения соответствующих битов маски. В результате в элемент результирующего вектора может попасть либо соответствующий элемент одного из аргументов, либо 0 (рис. 2.12).

Операции PERM принимают на вход один или два источника и переставляют в них элементы, опираясь на индексы, которые также подаются на вход. После этого два результата соединяются по логике, схожей с командами BLEND, либо могут выбираться старшие и младшие части соответствующих элементов данных (рис. 2.13). Предикатом при слиянии результатов по логике BLEND служит один бит индекса, а конечный результат записывается по маске.

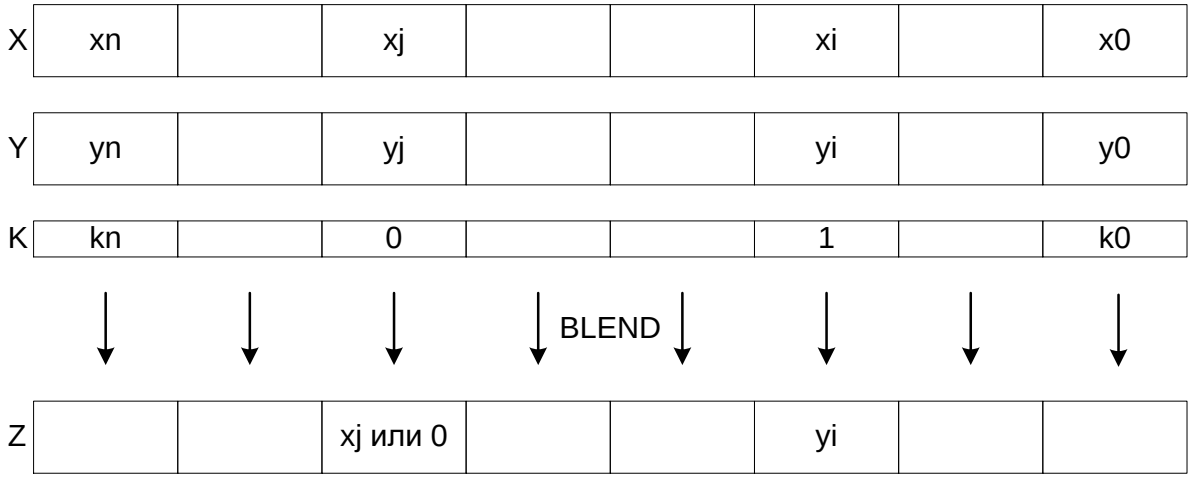

Рис. 2.12. Схема работы операции слияния BLEND элементов двух векторов.

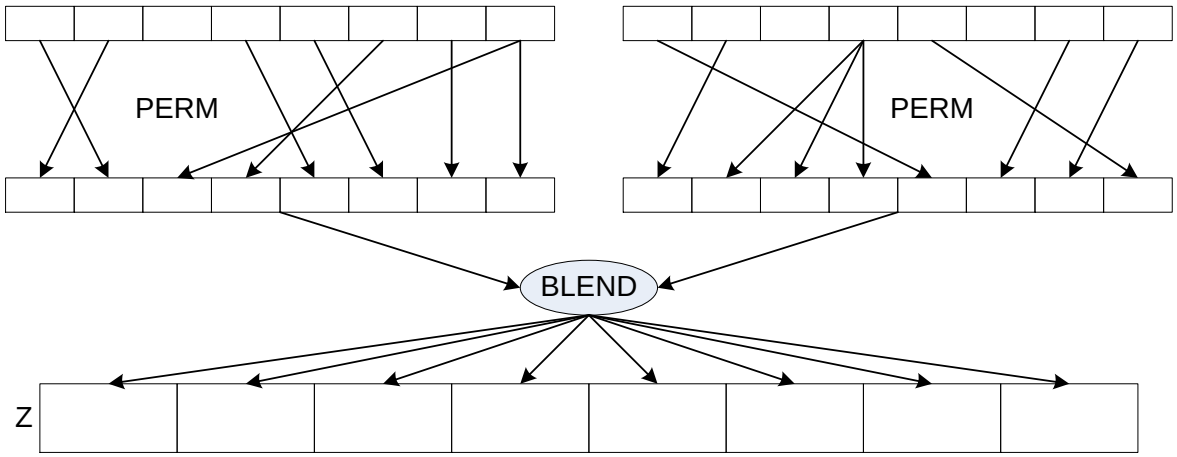

Рис. 2.13. Схема работы одного из типов операции PERM.

2.3.8 Простые операции пересылок

Примеры операций: VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64, VMOVDQU32, VMOVDQU64, VMOVNTDQA, VMOVNTDQ, VMOVNTPD, VMOVNTPS, VMOVUPD, VMOVUPS, VPMOVSXBD, VPMOVSXBQ, VPMOVSXWD, VPMOVSXWQ, VPMOVSXDQ, VPMOVZXBD, VPMOVZXWD, VPMOVZXDQ, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVD2M, VPMOVQ2M, VPMOVM2D, VPMOVM2Q, VMOVDQU8, VMOVDQU16, VPMOVSXBW, VPMOVZXBW, VPMOVB2M, VPMOVW2M, VPMOVM2B, VPMOVM2W, VPMOVWB, VPMOVSWB, VPMOVUSWB.

На данной группе инструкций нет смысла останавливаться подробно. Операции из этого множества выполняют простые пересылки последовательных данных между разными регистрами, а также между памятью и регистром. В данную категорию попадают также операции пересылки со знаковым расширением и с расширением нулем. Также сюда включены пересылки между регистрами данных и масками (*2M, M2* операции).

2.3.9 Операции пересылок с дублированием и выбором элементов

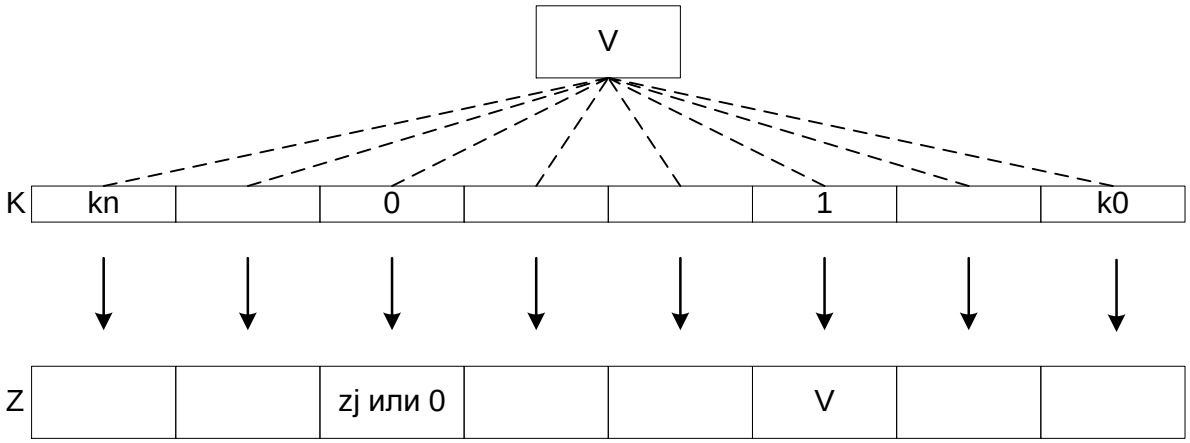

Рис. 2.14. Схема работы операции BROADCAST.

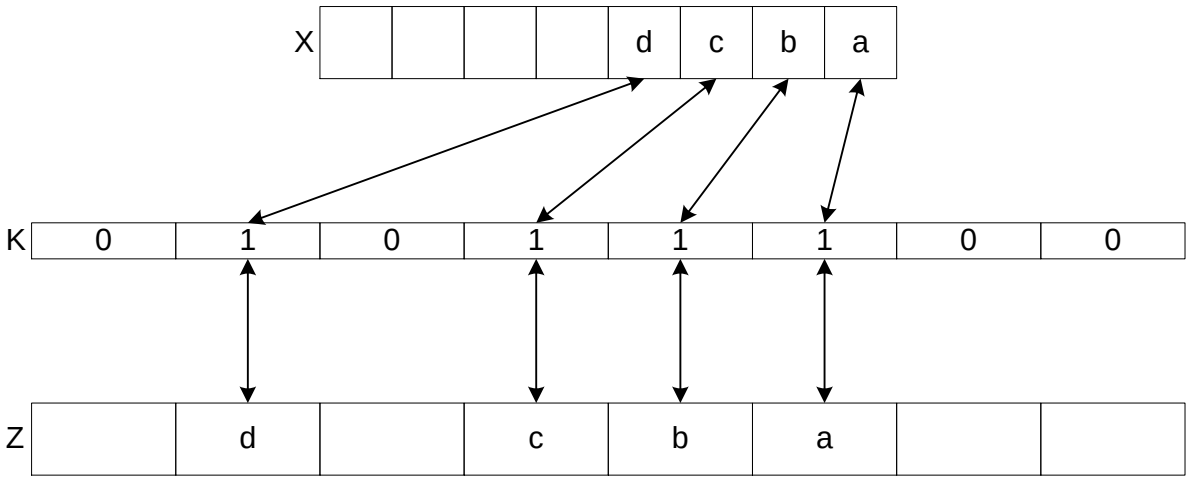

Рис. 2.15. Схема работы операций COMPRESS/EXPAND.

Примеры операций: VMOVDDUP, VMOVSHDUP, VMOVSLDUP, VBROADCASTSD, VBROADCASTSS, VBROADCASTF32X4, VBROADCASTF64X4, VPBROADCASTD, VPBROADCASTQ, VBROADCASTI32X4, VBROADCASTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D, VBROADCASTF32X2, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTI32X2, VBROADCASTI64X2, VBROADCASTI32X8, VPBROADCASTB, VPBROADCASTW , VCOMPRESSPD, VCOMPRESSPS, VPCOMPRESSD, VPCOMPRESSQ, VEXPANDPD, VEXPANDPS, VPEXPANDD, VPEXPANDQ, VEXTRACTF32X4, VEXTRACTF64X4, VEXTRACTI32X4, VEXTRACTI64X4, VEXTRACTF64X2, VEXTRACTF32X8, VEXTRACTI64X2, VEXTRACTI32X8, VINSERTF32X4, VINSERTF64X4, VINSERTI32X4, VINSERTI64X4, VINSERTF64X2, VINSERTF32X8, VINSERTI64X2, VINSERTI32X8, VGATHERDPS, VGATHERDPD, VGATHERQPD, VGATHERDD, VGATHERDQ, VGATHERQQ, VSCATTERDD, VSCATTERDQ, VSCATTERQQ, VSCATTERDPS, VSCATTERDPD, VSCATTERQPD.

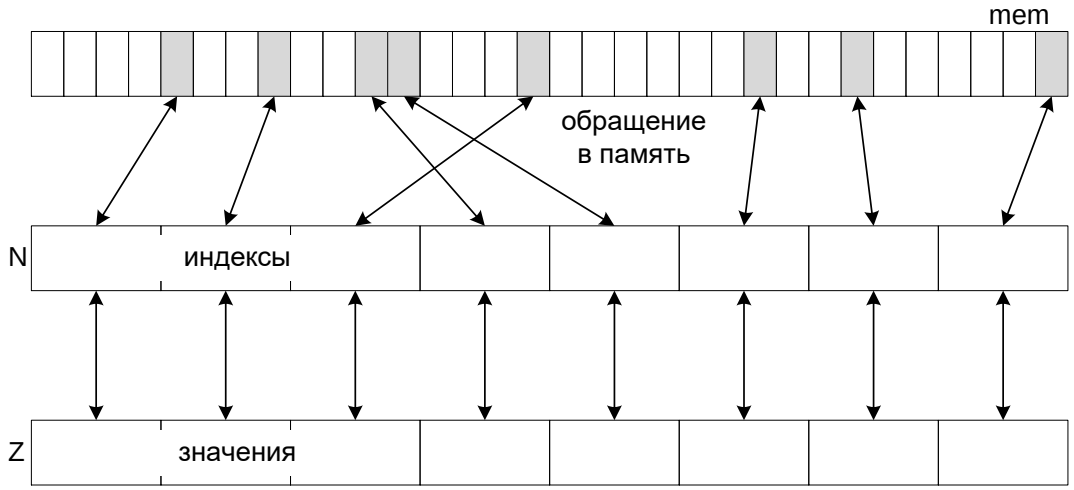

Рис. 2.16. Схема работы операций GATHER/SCATTER.

Операции BROADCAST выполняют дублирование одного элемента данных в вектор с использованием маски. В качестве элемента данных может выступать значение из памяти, часть регистра или маски (рис. 2.14). Мы не рассматриваем отдельно операции MOV*DUP, так как по сути это объединение вместе нескольких мелкогранулярных операций BROADCAST.

Операции COMPRESS и обратные им операции EXPAND предназначены для преобразования между плотными данными и разреженными (рис. 2.15). Таким образом обеспечивается выбор нужных данных для обработки и сохранение результатов обратно. Мы не рассматриваем отдельно операции EXTRACT, которые извлекают из регистра ZMM его часть меньшего размера и обратные им операции INSERT, выполняющие вставку данных в часть регистра ZMM.

Операции GATHER осуществляют выборку из памяти элементов данных с определенными индексами, которые подаются на вход команде в отдельном векторе. Операции SCATTER выполняют обратное действие – сохраняют значения в память с использованием вектора индексов (рис. 2.16). Маски используются для обоих классов команд для определения элементов вектора, участвующих в операции.

2.3.10 Операции предварительной подкачки данных

Примеры операций: VGATHERPF0DPS, VGATHERPF0QPS, VGATHERPF0DPD, VGATHERPF0QPD, VGATHERPF1DPS, VGATHERPF1QPS, VGATHERPF1DPD, VGATHERPF1QPD, VSCATTERPF0DPS, VSCATTERPF0QPS, VSCATTERPF0DPD, VSCATTERPF0QPD, VSCATTERPF1DPS, VSCATTERPF1QPS, VSCATTERPF1DPD, VSCATTERPF1QPD.

Операции предварительной подкачки данных используются для того, чтобы увеличить вероятность того, что к моменту исполнения команды данные уже будут в кэше. Грамотное использование операций предподкачки способно существенно снизить потери от промахов в кэш.

2.3.11 Другие операции

Примеры операций: VPTERNLOGD, VPTERNLOGQ, VFPCLASSPD, VFPCLASSPS, VPMADDUBSW, VPMADDWD, VPSADBW, VDBPSADBW, VPCONFLICTD,

VPCONFLICTQ.

Инструкции TERNLOG (ternary logic) предназначены для выполнения логических функций от трех аргументов над 512-битными векторами. Конкретная функция кодируется в отдельном аргументе (всего доступны 256 различных логических функций).

Инструкции FPCLASS (floating point class) производят проверку каждого из упакованных вещественных значений на принадлежность определенным классам (денормализованные значения, бесконечности, нули, значения not-a-number) и генерируют результирующую маску.

Операции MADD (multiply add) попарно перемножают упакованные данные, хранящиеся в двух операндах, после чего производят горизонтальное сложение двух соседних результатов.

Инструкции SAD (sum of absolute differences) предназначены для суммирования абсолютных разностей между элементами векторов двух аргументов.

Инструкции CONFLICT для каждого элемента данных первого аргумента ищут его дубликат во втором аргументе и в случае совпадения выставляют соответствующий бит в результирующей маске.

Глава 3. Использование инструкций AVX-512 в программном коде

Векторизация циклов в вычислительных задачах играет важную роль, так как способна в разы повысить производительность результирующего исполняемого кода. Во многих случаях с применением векторизации справляется оптимизирующий компилятор. Особенно это относится к классическим шаблонным вычислениям (например, перемножение матриц или суммирование элементов массива), если отсутствуют помехи в виде зависимостей или невыровненных данных. В этом случае приходится применять ручное преобразование кода, которое может заключаться как в переписывании отдельных фрагментов кода в другой форме на языке высокого уровня, так и применять низкоуровневое программирование с помощью команд ассемблера. Компромиссным выходом является применение интринсиков – специальных высокоуровневых оберток над операциями AVX-512, с помощью которых можно подсказать компилятору применение конкретных операций.

В данном разделе описаны некоторые модельные примеры фрагментов исходного кода, к которым может быть применена векторизация с использованием набора AVX-512. В некоторых случаях для этого требуется предварительно выполнить эквивалентные преобразования рассматриваемого кода.

3.1 Векторизация плоского цикла

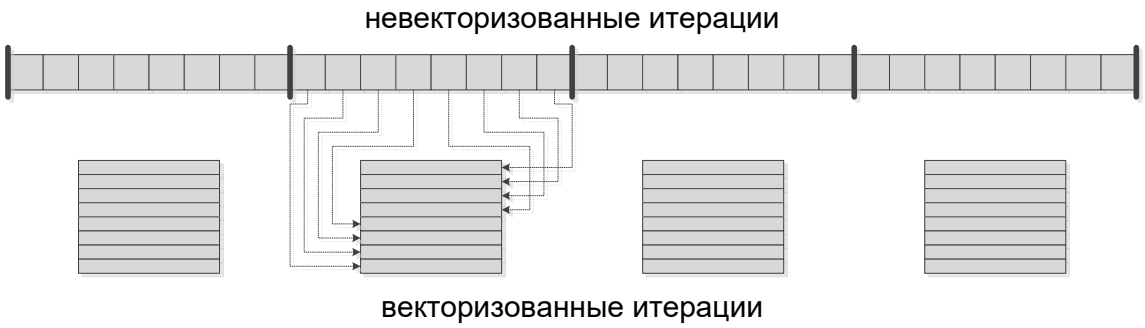

Рис. 3.1. Векторизация плоского цикла.

Самый простой вид циклов для векторизации – плоские циклы. Это циклы, в которых отсутствуют побочные эффекты, зависимости между итерациями, то есть все итерации могут выполняться параллельно. Также в таких циклах обращение к элементам всех массивов на одной итерации цикла выполняется по одному индексу (в идеале на i-ой итерации цикла возможны только обращения вида a[i]) (рис. 3.1). Кроме того на данном этапе будем рассматривать такие циклы, которые работают с выровненными на 64 байта массивами. Такие циклы идеально подходят для векторизации и компилятор применяет ее.

Рассмотрим простейший пример плоского цикла, в котором тело представляет собой чтение элементом трех массивов, выполнение арифметических операций и запись значения в другой массив:

Листинг 1. Пример простейшего плоского цикла.

```
for (int i = 0; i < N; i++)
{
    d[i] = a[i] * b[i] + c[i];
}
```

Такой цикл ожидаемо векторизуется и упаковывается в одну комбинированную FMA операцию:

Листинг 2. Ассемблерный код для цикла из листинга 1.

```
vmovups a(,%rax,8), %zmm1
vmovups b(,%rax,8), %zmm0
vfmadd213pd c(,%rax,8), %zmm0, %zmm1
vmovupd %zmm1, d(,%rax,8)
```

Приведем печать трассировки оптимизации, выдаваемой по опции -qopt-report, в которой видно применение векторизации цикла.

Листинг 3. Диагностика компилятора icc при векторизации цикла из листинга 1.

```
LOOP BEGIN at test-02-simple.cc(10,5)
   remark #15388: vectorization support:
                  reference d[i] has aligned access
   remark #15388: vectorization support:
                  reference a[i] has aligned access
   remark #15388: vectorization support:
                  reference b[i] has aligned access
   remark #15388: vectorization support:
                  reference c[i] has aligned access
   remark #15305: vectorization support: vector length 8
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 3
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 9
   remark #15477: vector cost: 0.870
   remark #15478: estimated potential speedup: 10.280
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=12500
LOOP END
```

В данной трассировке видна информация о выровненности рассматриваемых массивов. Решение компилятора о применении векторизации цикла, а также теоретическая оценка потенциального ускорения цикла после применения векторизации. Кроме того, так как в примере известно точное количество итераций цикла, то выводится информация о количестве итераций векторизованного варианта.

3.2 Векторизация пролога/эпилога плоского цикла

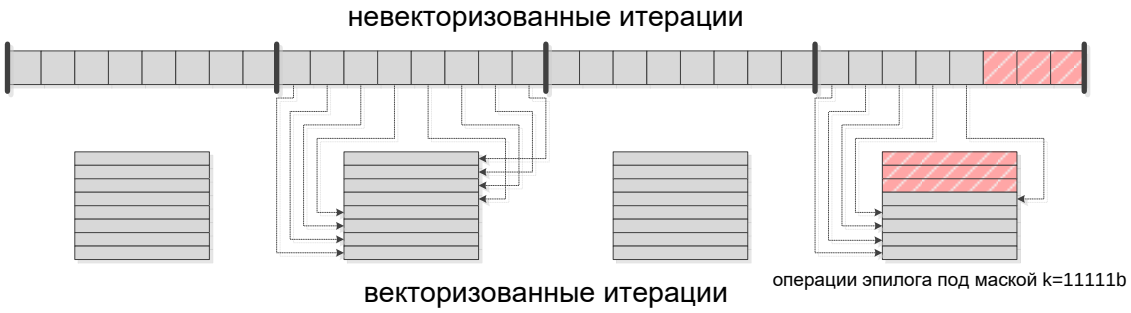

Рис. 3.2. Векторизация эпилога векторизованного цикла.

Бывают ситуации, когда количество итераций цикла не позволяет векторизовать его полностью (количество итераций не кратно степени параллельности). В этом случае остается небольшое количество итераций, которые необходимо выполнить отдельно от остального цикла. Данный довесок называется прологом, если он выполняется перед основным циклом, или эпилогом, его выполняется после основного цикла. В случае, когда нет информации о выравненности одного из массивов, пролог и эпилог создаются компилятором автоматически для обеспечения нужного выравнивания. И хотя основное значение понятий пролога и эпилога связано с конвейеризацией циклов, в случае векторизации употребление данных терминов также приемлемо.

С помощью команд из набора AVX-512 выполнение пролога и эпилога осуществляется с помощью масок применения операций. То есть выполнение пролога и эпилога отличается от выполнения тела цикла только тем, что в маске применения операций несколько первых или последних битов выставлены в 0 (рис. 3.2). Для простоты далее не будем учитывать данные эффекты, считая, что количество итераций циклов и выравнивание массивов с данными позволяют обойтись без прологов и эпилогов циклов.

3.3 Векторизация плоского цикла с условными операциями

Часто в цикле присутствуют операции ветвления. Простейший случай таких операций это наличие оператора if в языке программирования C. При отсутствии других помех векторизации в виде зависимостей между итерациями и невыровненными данными компилятор легко справляется с векторизацией таких циклов с помощью использования маскированных операций. Применение операций с масками для реализации циклов с разветвленным телом активно применяется во всем мире и описан в большом количестве работ по оптимизации вычислений. Для иллюстрации применения масок в теле цикла с разветвленным управлением рассмотрим пример с вычислением результата в зависимости от конкретного условия. Следует различать два вида условий. Первый вид условия – условие по данным, которые загружаются в теле цикла, такой тип условия рассмотрен в примере ниже.

Листинг 4. Пример плоского цикла с условными операциями.

```
for (int i = 0; i < N; i++)
{
    if (a[i] > 0.0)
    {
        c[i] = a[i] + b[i];
    }
    else
    {
        c[i] = a[i] * b[i];
    }
}
```

Тело данного цикла трансформируется в следующий исполняемый код, к котором различные элементы результирующего вектора записываются под разными масками, определяемыми условием a[i] $>$ 0.0.

Листинг 5. Ассемблерный код для цикла из листинга 4.

```
vmovups   a(,%rax,8), %zmm1
vmulpd    b(,%rax,8), %zmm1, %zmm2
vcmppd    $14, %zmm0, %zmm1, %k1
vaddpd    b(,%rax,8), %zmm1, %zmm2{%k1}
vmovupd   %zmm2, c(,%rax,8)
```

Второй случай условия в теле цикла – это условие по индуктивной переменной. Для формирования данного условия требуется больше накладных расходов, так как из скалярных выражений над индуктивной переменной требуется сформировать вектор значений, который далее можно использовать для упакованной операции получения маски.

3.4 Векторизация цикла с вложенным циклом while

Данный случай имеет важное практическое значение в прикладных задачах. Вложенные циклы с неизвестным количеством итераций внутри вычислительного ядра приложения встречаются, например, при построении множества Мандельброта или в реализации решения задачи Римана о распаде произвольного разрыва. Типичным примером в таких вычислениях является цикл с простыми границами типа "int i = 0; i $<$ N; i++"с известным числом итераций и вложенный в него цикл с неизвестным небольшим числом итераций. Данный внутренний цикл не дает применить векторизацию, что негативно влияет на производительность. Использование маскированных операций позволяет векторизовать внешний цикл несмотря на наличие неудобного внутреннего. Рассмотрим ниже тестовый пример.

Листинг 6. Пример плоского цикла с вложенным циклом while.

```
for (int i = 0; i < N; i++)
{
    while (x[i] > 1.0)
    {
```

```
5           x[i] /= 2.0;
6       }
7   }
```

Для векторизации данного цикла применяется следующий прием. Сначала соседние элементы массива x, загружаются и применяется упакованная операция сравнения элементов с единицей. По результатам сравнения вырабатывается маска, по которой должна осуществляться операция деления. Если получившаяся маска пустая, то обработка всех элементов закончена. Если хотя бы один бит маски выставлен в единицу, то выполняется операция деления, а дальше выполняется следующая итерация внутреннего цикла, на которой выполняется упакованная операция сравнения с учетом маски, полученной на предыдущей итерации. Такая параллельная обработка итераций внешнего цикла без изменении логики внутреннего рассмотрена в приведенном ниже коде.

Листинг 7. Пример псевдокода для векторизации цикла из листинга 6.

```
1   for (int i = 0; i < N; i += 8)
2   {
3       __mmask8 msk = 0xFF;
4       __mmask8 mski;
5       __m512d z = _mm512_mask_loadu_pd(_mm512_setzero_pd(),
6                                        0xFF, &x[i]);
7
8       do
9       {
10          mski = _mm512_mask_cmplt_pd_mask(msk, c1, z);
11          msk &= mski;
12          z = _mm512_mask_div_pd(z, msk, z, c2);
13      }
14      while (msk != 0x0);
15
16      _mm512_mask_storeu_pd(&x[i], 0xFF, z);
17  }
```

Количество итераций внутреннего цикла для конкретного i равно максимальному количеству итераций исходного цикла для индуктивной переменной от i до i + 7. Таким образом, в результате применения ручной оптимизации выполнение семи итераций внешнего цикла осуществляется бесплатно. Оптимизирующий компилятор не в состоянии самостоятельно применить данное преобразование, поэтому реализовывать его нужно с помощью интринсиков вручную. Применение данного подхода к построению множества Мандельброта можно найти в [todo].

3.5 Векторизация цикла с маловероятной невекторизуемой веткой

Рассмотрим случай, когда тело векторизуемого цикла представляет собой в целом плоские вычисления, однако в нем существует маловероятная ветка, содержащая нелинейные и невекторизуемые вычисления. Это могут быть вызовы функций или просто вычисления со сложной логикой. В общем случае запишем такой цикл в следующем виде:

Листинг 8. Пример плоского цикла с маловероятной веткой.

```
for (int i = 0; i < N; i++)
{
    <flat calculations>

    if (cond)
    {
        continue; // prob. ~100%
    }
    else
    {
        <calculations with low probability>
    }
}
```

Оптимизирующий компилятор не в состоянии векторизовать такой цикл, так как теоретически маловероятные вычисления могут влиять на плоский участок тела цикла. Однако, если программист уверен в отсутствии побочных эффектов маловероятной ветки, то он может вручную переписать данный цикл, расщепив его на два цикла. В первом из получившихся циклов просто вычисляется условие маловероятной ветки и записывается во временный массив. Во втором цикле обрабатывается весь массив условий и выполняются маловероятные ветки.

Можно уменьшить размер временного массива, если сохранять не все результаты маловероятных условий, а лишь маски соответствующих упакованных операций. В этом случае размер массива оказывается меньше в несколько раз, однако возрастают накладные расходы на его обработку во втором цикле.

Ниже приведен фрагмент кода с использованием временного массива, который содержит маловероятные условия.

Листинг 9. Преобразование цикла из листинга 8 для векторизации.

```
for (int i = 0; i < N; i++)
{
    <flat calculations>
    tmp[i] = cond;
}

for (int i = 0; i < N; i++)
{
    if (!tmp[i])
    {
        <calculations with low probability>
    }
}
```

В оптимизированном варианте основная часть вычислений приходится на первый цикл, при этом он не содержит сложного управления, а значит подходит для векторизации. Второй цикл, конечно, исполняется неэффективно, но его время исполнения настолько

мало, что им можно пренебречь. В результате время исполнения таких шаблонов можно сократить в разы.

3.6 Векторизация свертки над массивом

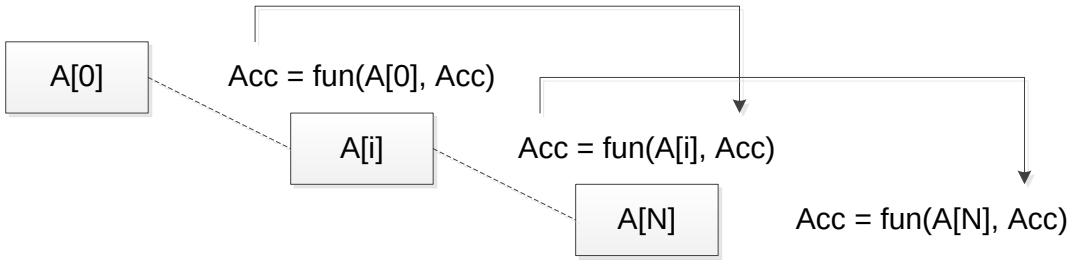

Рис. 3.3. Механизм работы свертки над массивом.

В программировании сверткой называется преобразование структуры данных к единственному значению при помощи применения заданной функции. В функциональном программировании свертка является функцией высшего порядка. В отношении массива можно рассмотреть механизм свертки следующим образом: на входе в свертку подается массив, функция свертки и начальное значение аккумулятора. На первой итерации функция применяется к первому элементу массива и аккумулятору, результат применения записывается обратно в аккумулятор. Далее вычисления производятся по аналогии, значение аккумулятора после применения всех итераций является результатом свертки (рис. 3.3). Свертки также можно применять к нескольким массивам сразу, однако этот случай легко сводится к свертке над одним массивом.

Простейшим примером свертки, выполняемой над массивом, является вычисление суммы его элементов (заметим, что операция сложения вещественных чисел некоммутативна в общем случае). Однако, сумма элементов массива является обычным шаблоном для оптимизирующего компилятора и не представляет сложности для векторизации. Поэтому для усложнения примера добавим искусственное действие над аккумулятором в конце каждой итерации.

Листинг 10. Пример цикла с использованием свертки над массивом.

```
1 double sum = 0.0;
2
3 for (int i = 0; i < N; i++)
4 {
5     sum += A[i];
6     sum /= 2.0;
7 }
```

Такой цикл векторизовать компилятору уже не под силу из-за существующей зависимости по данным. Поэтому для его оптимизации нужно выполнить ручное преобразование кода. Единственным преобразованием, которое способно помочь разрешить зависимости

между соседними итерациями в данном случае, является раскрутка цикла и дальнейший сбор подвыражений. Конечно можно попробовать поручить компилятору выполнить преобразования самостоятельно, но тогда придется разрешать перестановку вещественных операций. Мы выполним преобразования вручную. Тестовому примеру соответствует следующий код, который выполняет параллельно 8 итераций исходного цикла:

Листинг 11. Пример псевдокода преобразованного цикла из листинга 10 для векторизации.

```
double sum = 0.0;

for (int i = 0; i < N; i += 8)
{
    __m512d v = _mm512_set_pd(1.0 / 256.0, 1.0 / 128.0,
                              1.0 / 64.0, 1.0 / 32.0,
                              1.0 / 16.0, 1.0 / 8.0,
                              1.0 / 4.0, 1.0 / 2.0);
    __m512d z = _mm512_mask_loadu_pd(_mm512_setzero_pd(),
                                     0xFF, &a[i]);

    z = _mm512_mask_mul_pd(z, 0xFF, z, v);

    sum /= 256.0;
    sum += _mm512_reduce_add_pd(z);
}
```

Единственным неоптимальным местом в данном коде является интринсик $\_mm512\_reduce\_add\_pd$, который имитирует горизонтальное сложение элементов вектора. В наборе команд AVX-512 нет полноценного функционала для горизонтального сложения всех элементов вектора, поэтому для конкретных примеров целесообразно выбирать индивидуальные подходы для уменьшения накладных расходов на данную операцию.

Глава 4. Компиляция программного кода с инструкциями AVX-512

4.1 Компиляция с помощью GCC

В данном разделе приведено описание основных флагов компилятора gcc, которые используются при компиляции приложений, использующих AVX-512 [27].

`-march=knl` - указание компилятору генерировать код с использованием инструкций из наборов AVX-512 F, AVX-512 PF, AVX-512 ER, AVX-512 CD. Опция предназначена для компиляции программ для микропроцессора Intel Xeon Phi Knights Landing.

`-march=skylake-avx512` - указание компилятору генерировать код с использованием инструкций из наборов AVX-512 F, AVX-512 VL, AVX-512 BW, AVX-512 DQ, AVX-512 CD. Опция предназначена для компиляции программ для микропроцессора Intel Xeon Skylake.

`-mavx512f` - разрешение компилятору генерировать код с использованием инструкций из набоора AVX-512 F.

`-mavx512pf` - разрешение компилятору генерировать код с использованием инструкций из набоора AVX-512 PF.

`-mavx512er` - разрешение компилятору генерировать код с использованием инструкций из набоора AVX-512 ER.

`-mavx512cd` - разрешение компилятору генерировать код с использованием инструкций из набоора AVX-512 CD.

`-mavx512vl` - разрешение компилятору генерировать код с использованием инструкций из набоора AVX-512 VL.

`-mavx512bw` - разрешение компилятору генерировать код с использованием инструкций из набоора AVX-512 BW.

`-mavx512dq` - разрешение компилятору генерировать код с использованием инструкций из набоора AVX-512 DQ.

4.2 Компиляция с помощью ICC

В данном разделе приведено описание основных флагов компиялятора icc, котрые используются при компиляции приложений, использующих AVX-512 [28].

`-axCOMMON_AVX512` – указание компилятору генерировать код с использованием инструкций из наборов AVX-512 F и AVX-512 CD и выполнять данный код в случае поддержки этих инструкций целевой архитектурой.

`-axCORE_AVX512` – указание компилятору генерировать код с использованием инструкций из наборов AVX-512 F, AVX-512 CD, AVX-512 DQ, AVX-512 BW, AVX-512 VL и выполнять данный код в случае поддержки этих инструкций целевой архитектурой.

`-xMIC_AVX512` – указание компилятору генерировать код использованием инструкций из наборов AVX-512 F, AVX-512 CD, AVX-512 ER, AVX-512 PF. Данная опция предназначена для компиляции кода под микропроцессоры Intel Xeon Phi Knights Landing.

`-qopt-report` – указание компилятору выдавать диагностическую информацию о проводимых оптимизациях. В качестве параметра подается уровень диагностики. Максимальный уровень равен 5.

4.3 Использование функций-интринсиков

Для удобства применения инструкций из набора AVX-512 предусмотрена библиотека специальных функций интринсиков, полный список и описание данных функций доступно

на сайте [29].

Для использования функций-интринсиков в программе нужно подключить заголовочный файл `<immintrin.h>`.

Функции-интринсики покрывают не все инструкции AVX-512, однако избавляют от необходимости вручную писать ассемблерный код и позволяют использовать встроенные типы данных для 512-битных векторов (`__m512`, `__m512i`, `__m512d`). Некоторые интринсики соответствуют не отдельной команде, а целой последовательности, как например для операции сложения всех элементов вектора. Из множества интринсиков можно выделить следующие группы функций, схожие по структуре. Функции swizzle, shuffle, permute и permutevar осуществляют перестановку элементов вектора и раскрываются в последовательность операций, в которой присутвует shuf и пересылка по маске. Для большего числа операций AVX-512 реализованы соответствующие инстринсики, раскрывающиеся в одну конкретную операцию. Среди них арифметические операции, побитовые операции, операции чтения из памяти и записи в память, операции конвертации, слияние двух векторов, нахождение обратных значений, получение минимума и максимума из двух значений, операции сравнения, операции с масками, комбинированные операции и другие. Некоторые инстринсики, особенно предназначенные для выполнения упакованных трансцендентных операций, раскрываются просто в вызов библиотечной функции (например `_mm512_log_ps`, `_mm512_hypot_ps`, тригонометрические функции).

Глава 5. Примеры практического применения векторизации с использованием инструкций AVX-512

В данном разделе собраны примеры векторизации участков программного кода, использованные в ходе проведения научных исследований.

5.1 Применение векторизации для матричных операций

5.1.1 Теоретическая часть

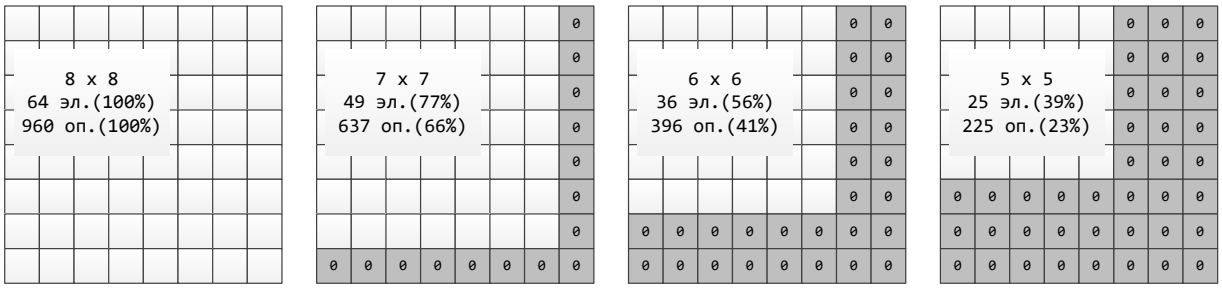

Рис. 5.1. Иллюстрация матриц размера 8x8, 7x7, 6x6, 5x5, представленных как подматрицы матрицы 8x8.

При проведении вычислений с помощью метода RANS/ILES высокого разрешения расчетные параметры компонуются внутри матриц размером 5x5, при этом данные матрицы являются подматрицами матриц размера 8x8 (это делается из соображений выровненности расположения в памяти). Наиболее частой операцией при проведении вычислений является перемножение таких матриц (данная функция занимает до 40% расчетного времени). Поэтому для повышения эффективности проведения расчетов с использованием метода RANS/ILES требуется эффективная реализация функций для работы с обозначенными объектами.

В данном разделе рассматриваются матрицы размера 8x8 (элементами которых имеют тип float) и приводится эффективная реализация перемножения таких матриц с помощью инструкций AVX-512. Реализация перемножения для матриц размера 7x7, 6x6, 5x5 выполняется аналогично (рис. 5.1).

Для создания векторизованного кода перемножения двух матриц запишем в явном виде значения элементов i-й строки результирующей матрицы:

$$\begin{cases} r_{i0} = a_{i0}b_{00} + a_{i1}b_{10} + \ldots + a_{i7}b_{70} \\ \ldots, \\ r_{i7} = a_{i0}b_{07} + a_{i1}b_{17} + \ldots + a_{i7}b_{77} \end{cases} \tag{5.1}$$

или в векторной форме

$$\overline{r_i} = a_{i0}\overline{b_0} + a_{i1}\overline{b_1} + \ldots + a_{i7}\overline{b_7}. \tag{5.2}$$

Аналогичные выражения можно записать для строки с номером $i + 1$:

$$\begin{cases} r_{i+1,0} = a_{i+1,0}b_{00} + a_{i+1,1}b_{10} + \ldots + a_{i+1,7}b_{70}, \\ \ldots \\ r_{i+1,7} = a_{i+1,0}b_{07} + a_{i+1,1}b_{17} + \ldots + a_{i+1,7}b_{77} \end{cases} \tag{5.3}$$

или в векторной форме

$$\overline{r_{i+1}} = a_{i+1,0}\overline{b_0} + a_{i+1,1}\overline{b_1} + \ldots + a_{i+1,7}\overline{b_7}. \tag{5.4}$$

Учитывая то, что zmm регистры содержат по 16 элементов типа float, то целесообразно объединить приведенные выше формулы в одну, записанную в векторном виде следующим образом:

$$\left(\frac{\overline{r_i}}{r_{i+1}}\right) = \left(\left(\frac{\overline{a_{i0}}}{a_{i+1,0}}\right) \circ \left(\frac{\overline{b_0}}{\overline{b_0}}\right)\right) + \left(\left(\frac{\overline{a_{i1}}}{a_{i+1,1}}\right) \circ \left(\frac{\overline{b_1}}{\overline{b_1}}\right)\right) + \ldots + \left(\left(\frac{\overline{a_{i7}}}{a_{i+1,7}}\right) \circ \left(\frac{\overline{b_7}}{\overline{b_7}}\right)\right), \tag{5.5}$$

где $\left(\frac{\overline{r_i}}{r_{i+1}}\right)$ обозначает комбинированный вектор состоящий из векторов $\overline{r_i}$ и $\overline{r_{i+1}}$, $\left(\frac{\overline{b_j}}{\overline{b_j}}\right)$ обозначает комбинированный вектор, состоящий из двух копий вектора $\overline{b_j}$, а выражение $\left(\frac{\overline{a_{ij}}}{a_{i+1,j}}\right)$ обозначает вектор, первые 8 элементов которого равны a_{ij}, а остальные 8 элементов – $a_{i+1,j}$ (“$\circ$” – произведение Адамара, или поэлементное произведение векторов). Заметим, что получающийся по данной формуле комбинированный вектор $\left(\frac{\overline{r_i}}{r_{i+1}}\right)$ расположен в памяти последовательно, и записывать его в память можно с помощью интринсика $\_mm512\_store\_ps$. При этом предполагаем, что значение i четно, то есть вектор $\left(\frac{\overline{r_i}}{r_{i+1}}\right)$ выровнен в памяти должным образом. Другие комбинированные векторы в данном выражении получаются с помощью инструкции perm (интринсик $\_mm512\_permutexvar\_ps$), примененной к соответствующим загруженным соседним строкам матриц a и b. Таким образом, при реализации вышеприведенной формулы не требуется использование медленных инструкций gather/scatter, так как столбцы матриц не читаются и не записываются (работа ведется только со строками). После того, как требуемые векторы сформированы, нужно выполнить их попарное поэлементное перемножение, после чего сложить в один вектор (8 операций поэлементного умножения, 7 операций сложения). Эти действия можно выполнить, используя комбинированные операции fmadd (рис. 5.2).

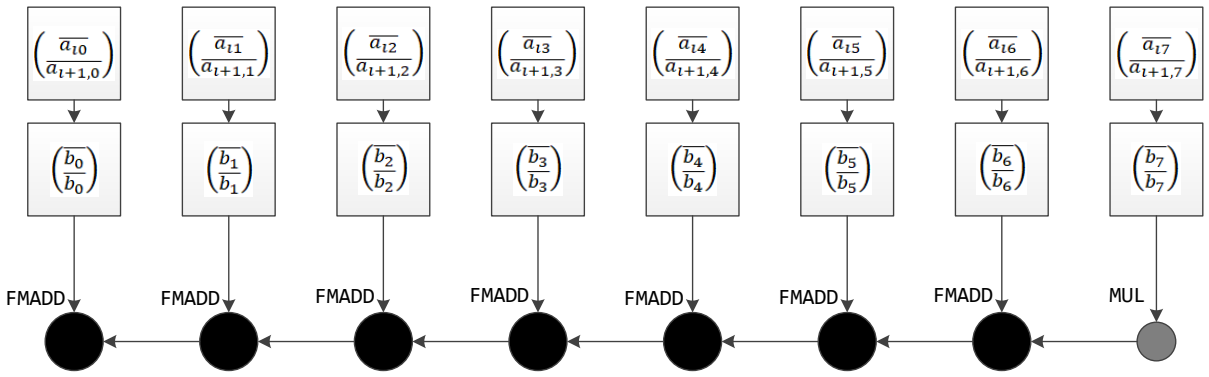

Рис. 5.2. Схема вычисления двух соседних строк результирующей матрицы путем последовательного сложения попарно перемноженных векторов.

5.1.2 Реализация перемножения матриц

Приведем функцию перемножения двух матриц размера 8x8, механизм которой был рассмотрен в предыдущем разделе. Для реализации выполним полную загрузку обеих

матриц a и b. На это потребуется 8 операций обращения в память, так как за одну операцию загружаются две соседние строки матрицы. Далее требуется сформировать 8 векторов для элементов матрицы b, для чего потребуется еще 8 операций perm (для каждой пары загруженных строк матрицы b нужно выполнить дублирование первой строки и дублирование второй строки). Формирование векторов индексов для операций perm не требует вычислительного времени, так как индексы являются статическими и вычисляются на этапе компиляции.

После подготовки всех необходимых данных выполняется вычисление значений результирующей матрицы. Приведенный ниже блок операций (макрос BLOCK) осуществляет вычисление двух соседних строк результирующей матрицы. Реализация блока состоит из 8 операций perm, 1 операции mul и 7 операций fmadd, кроме того выполняется одна операция записи в память. Всего выполняется четыре таких блока, что в сумме и с учетом операций подготовки данных приводит к следующему итогу: 8 простых операций чтения из памяти, 40 операций perm, 4 операции mul, 28 операций fmadd, 4 простые операции записи в память. Итоговый код (без отображения некоторых повторяющихся участков) приведен на листинге ниже

Листинг 12. Векторная реализация перемножения матриц 8x8.

```
void mul_8x8_opt(float * __restrict a, float * __b, float *
    __restrict r)
{
    ....................................
    // Indices for copiyng the first and the second halfs of zmm
        register.
    __m512i ind_df = _mm512_set_epi32( 7,  6,  5,  4,  3,  2, 1, 0,
                                       7,  6,  5,  4,  3,  2, 1, 0);
    __m512i ind_ds = _mm512_set_epi32(15, 14, 13, 12, 11, 10, 9, 8,
                                      15, 14, 13, 12, 11, 10, 9, 8);

    // Loading all lines of matrix b (b0-b7) with duplication.
    __m512 b0 = LD(&b[0]);
    __m512 b1 = _mm512_permutexvar_ps(ind_ds, b0);
    b0 = _mm512_permutexvar_ps(ind_df, b0);
    ....................................
    __m512 b6 = LD(&b[6 * V8]);
    __m512 b7 = _mm512_permutexvar_ps(ind_ds, b6);
    b6 = _mm512_permutexvar_ps(ind_df, b6);

    // Loading all lines of matrix a (two lines in one zmm register).
    __m512 a0 = LD(&a[0]);
    ....................................
    __m512 a6 = LD(&a[6 * V8]);

    // Indices for matrix a element choosing.
    __m512i ind_0 = _mm512_set_epi32( 8,  8,  8,  8,  8,  8,  8,  8,
                                      0,  0,  0,  0,  0,  0,  0,  0);
    ....................................
    __m512i ind_7 = _mm512_set_epi32(15, 15, 15, 15, 15, 15, 15, 15,
                                      7,  7,  7,  7,  7,  7,  7,  7);

// Definition of main calculations block.
#define BLOCK(N, A)                             \
```

```
33      ST(&r[N * V8],                                  \
34        FMADD(PERMXV(ind_0, A), b0,                   \
35          FMADD(PERMXV(ind_1, A), b1,                 \
36            FMADD(PERMXV(ind_2, A), b2,               \
37              FMADD(PERMXV(ind_3, A), b3,             \
38                FMADD(PERMXV(ind_4, A), b4,           \
39                  FMADD(PERMXV(ind_5, A), b5,         \
40                    FMADD(PERMXV(ind_6, A), b6,       \
41                      MUL(PERMXV(ind_7, A), b7)))))))));
42
43      // Calculate and save the result.
44      BLOCK(0, a0);
45      BLOCK(2, a2);
46      BLOCK(4, a4);
47      BLOCK(6, a6);
48
49  #undef BLOCK
50
51  }
```

Заметим, что 28 векторных операций fmadd и 4 векторные операции mul соответствуют $(28 \cdot 2 + 4) \cdot 16 = 960$ скалярным операциям, что в точности совпадает с количеством скалярных операций, требуемых для выполнения перемножения двух матриц размера 8x8. Таким образом, в предложенной реализации нет лишних арифметических операций, и результат каждой выполненной операции влияет на конечный результат. При реализации перемножения матриц размера 7x7, 6x6, 5x5 удаляются заведомо лишние векторные операции (например, умножение на вектор, все элементы которого равны нулю), однако все равно остаются элементы векторов, обработка которых избыточна, что приводит к снижению эффективности векторизации в этих случаях.

С помощью использования предложенного метода удалось добиться ускорения перемножения матриц размера 8x8 почти в 6 раз, при уменьшении размера матриц эффективность снижается, для матриц размера 5x5 ускорение достигает 2.5 раз [30].

5.2 Векторизация анализа пересечений геометрических примитивов

При численном решении задач газовой динамики часто приходится сталкиваться с телами, обладающими сложной геометрией. Для таких тел построение согласованной расчетной сетки может быть крайне трудозатратной задачей. Альтернативой в данном случае является использование метода погруженной границы. Данный метод позволяет использовать для расчетов несогласованную сетку и даже простую декартову сетку, что сильно упрощает выполнение расчетов. В рамках реализации метода погруженной границы требуется решать частную задачу определения пересечения треугольника и прямоугольного параллелепипеда в пространстве для большого количества данных геометрических примитивов (рис. 5.3).

5.2.1 Задача о пересечении треугольника и прямоугольного параллелепипеда

В данном разделе рассмотрим математическую постановку и метод решения задачи об определении пересечения треугольника и прямоугольного параллелепипеда в простран-

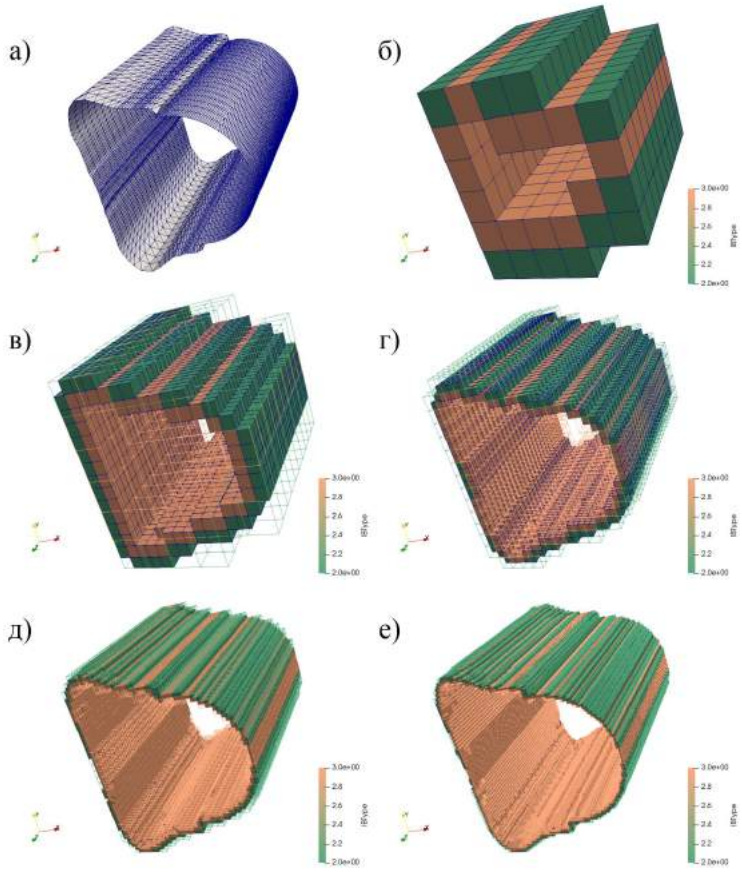

Рис. 5.3. Иллюстрация пересечения поверхностной сетки и объемных сеток с разным количеством ячеек. На рисунках цветами обозначены только граничные и фиктивные ячейки объемной сетки (граничные – зеленые, фиктивные – бежевые). а) Исходная геометрия поверхностной сетки. б) Размер объемной сетки 10x10x10. в) Размер объемной сетки 25x25x25. г) Размер объемной сетки 50x50x50. д) Размер объемной сетки 100x100x100. е) Размер объемной сетки 150x150x150.

стве.

Пусть треугольник задан тремя точками: $A(x_A, y_A, z_A)$, $B(x_B, y_B, z_B)$, $C(x_C, y_C, z_C)$. Тогда координаты любой точки $P(x, y, z)$, находящейся внутри треугольника, можно представить следующим образом:

$$\begin{cases} x = x_A + (x_B - x_A)\alpha + (x_C - x_A)\beta \\ y = y_A + (y_B - y_A)\alpha + (y_C - y_A)\beta \\ z = z_A + (z_B - z_A)\alpha + (z_C - z_A)\beta \end{cases} \tag{5.6}$$

где $\alpha \geq 0$, $\beta \geq 0$, $\alpha + \beta \geq 1$.

Геометрическим местом точек прямоугольного параллелепипеда является множество точек $P(x, y, z)$, координаты которых удовлетворяют следующей системе неравенств:

$$\begin{cases} x_l \leq x \leq x_h \\ y_l \leq y \leq y_h \\ z_l \leq z \leq z_h \end{cases} \tag{5.7}$$

Для установления факта пересечения треугольника и прямоугольного параллелепипеда нужно определить, имеет ли решение приведенная ниже система неравенств относительно α и β:

$$\begin{cases} x_l \le x_A + (x_B - x_A)\alpha + (x_C - x_A)\beta \le x_h \\ y_l \le y_A + (y_B - y_A)\alpha + (y_C - y_A)\beta \le y_h \\ z_l \le z_A + (z_B - z_A)\alpha + (z_C - z_A)\beta \le z_h \\ \alpha \ge 0 \\ \beta \ge 0 \\ \alpha + \beta \le 1 \end{cases} \tag{5.8}$$

Для подобных систем неравенств существует много различных способов решения, описание которых можно найти, например, в [31]. В нашем случае система довольно простая, содержит две переменные, и к ней может быть применен метод свертывания конечных систем линейных неравенств, описанный в [32].

Для решения методом свертывания преобразуем систему неравенств так, чтобы она содержала неравенства только вида $k_\alpha\alpha + k_\beta\beta + k \le 0$. После выполнения всех преобразований система неравенств примет следующий вид:

$$\begin{cases} (x_B - x_A)\alpha + (x_C - x_A)\beta + (x_A - x_h) \le 0 \\ (x_A - x_B)\alpha + (x_A - x_C)\beta + (x_l - x_A) \le 0 \\ (y_B - y_A)\alpha + (y_C - y_A)\beta + (y_A - y_h) \le 0 \\ (y_A - y_B)\alpha + (y_A - y_C)\beta + (y_l - y_A) \le 0 \\ (z_B - z_A)\alpha + (z_C - z_A)\beta + (z_A - z_h) \le 0 \\ (z_A - z_B)\alpha + (z_A - z_C)\beta + (z_l - z_A) \le 0 \\ -1 \cdot \alpha + 0 \cdot \beta \le 0 \\ 0 \cdot \alpha + (-1)\beta \le 0 \\ \alpha + \beta + (-1) \le 0 \end{cases} \tag{5.9}$$

Так как система содержит всего две переменные, то после выполнения одного шага свертывания (или деформации системы) она превратится в систему неравенств относительно одной переменной, проверка разрешимости которой не представляет труда. Будем выполнять деформацию системы с целью исключить из нее переменную α. Для этого составим новую систему, в которую войдут все неравенства исходной системы вида $k_\beta\beta + k \le 0$, а каждая пара неравенств

$$\begin{cases} k_\alpha^1\alpha + k_\beta^1 + k^1 \le 0 \\ k_\alpha^2\alpha + k_\beta^2 + k^2 \le 0 \end{cases} \tag{5.10}$$

где $k_\alpha^1 < 0$, а $k_\alpha^2 > 0$ войдет в деформированную систему в виде

$$(k_\beta^1 k_\alpha^2 - k_\beta^2 k_\alpha^1)\beta + (k^1 k_\alpha^2 - k^2 k_\alpha^1) \le 0 \tag{5.11}$$

Так как исходная система содержит 9 уравнений, по крайней мере одно из которых имеет нулевой коэффициент при переменной α, а из оставшихся восьми половина коэффициентов при переменной α неотрицательны, а половина неположительны, то деформированная система будет содержать не более 17 уравнений.

5.2.2 Векторизация вычислений

Рассмотрим реализацию функции
$tri\_box\_intersect(xa, ya, za, xb, yb, zb, xc, yc, zc, xl, xh, yl, yh, zl, zh) \to int$, анализирующую

наличие пересечения треугольника и прямоугольного параллелепипеда. Функция возвращает 1, если пересечение есть, и 0, если пересечения нет. Логика работы функции следующая. Сначала коэффициенты системы неравенств 5.9 заносятся в двумерный массив коэффициентов b[bec][3], где bec (basic equations count) – количество исходных неравенств системы (в нашем случае 9). Затем выполняется один шаг свертывания системы с одновременным поиском множества решения для переменной β. Перед началом свертывания множество допустимых значений для переменной β принимается в виде отрезка [0, 1] (lo = 0, hi = 1). По мере свертывания системы неравенств 5.9 происходит сокращение данного множеств решений. Если на каком-то этапе свертывания множество решений обращается в пустое (lo > hi), то функция заканчивает работу и возвращает 0. Если после выполнения всех действий свертывания множество решений осталось ненулевым, то это означает наличие пересечения, и функция возвращает 1. Реализация свертывания системы уравнений с помощью метода из [32] представлена на листинге ниже:

Листинг 13. Исходная реализация свертывания системы линейных неравенств для определения пересечения треугольника и прямоугольного параллелепипеда.

```
for (i = 0; i < bec; i++)
{
    bi0 = b[i][0];

    if (bi0 == 0.0)
    {
        if (!upgrade(b[i][1], b[i][2], &lo, &hi))
        {
            return 0;
        }
    }
    else
    {
        for (j = i + 1; j < bec; j++)
        {
            if (bi0 * b[j][0] < 0.0)
            {
                f0 = bi0 * b[j][1] - b[j][0] * b[i][1];
                f1 = bi0 * b[j][2] - b[j][0] * b[i][2];

                if (bi0 < 0.0)
                {
                    f0 = -f0;
                    f1 = -f1;
                }

                if (!upgrade(f0, f1, &lo, &hi))
                {
                    return 0;
                }
            }
        }
    }
}

return 1;
```

Получившийся код можно охарактеризовать как имеющий сложное управление, уровень вложенности конструкций в нем достигает 5 (for-if-for-if-if), к тому же участок содержит 3 выхода из функции. Функция upgrade, которая вызывается из кода, предназначена для обновления текущего множества допустимых значений для переменной β с учетом нового полученного ограничения вида $k_\beta\beta + k \leq 0$, коэффициенты которого передаются в первом и втором параметрах. Текущее множество решений является отрезком с границами, хранящимися в переменных lo и hi, и в зависимости от знака коэффициента k_β одна из этих границ внутри вызова функции upgrade может измениться (граница lo может увеличиться, либо граница hi может уменьшиться). Если после обновления множества решений оно оказывается пустым (нижняя граница становится больше верхней), то функция upgrade возвращает 0, в противном случае она возвращает 1.

После рассмотрения реализации функции $tri\_box\_intersect$ можно перейти к векторизации вычислений. Для анализа пересечения двух сеток необходимо вызывать функцию $tri\_box\_intersect$ многократно с разными наборами входных параметров. Для оптимизации этого процесса реализуем функцию $tri\_box\_intersect\_16$, объединяющую внутри себя обработку 16 вызовов функции $tri\_box\_intersect$ (используется объединение 16 вызовов, так как это совпадает с количеством элементов вещественных данных одинарной точности в одном векторе $\_\_m512$, $VEC\_WIDTH = 16$).

Ниже представлена реализация функции $tri\_box\_intersect\_16$ в первом приближении.

Листинг 14. Исходная реализация функции, объединяющей 16 вызовов функции $tri\_box\_intersect$.

```
void tri_box_intersect_16(float *xa, float *ya, float *za,
                          float *xb, float *yb, float *zb,
                          float *xc, float *yc, float *zc,
                          float *xl, float *xh,
                          float *yl, float *yh,
                          float *zl, float *zh,
                          int *r)
{
    for (int i = 0; i < 16; i++)
    {
        r[i] = tri_box_intersect(xa[i], ya[i], za[i],
                                 xb[i], yb[i], zb[i],
                                 xc[i], yc[i], zc[i],
                                 xl[i], xh[i], yl[i], yh[i], zl[i],
                                     zh[i]);
    }
}
```

Конечно в реальной задаче требуется обрабатывать миллионы вызовов, однако они могут быть разбиты на группы по 16 и обработаны с помощью функции $tri\_box\_intersect\_16$, поэтому остановимся подробнее на векторизации данной функции.

Для векторизации функции $tri\_box\_intersect\_16$ следует выполнить подстановку тела функции $tri\_box\_intersect$ в место ее вызова. После выполнения данного преобразования получаем программный контекст, содержащий сложное управление и в частности гнездо из трех вложенных циклов, не считая условий. Данный контекст не может быть

векторизован компилятором автоматически, поэтому будем проводить его векторизацию в ручном режиме с использованием функций-интринсиков, которые позволяют напрямую использовать инструкции AVX-512 в синтаксисе языка программирования C без использования ассемблера. Заметим, что между итерациями внешнего цикла отсутствуют зависимости, то есть цикл является плоским, и его итерации могут быть выполнены в любом порядке, в том числе и одновременно. Такие циклы поддаются векторизации путем перевода тела в предикатное представление и заменой скалярных инструкций на векторные. Наиболее тонким местом при векторизации тела плоского цикла являются условия, то есть наличие конструкций if-else. Альтернативные ветви таких конструкций должны быть объединены в предикатном коде под противоположными предикатами. Наличие большого количества условных операторов в исходном коде порождает множество инструкций под нулевыми предикатами в результирующем коде, что негативно сказывается на производительности. Для уменьшения количества условных операторов, можно использовать математические тождества, заменяющие условные конструкции на команды, имеющие векторные аналоги в наборе инструкций AVX-512. Для данных целей хорошо подходят такие векторные команды как abs, min, max, blend, avg и другие. Например в рассматриваемом коде вычисление значений f0 и f1 с учетом условия b[i][0] * b[j][0] $<$ 0 может быть заменено на следующее:

Листинг 15. Применение тождеств для использования векторных инструкций.

```
1 f0 = fabs(bi0) * b[j][1]
2      + fabs(b[j][0]) * b[i][1]
3 f1 = fabs(bi0) * b[j][2]
4      + fabs(b[j][0]) * b[i][2]
```

Похожие трудности вызывает внешнее условие if на листинге 13. Данное условие имеет альтернативную ветку, содержащую цикл. Слияние данных двух ветвей снижает производительность результирующего кода, поэтому в данном случае выгодно применить расщепление внешнего цикла по конструкции if-else (по-другому данное преобразование называют loop distribution). При этом образуются два гнезда циклов, каждое из которых может быть векторизовано независимо. Заметим, что описанное преобразование в общем смысле не является эквивалентным, так как обе ветки условия if-else, а значит и тела образовавшихся циклов содержат выходы из функции, выполнение расщепления цикла может изменить условие, провоцирующее выход из функции. По этой причине компилятор не способен выполнить данное преобразование автоматически. Однако с точки зрения результата функции данное преобразование корректно, поэтому мы его и применяем.

Отметим еще один крайне положительный момент в рассматриваемом программном контексте. Условием выхода из вложенных циклов является достижение индуктивной переменной значения bec. Данное значение является константой, а значит не зависит от номера итерации, поэтому в векторный код это условие может быть перенесено без изменений. В случае зависимости условия выхода из цикла от номера итерации само условие должно быть также векторизовано, что может привести к потерям производительности. Однако в рассматриваемом коде проблем векторизации циклов с нерегулярным количеством итераций нет. На листингах ниже приведем получившийся код для функции $tri\_box\_intersect\_16$, а также его схематичную трансформацию в векторный аналог.

Листинг 16. Схема программного кода перед переводом в векторную форму.

```
float b[bec][3][VEC_WIDTH];

<init b>;

for (w = 0; w < VEC_WIDTH; w++) r[w] = 1;

for (w = 0; w < VEC_WIDTH; w++)
{
    lo = 0.0;
    hi = 1.0;

    for (i = 0; i < bec; i++)
    {
        upgrade(b[i][0][w] == 0.0,
                b[i][1][w], b[i][2][w], &lo, &hi);
        if (lo > hi) break;
    }

    for (i = 0; i < bec; i++)
    {
        bi0 = b[i][0][w];
        abi0 = fabs(bi0);

        for (j = i + 1; j < bec; j++)
        {
            bj0 = b[j][0][w];
            abj0 = fabs(bj0);

            upgrade(bi0 * bj0 < 0.0,
                    abi0 * b[j][1][w] + abj0 * b[i][1][w],
                    abi0 * b[j][2][w] + abj0 * b[i][2][w],
                    &lo, &hi);
            if (lo > hi) break;
        }

        if (lo > hi) break;
    }

    if (lo > hi) r[w] = 0;
}
```

Листинг 17. Векторная форма получившегося программного кода из листинга 16.

```
__m512 b[bec][3];

<init b>;

_mm512_store_epi32(r, _mm512_set1_epi32(1));
```

```
__m512 lo = z0;
__m512 hi = z1;

for (i = 0; i < bec; i++)
{
    upgrade(_mm512_cmpeq_ps_mask(b[i][0], z0),
            b[i][1], b[i][2], &lo, &hi);
    if (!_mm512_cmplt_ps_mask(lo, hi)) break;
}

for (i = 0; i < bec; i++)
{
    bi0 = b[i][0];
    abi0 = ABS(bi0);

    for (j = i + 1; j < bec; j++)
    {
        bj0 = b[j][0];
        abj0 = ABS(bj0);

        upgrade(_mm512_cmplt_ps_mask(MUL(bi0, bj0), z0),
                FMADD(abi0, b[j][1], MUL(abj0, b[i][1])),
                FMADD(abi0, b[j][2], MUL(abj0, b[i][2])),
                &lo, &hi);
        if (!_mm512_cmplt_ps_mask(lo, hi)) break;
    }

    if (!_mm512_cmplt_ps_mask(lo, hi)) break;
}

_mm512_mask_store_epi32(r,
                        _mm512_cmplt_ps_mask(hi, lo),
                        _mm512_set1_epi32(0));
```

Из листингов 16 и 17 видно, что правильно составленный предикатный код может быть довольно просто переведен в векторный аналог. Для этого должны соблюдаться некоторые простые требования. Во-первых, необходимо удалить все else ветки в условных операторах. Этого можно добиться, например, путем расщепления оператора if-else на два противоположных условия. После этого любое условие if легко трансформируется в предикат на весь блок кода, находящийся под условием. Во-вторых, вызовы функции не должны находиться под предикатами. Вместо этого в нашем случае скалярное условие вызова функции upgrade трансформировалось в аргумент данной функции, и после этого код легко поддается векторизации. В остальном все вещественные скалярные инструкции были просто заменены на векторные аналоги. Также была выполнена векторизация функции upgrade с помощью слияния всех веток выполнения под их предикатами, результирующий код данной функции представлен на листинге ниже.

Листинг 18. Векторная реализация функции upgrade с пропагированным условием вызова внутрь функции.

```
void upgrade(__mmask16 m, __m512 f0, __m512 f1,
             __m512 *lo, __m512 *hi)
{
    __mmask16 c_f0z = _mm512_cmpeq_ps_mask(f0, z0);
    __mmask16 c_f0n = _mm512_cmplt_ps_mask(f0, z0);
    __mmask16 c_f0p = ~(c_f0z | c_f0n);
    __mmask16 c_f1p = _mm512_cmplt_ps_mask(z0, f1);
    __m512 k = _mm512_mask_div_ps(k, ~(m & c_f0z), f1, f0);

    k = SUB(z0, k);
    *lo = _mm512_mask_add_ps(*lo, m & c_f0z & c_f1p, *hi, z1);
    *hi = _mm512_mask_min_ps(*hi, m & c_f0p, *hi, k);
    *lo = _mm512_mask_max_ps(*lo, m & c_f0n, *lo, k);
}
```

Выполненные преобразования позволили добиться ускорения функции $tri\_box\_intersect\_16$ в 6.7 раз [33].

5.3 Векторизация точного римановского решателя задачи распада произвольного разрыва

5.3.1 Описание римановского решателя

Рассматриваемая в данном разделе реализация римановского решателя находится в открытом доступе в сети Интернет в составе библиотеки NUMERICA [34]. Нас в данном случае будет интересовать одномерный случай для однокомпонентной среды, реализованный в виде чистой функции (функции без побочных эффектов, результат работы функции зависит только от значений входных параметров), которая по значениям плотности, скорости и давления газа слева и справа от разрыва, находит значения этих же величин на самом разрыве в нулевой момент времени после устранения перегородки.

$$U_l = \begin{pmatrix} d_l \\ u_l \\ p_l \end{pmatrix}, U_r = \begin{pmatrix} d_r \\ u_r \\ p_r \end{pmatrix}, U = \begin{pmatrix} d \\ u \\ p \end{pmatrix} = riem(U_l, U_r) \tag{5.12}$$

В формуле (5.12) через d_l, u_l, p_l обозначены плотность, скорость и давление газа слева от разрыва (они объединены в структуру U_l – состояние газа слева от разрыва). Аналогично через d_r, u_r, p_r обозначены плотность, скорость и давление газа справа от разрыва, объединенные в состояние газа U_r. Переменными d, u, p обозначены плотность, скорость и давление газа, полученные в результате решения задачи Римана.

Библиотека NUMERICA реализована на языке программирования FORTRAN, поэтому векторизация данного кода с использованием функций-интринсиков напрямую невозможна, поэтому использовалась портированная на язык программирования C версия кода.

На рис. 5.4 показана схема работы римановского решателя с обозначенными потоками данных и вызовами всех входящих в реализацию функций. Функция `riemann` осуществляет вычисление скорости звука справа и слева, выполняет проверку на образование вакуума и последовательно вызывает функции `starpu` и `sample`. Функция `starpu` вычисляет значения скорости и давления в среднем регионе между левой и правой волнами (star region), при этом функция содержит цикл с неизвестным количеством итераций для решения нелинейного уравнения итерационным методом Ньютона, внутри которого расположены

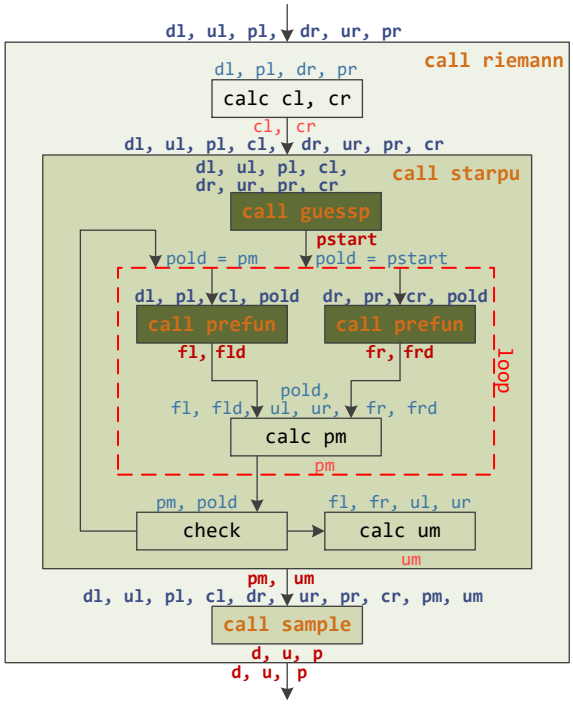

Рис. 5.4. Схема потока данных в римановском решателе.

вызовы других функций (`prefun`). Функции `guessp` и `prefun` содержат только арифметические вычисления и простые условия и являются наиболее простыми с точки зрения векторизации. Наконец последняя функция sample определяет окончательную конфигурацию разрыва путем вычисления множества условий. Данная функция содержит очень разветвленное управление, вложенность условий в ней достигает четырех, что затрудняет применение векторизации.

В процессе счета с помощью численных методов, базирующихся на римановском решателе, выполняется множество вызовов функции `riemann` с различными наборами входных данных (на каждой итерации счета выполняется один вызов для каждой грани каждой ячейки расчетной сетки). Так как функция riemann является чистой, то вызовы для разных наборов входных данных (`dl`, `ul`, `pl`, `dr`, `ur`, `pr`) являются независимыми и возникает желание объединения вызовов с целью эффективного задействования векторных (поэлементных) инструкций. В качестве такого объединенного вызова будем рассматривать функцию, в которую вместо атомарных данных типа `float` будут подаваться соответствующие векторы, содержащие по 16 элементов.

$$\overline{U_l} = \begin{pmatrix} \overline{d_l} \\ \overline{u_l} \\ \overline{p_l} \end{pmatrix}, \overline{U_r} = \begin{pmatrix} \overline{d_r} \\ \overline{u_r} \\ \overline{p_r} \end{pmatrix}, \overline{U} = \begin{pmatrix} \overline{d} \\ \overline{u} \\ \overline{p} \end{pmatrix} = riem(\overline{U_l}, \overline{U_r}) \tag{5.13}$$

В формуле (5.13) все переменные $\overline{d_l}$, $\overline{u_l}$, $\overline{p_l}$, $\overline{d_r}$, $\overline{u_r}$, $\overline{p_r}$, $\overline{d}$, $\overline{u}$, $\overline{p}$ являются векторами длины 16. Например, вектор $\overline{d}$ содержит 16 значений плотности газа, полученных при решении 16 задач Римана, объединенных в один вызов. Аналогично с другими переменными.

При этом с векторными данными можно производить те же действия, что и с базовыми типами – выполнять вычисления, передавать в функции, возвращать в качестве результата. В процессе оптимизации для наглядности будем избегать подстановки тела функции в точку вызова. Таким образом, в результате векторизации нашей целью является полу-

чение векторных аналогов всех используемых в римановском решателе описанных выше функций.

5.3.2 Векторизация простого контекста

Наиболее простым контекстом для векторизации вычислений при объединении вызовов является функция `prefun` (функция `guessp` обладает похожими свойствами и не рассматривается). Невекторизованный вариант функции представлен на листинге 19.

Листинг 19. Оригинальная версия функции `prefun`.

```
void prefun(float &f, float &fd, float &p,
            float &dk, float &pk, float &ck)
{
    float ak, bk, pratio, qrt;

    if (p <= pk)
    {
        pratio = p / pk;
        f = G4 * ck * (pow(pratio, G1) - 1.0);
        fd = (1.0 / (dk * ck)) * pow(pratio, -G2);
    }
    else
    {
        ak = G5 / dk;
        bk = G6 * pk;
        qrt = sqrt(ak / (bk + p));
        f = (p - pk) * qrt;
        fd = (1.0 - 0.5 * (p - pk) / (bk + p)) * qrt;
    }
}
```

Данный код содержит только простые арифметические операции, извлечение квадратного корня, возведение в степень и сравнение. Для всех этих действий в наборе команд AVX-512 предусмотрены векторные аналоги. Код функции векторизуется путем замены арифметических операций на векторные аналоги. Вычисление значений, находящихся под условием (`if-else`) векторизуется путем использования векторных предикатов, полученных с помощью векторного сравнения.

Листинг 20. Векторизованная версия функции `prefun`.

```
void prefun_16(__m512 *f, __m512 *fd, __m512 p,
               __m512 dk, __m512 pk, __m512 ck,
               __mmask16 m)
{
    __m512 pratio, ak, bkp, ppk, qrt;
    __mmask16 cond, ncond;

    // Conditions.
    cond = _mm512_mask_cmp_ps_mask(m, p, pk, _MM_CMPINT_LE);
    ncond = m & ~cond;
```

```
11
12      // The first branch.
13      if (cond != 0x0)
14      {
15          pratio = _mm512_mask_div_ps(z, cond, p, pk);
16          *f = _mm512_mask_mul_ps(*f, cond, MUL(g4, ck),
17                   SUB(_mm512_mask_pow_ps(z, cond, pratio, g1), one));
18          *fd = _mm512_mask_div_ps(*fd, cond,
19                    _mm512_mask_pow_ps(z, cond, pratio, SUB(z, g2)),
20                    MUL(dk, ck));
21      }
22
23      // The second branch.
24      if (ncond != 0x0)
25      {
26          ak = _mm512_mask_div_ps(z, ncond, g5, dk);
27          bkp = FMADD(g6, pk, p);
28          ppk = SUB(p, pk);
29          qrt = _mm512_mask_sqrt_ps(z, ncond,
30                                    _mm512_mask_div_ps(z, ncond, ak,
                                          bkp));
31          *f = _mm512_mask_mul_ps(*f, ncond, ppk, qrt);
32          *fd = _mm512_mask_mul_ps(*fd, ncond, qrt,
33                    FNMADD(_mm512_mask_div_ps(z, ncond, ppk, bkp),
34                           SET1(0.5), one));
35      }
36  }
```

В проведенной векторизации функции `prefun` стоит отметить три момента. Так как векторная операция деления является медленной, то использовались следующие тождества для сокращения количества делений:

$$\frac{a}{\frac{b}{c}} = \frac{ac}{b}, \frac{\frac{a}{b}}{c} = \frac{a}{bc} \tag{5.14}$$

Так как функция `prefun` вызывается внутри цикла в функции `starpu`, то в ее реализацию необходимо добавить дополнительный аргумент-маску, по которой выбираются элементы, обрабатываемые в данном вызове. Более подробно это будет описано ниже, когда будет обсуждаться векторизация функции `starpu`.

Еще одним важным моментом векторизации является оптимизация работы с условиями. В процессе выполнения векторизации у нас сформировались две группы векторных инструкций, выполняемых под предикатами `cond`. Так как для физических задач характерно непрерывное изменение физических величин, то можно утверждать, что наборы данных, обрабатываемых в векторизованном вызове, близки. Другими словами, все векторы, фигурирующие в вычислениях, содержат близкие значения. Это же утверждение относится и к векторам предикатов. Действительно, сбор статистики исполнения показал, что для физических задач наиболее частыми значениями векторов предикатов (в данном случае `cond` и `ncond`) являются значения `0x0` и `0xFFFF`. Таким образом, проверки предикатных векторных регистров на пустоту сразу отсекает целые блоки лишних операций, выполняемых с нулевой маской.

5.3.3 Векторизация сильно разветвленных условий

Функция `sample` содержит сильно разветвленное управление с уровнем вложенности условий, равным 4.

Листинг 21. Оригинальная версия функции `sample`.

```
void sample(float dl, float ul, float pl, float cl,
            float dr, float ur, float pr, float cr,
            const float pm, const float um,
            float &d, float &u, float &p)
{
    float c, cml, cmr, pml, pmr, shl, shr, sl, sr, stl, str;

    if (0.0 <= um)
    {
        if (pm <= pl)
        {
            shl = ul - cl;

            if (0.0 <= shl)
            {
                < d, u, p = dl, ul, pl >
            }
            else
            {
                cml = cl * pow(pm / pl, G1);
                stl = um - cml;

                if (0.0 > stl)
                {
                    d = dl * pow(pm / pl, 1.0 / GAMA);
                    u = um;
                    p = pm;
                }
                else
                {
                    < high-density code, low prob >
                }
            }
        }
        else
        {
            pml = pm / pl;
            sl = ul - cl * sqrt(G2 * pml + G1);

            if (0.0 <= sl)
            {
                < d, u, p = dl, ul, pl >
            }
            else
            {
                d = dl * (pml + G6) / (pml * G6 + 1.0);
                u = um;
```

```
48                  p = pm;
49              }
50          }
51      }
52      else
53      {
54          < symmetrical branch >
55      }
56  }
```

Дерево условий, построенное для данной функции содержит 10 листовых узлов, в каждом из которых определяется значение газодинамических величин `d`, `u`, `p`. Прямое вычисление векторных предикатов всех листовых узлов и выполнение их кода под этими предикатами приводит к замедлению результирующего кода, поэтому при векторизации данной функции выполнялись следующие действия.

Во-первых, было замечено, что 4 линейных участка содержат определение газодинамических параметров `d`, `u` и `p`, которое можно было изменить на инициализацию с помощью выноса операций присваивания вверх по коду функции. Таким образом, 4 линейных участка были удалены. При этом данная инициализация параметров `d`, `u`, `p` не содержит арифметических операций (параметры инициализируются аргументами функции `dl`, `ul`, `pl` или `dr`, `ur`, `pr`), а значит может быть выполнена с помощью векторных операций слияния `blend`.

Далее было отмечено, что функция `sample` обрабатывает одинаковым образом правый и левый профиль распада разрыва с незначительными изменениями. С помощью простой замены переменных, заключающейся в изменении знака значения скорости, удалось выполнить слияние кода для двух поддеревьев, опирающихся на условие $\texttt{pm} \leq \texttt{pl}$, что позволило вдвое уменьшить объем расчетного кода и ожидаемо сократило время исполнения на 45%.

После выполнения сокращения кода количество листовых узлов дерева условий сократилось до трех. Однако даже в этом случае прямое слияние кода с использованием векторных предикатов оказалось неэффективным. Виной тому послужил маловероятный участок кода, тяжелые операции, среди которых присутствуют вызовы функции возведения в степень. Векторный предикат данного участка кода более чем в 95 процентах случаев имеет значение `0x0`, поэтому перед выполнением данного участка кода целесообразно выполнить проверку данного предиката на пустоту (что соответствует выносу маловероятной ветви исполнения из тела функции). Вынос маловероятной ветки исполнения из основного программного контекста способен существенно ускорить исполняемый код, так как наличие большого количества подобных редких вычислений может служить причиной к отказу от векторизации.

Для остального кода можно производить слияние с учетом замечаний, описанных в предыдущем разделе. В результате векторизованная функция `sample` была ускорена более чем в 10 раз. Итоговый векторный код представлен на листинге 22.

Листинг 22. Векторизованная версия функции `sample`.

```
1  void sample_16(__m512 dl, __m512 ul, __m512 pl, __m512 cl,
2                 __m512 dr, __m512 ur, __m512 pr, __m512 cr,
3                 __m512 pm, __m512 um,
4                 __m512 *d, __m512 *u, __m512 *p)
5  {
```

```
    __m512 c, ums, pms, sh, st, s, uc;
    __mmask16 cond_um, cond_pm, cond_sh, cond_st, cond_s, cond_sh_st;

    // d/u/p/c/ums
    cond_um = _mm512_cmp_ps_mask(um, z, _MM_CMPINT_LT);
    *d = _mm512_mask_blend_ps(cond_um, dl, dr);
    *u = _mm512_mask_blend_ps(cond_um, ul, ur);
    *p = _mm512_mask_blend_ps(cond_um, pl, pr);
    c = _mm512_mask_blend_ps(cond_um, cl, cr);
    ums = um;
    *u = _mm512_mask_sub_ps(*u, cond_um, z, *u);
    ums = _mm512_mask_sub_ps(ums, cond_um, z, ums);

    // Calculate main values.
    pms = DIV(pm, *p);
    sh = SUB(*u, c);
    st = FNMADD(POW(pms, g1), c, ums);
    s = FNMADD(c, SQRT(FMADD(g2, pms, g1)), *u);

    // Conditions.
    cond_pm = _mm512_cmp_ps_mask(pm, *p, _MM_CMPINT_LE);
    cond_sh = _mm512_mask_cmp_ps_mask(cond_pm, sh, z, _MM_CMPINT_LT);
    cond_st = _mm512_mask_cmp_ps_mask(cond_sh, st, z, _MM_CMPINT_LT);
    cond_s = _mm512_mask_cmp_ps_mask(~cond_pm, s, z, _MM_CMPINT_LT);

    // Store.
    *d = _mm512_mask_mov_ps(*d, cond_st,
                            MUL(*d, POW(pms, SET1(1.0 / GAMA))));
    *d = _mm512_mask_mov_ps(*d, cond_s MUL(*d, DIV(ADD(pms, g6),
                                                   FMADD(pms, g6, one
                                                       ))));
    *u = _mm512_mask_mov_ps(*u, cond_st | cond_s, ums);
    *p = _mm512_mask_mov_ps(*p, cond_st | cond_s, pm);

    // Low prob - ignore it.
    cond_sh_st = cond_sh & ~cond_st;
    if (cond_sh_st != 0x0)
    {
        *u = _mm512_mask_mov_ps(*u, cond_sh_st, MUL(g5, FMADD(g7, *u,
            c)));
        uc = DIV(*u, c);
        *d = _mm512_mask_mov_ps(*d, cond_sh_st, MUL(*d, POW(uc, g4)))
            ;
        *p = _mm512_mask_mov_ps(*p, cond_sh_st, MUL(*p, POW(uc, g3)))
            ;
    }

    // Final store.
    *u = _mm512_mask_sub_ps(*u, cond_um, z, *u);
}
```

5.3.4 Векторизация гнезда циклов

Наиболее сложным контекстом для векторизации кода является функция `starpu`, содержащая цикл с неизвестным количеством итераций.

Листинг 23. Оригинальная версия функции `starpu`.

```
void starpu(float dl, float ul, float pl, float cl,
            float dr, float ur, float pr, float cr,
            float &p, float &u)
{
    const int nriter = 20;
    const float tolpre = 1.0e-6;
    float change, fl, fld, fr, frd, pold, pstart, udiff;

    guessp(dl, ul, pl, cl, dr, ur, pr, cr, pstart);
    pold = pstart;
    udiff = ur - ul;

    int i = 1;

    for ( ; i <= nriter; i++)
    {
        prefun(fl, fld, pold, dl, pl, cl);
        prefun(fr, frd, pold, dr, pr, cr);
        p = pold - (fl + fr + udiff) / (fld + frd);
        change = 2.0 * abs((p - pold) / (p + pold));

        if (change <= tolpre)
        {
            break;
        }

        if (p < 0.0)
        {
            p = tolpre;
        }

        pold = p;
    }

    if (i > nriter)
    {
        cout << "divergence in Newton-Raphson iteration" << endl;
        exit(1);
    }

    u = 0.5 * (ul + ur + fr - fl);
}
```

Цикл, расположенный в данной функции, кроме неизвестного количества итераций содержит также условные переходы (`if`, `break`) и вызовы функций prefun, что также

усложняет его векторизацию. Перед выполнением векторизации данный цикл необходимо преобразовать в предикатную форму, в которой тело не должно содержать операций перехода. Все инструкции цикла выполняются под своими предикатами, а выполнение цикла прерывается при условии обнуления всех предикатов. При этом стоит заметить, что вызовы функций prefun также должны обладать соответствующими предикатами. После преобразования тела цикла в предикатную форму, он может быть векторизован, после чего предикаты инструкций заменятся на векторные регистры-маски (именно в этом месте появляется дополнительный параметр векторизованной функции `prefun` в виде маски). Результат векторизации функции `starpu` представлен на листинге ниже.

Листинг 24. Векторизованная версия функции `starpu`.

```
void starpu_16(__m512 dl, __m512 ul, __m512 pl, __m512 cl,
               __m512 dr, __m512 ur, __m512 pr, __m512 cr,
               __m512 *p, __m512 *u)
{
    __m512 two, tolpre, tolpre2, udiff, pold, fl, fld, fr, frd,
        change;
    __mmask16 cond_break, cond_neg, m;
    const int nriter = 20;
    int iter = 1;

    two = SET1(2.0);
    tolpre = SET1(1.0e-6);
    tolpre2 = SET1(5.0e-7);
    udiff = SUB(ur, ul);

    guessp_16(dl, ul, pl, cl, dr, ur, pr, cr, &pold);

    // Start with full mask.
    m = 0xFFFF;

    for (; (iter <= nriter) && (m != 0x0); iter++)
    {
        prefun_16(&fl, &fld, pold, dl, pl, cl, m);
        prefun_16(&fr, &frd, pold, dr, pr, cr, m);
        *p = _mm512_mask_sub_ps(*p, m, pold,
                                _mm512_mask_div_ps(z, m,
                                                   ADD(ADD(fl, fr),
                                                       udiff),
                                                   ADD(fld, frd)));
        change = ABS(_mm512_mask_div_ps(z, m, SUB(*p, pold),
                                        ADD(*p, pold)));
        cond_break = _mm512_mask_cmp_ps_mask(m, change,
                                             tolpre2, _MM_CMPINT_LE);
        m &= ~cond_break;
        cond_neg = _mm512_mask_cmp_ps_mask(m, *p, z, _MM_CMPINT_LT);
        *p = _mm512_mask_mov_ps(*p, cond_neg, tolpre);
        pold = _mm512_mask_mov_ps(pold, m, *p);
    }

    // Check for divergence.
    if (iter > nriter)
```

```
40      {
41          cout << "divergence in Newton-Raphson iteration" << endl;
42          exit(1);
43      }
44
45      *u = MUL(SET1(0.5), ADD(ADD(ul, ur), SUB(fr, fl)));
46  }
```

На листинге видна изначальная инициализация полной маски выполнения векторизованных итераций цикла. По мере работы цикла маска истощается, и при полном ее обнулении цикл завершает работу.

Стоит отметить, что векторизация цикла с неизвестным числом итераций может быть довольно опасной, так как количество итераций векторизованного цикла равно максимуму из количеств итераций циклов из 16 объединяемых вызовов оригинальной невекторизованной функции. При большой разнице в количестве итераций оригинального кода возникает падение эффективности.

5.3.5 Анализ результатов

Описанные в разделе подходы к векторизации функций римановского решателя были реализованы на языке программирования C с использованием функций-инстринсиков и опробованы на микропроцессорах Intel Xeon Phi 7290.

Тестирование производительности выполнялось на массивах входных данных, собранных при решении стандартных тестовых задач: задача Сода, задача Лакса, задача о слабой ударной волне, задача Эйнфельдта, задача Вудворда-Колелла, задача Шу-Ошера и других.

Можно отметить, что эффект от векторизации простого контекста варьируется в пределах от 5.1 до 5.8 раза (для функций `guessp` и `prefun`). Также следует отметить существенный эффект от оптимизации условий (проверка на пустоту маски предикатов, под которой находится выполнение блока операций). Это довольно простое преобразование, приводит к ускорению кода от 1.4 до 1.7 раз (для функций `sample` и `prefun`) в зависимости от того, насколько близкими являются условия с соседних итераций векторизуемого цикла.

Отдельно на диаграмме выделен эффект от применения оптимизации замены переменных, позволившей в 1.8 раз ускорить функцию `sample` путем слияния двух поддеревьев графа потока управления (то есть было выполнено удаление дублирующих линейных участков).

В результате применения всех описанных оптимизаций удалось достичь ускорения отдельных участков выполнения программы в 10 и более раз, а суммарное ускорение всего римановского решателя составило 7 раз [35, 36]. Можно также отметить работы [37, 38] по данному направлению.

Список литературы

[1] **Rettinger C., Godenschwager C., Eibl S., et al.** Fully Resolved Simulations of Dune Formation in Riverbeds. // J. M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017, vol. 10266, pp. 3—21.

[2] **Krappel T., Riedelbauch S.** Scale Resolving Flow Simulations of a Francis Turbine Using Highly Parallel CFD Simulations. // W. E. Nagel et al. (Eds.): High Performance Computing in Science and Engineering'16, 2016, pp. 499–510.

[3] **Markidis S., Peng I. B., Träff J. L., et al.** The EPiGRAM Project: Preparing Parallel Programming Models for Exascale. // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016, vol. 9945, pp. 56—68.

[4] **Klenk B., Froning H.** An Overview of MPI Characteristics of Exascale Proxy Applications. // J. M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017, vol. 10266, pp. 217-–236.

[5] **Abduljabbar M., Markomanolis G. S., Ibeid H., et al.** Communication Reducing Algorithms for Distributed Hierarchical N-Body Problems with Boundary Distributions. // J. M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017, vol. 10266, pp. 79—96.

[6] **Van der Wijngaart R. F., Georganas E., Mattson T. G., et al.** A New Parallel Research Kernel to Expand Research on Dynamic Load-Balancing Capabilities. // J. M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017, vol. 10266, pp. 256—274.

[7] **Heller T., Kaiser H., Diehl P. et al.** Closing the Performance Gap with Modern C++. // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016, vol. 9945, pp. 18—31.

[8] **Роганов В. А., Осипов В. И., Матвеев Г. А.** Решение задачи Дирихле для уравнения Пуассона методом Гаусса-Зейделя на языке параллельного программирования T++. // Программные системы: теория и приложения, 2016, Том 7, Выпуск 3, с. 99–107.

[9] **Bramas B.** A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. // arXiv: 1704.08579 [cs.MS], https://arxiv.org/abs/1704.08579 (дата обращения 04.12.2019)

[10] **Соколов А. П., Щетинин В. Н., Сапелкин А. С.** Параллельный алгоритм реконструкции поверхности прочности композиционных материалов для архитектуры Intel MIC (Intel Many Integrated Core Architecture). // Программные системы: теория и приложения, 2016, Том 7, Выпуск 2, с. 3–25.

[11] Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. // Intel, October 2019.

[12] **Shabanov B. M., Rybakov A. A., Shumilin S. S.** Vectorization of high-performance scientific calculations using AVX-512 instruction set. // Lobachevskii Journal of Mathematics, 2019, Vol. 40, No. 5, p. 580–598.

[13] **Кузьминский М.** Из ускорителей в процессоры. // «Открытые системы. СУБД», 2016, вып. №3, https://www.osp.ru/os/2016/03/13050252/ (дата обращения 04.12.2019).

[14] **Jeffers J., Reinders J., Sodani A.** Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition. // Morgan Kaufmann, 2016.

[15] Intel представила новые 14-нанометровые процессоры Xeon Phi x205 (Knights Mill). // https://ichip.ru/novosti/intel-predstavila-novye-14-nanometrovye-processory-xeon-phi-x205-knights-mill-129566 (дата обращения 04.12.2019).

[16] Intel опубликовала данные о Xeon Phi поколения Knights Mill. // https://overclockers.ru/hardnews/show/88589/intel-opublikovala-dannye-o-xeon-phi-pokoleniya-knights-mill (дата обращения 04.12.2019).

[17] Продукция с прежним кодовым названием Knights Mill. // https://ark.intel.com/content/www/ru/ru/ark/products/codename/57723/knights-mill.html (дата обращения 04.12.2019).

[18] Intel представила серверные процессоры Xeon Skylake-SP. // https://overclockers.ru/hardnews/show/85513/intel-predstavila-servernye-processory-xeon-skylake-sp (дата обращения 04.12.2019).

[19] **Осколков И.** Знакомство с Intel Xeon Skylake-SP: смешать, но не взбалтывать. // https://servernews.ru/955164 (дата обращения 04.12.2019).

[20] Detailed Specifications of the "Skylake-SP" Intel Xeon Processor Scalable Family CPUs. // https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-skylake-sp-intel-xeon-processor-scalable-family-cpus/ (дата обращения 04.12.2019).

[21] Drilling down into the Xeon Skylake Architecture. // https://www.nextplatform.com/2017/08/04/drilling-xeon-skylake-architecture/ (дата обращения 04.12.2019).

[22] 2nd Generation Intel® Xeon® Scalable Processors with Intel® C620 Series Chipsets (Purley Refresh). // https://www.intel.com/content/www/us/en/design/products-and-solutions/processors-and-chipsets/cascade-lake/2nd-gen-intel-xeon-scalable-processors.html (дата обращения 04.12.2019).

[23] Обзор Intel Xeon Cascade Lake. // https://771xeon.ru/obzor-intel-xeon-cascade-lake/ (дата обращения 04.12.2019).

[24] **Осколков И.** Intel Xeon Cascade Lake: вы находитесь здесь. // https://servernews.ru/985088 (дата обращения 04.12.2019).

[25] Обзор: процессоры Intel Cascade Lake Xeon и память Optane DC Persistent Memory. // https://www.hardwareluxx.ru/index.php/artikel/hardware/prozessoren/46856-intel-cascade-lake-xeon-optane.html (дата обращения 04.12.2019).

[26] «Найди пять отличий». Разница поколений Scalable и — новая порция тестов. // https://habr.com/ru/company/first/blog/457496/ (дата обращения 04.12.2019).

[27] Using the GNU Compiler Collection. For GCC version 7.4.0. // Richard M. Stallman and the GCC Developer Community.

[28] Intel C++ Compiler 16.0 User and Reference Guide. // https://software.intel.com/en-us/download/intel-c-compiler-160-update-4-user-and-reference-guide (дата обращения 04.12.2019)

[29] Intel Intrinsics Guide. // https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (дата обращения 04.12.2019)

[30] **Бендерский Л. А., Рыбаков А. А., Шумилин С. С.** Векторизация перемножения малоразмерных матриц специального вида с использованием инструкций AVX-512. // Международный научный журнал «Современные информационные технологии и ИТ-образование», 2018, Т. 14, № 3, с. 594–602.

[31] **Зоркальцев В. И., Киселева М. А.** Системы линейных неравенств. // Учебное пособие, Иркутск, Издательство Иркутского государственного университета, 2007.

[32] **Черников Н. С.** Свертывание конечных систем линейных неравенств. // Доклады АН СССР, 1963, Т. 152, № 5, 1075–1078.

[33] **Рыбаков А. А.** Векторизация нахождения пересечения объемной и поверхностной сеток для микропроцессоров с поддержкой AVX-512. // Труды НИИСИ РАН, 2019, № 5.

[34] NUMERICA, A Library of Sources for Teaching, Research and Applications, by E. F. Toro // https://github.com/dasikasunder/NUMERICA (дата обращения 04.12.2019)

[35] **Rybakov A. A., Shumilin S. S.** Vectorization of the Riemann solver using the AVX-512 instruction set. // Program Systems: Theory and Applications, 2019, Vol. 10, № 3 (42), p. 41–58.

[36] **Рыбаков А. А., Шумилин С. С.** Векторизация римановского решателя с использованием набора инструкций AVX-512. // Программные системы: Теория и приложения, 2019, Т. 10, № 3 (42), с. 59–80.

[37] **Bader M., Breuer A., Höltz W., Rettenberger S.** Vectorization of an augmented Riemann solver for the shallow water equations. // Proceedings of the 2014 International Conference on High Performance Computing and Simulation, HPCS 2014, 2014, pp. 193–201.

[38] **Ferreira C. R., Mandli K. T., Bader M.** Vectorization of Riemann solvers for the single- ans multi-layer shallow water equations. // Proceedings of the 2018 International Conference on High Performance Computing and Simulation, HPCS 2018, 2018, pp. 415–422.

ПОДГОТОВЛЕНО:

Ведущий научный сотрудник А.А. Рыбаков

Младший научный сотрудник С.С. Шумилин

СОГЛАСОВАНО:

Заместитель директора П.Н. Телегин

Главный инженер В.М. Опалев

Главный программист О.И. Вдовикин