УДК 004.42, 004.051 doi: 10.15827/0236-235X.142.561-572 2023. Т. 36. № 4. С. 561–572

Создание инструментария для векторизации тела плоского цикла с помощью векторных инструкций AVX-512

А.А. Рыбаков
А.Н. Швиндт

Ссылка для цитирования
Рыбаков А.А., Швиндт А.Н. Создание инструментария для векторизации тела плоского цикла с помощью векторных инструкций AVX-512 // Программные продукты и системы. 2023. Т. 36. № 4. С. 561–572. doi: 10.15827/0236-235X.142.561-572
Информация о статье
Поступила в редакцию: 23.08.2023 После доработки: 01.09.2023 Принята к публикации: 04.09.2023

Аннотация. Работа направлена на изучение подходов по сокращению времени исполнения программ с помощью инструкций из набора AVX-512 для повышения эффективности векторизации программного кода. Векторизация является основной низкоуровневой оптимизацией, с помощью которой возможно кратно ускорить выполнение программы, а набор инструкций AVX-512 обладает рядом уникальных особенностей, позволяющих применять векторизацию в сложных программных контекстах. В работе исследуется программный контекст специального вида – плоский цикл, который при удовлетворении ряда требований может быть векторизован с помощью инструкций AVX-512 при практически произвольной сложности тела. Однако довольно часто оптимизирующий компилятор не в состоянии выполнить автоматическую векторизацию плоских циклов по причине наличия в них сложного управления, вызовов функций, гнезд циклов и других конструкций. Это приводит к тому, что векторизацию приходится выполнять вручную с использованием специальных функций-интринсиков. Эффективность векторизации напрямую зависит от плотности масок векторных операций, которые оказываются разреженными при сильно разветвленном управлении внутри тела плоского цикла. В работе предлагается инструментарий, позволяющий параллельно создавать скалярную и векторную версии плоского цикла и анализировать эффективность выполненной векторизации. При этом поддерживаются два режима сборки тестируемого кода: скалярная версия с эмуляцией и мониторингом векторных инструкций и векторная версия для исполнения на целевой машине. С одной стороны, это позволяет контролировать корректность выполненных преобразований кода, а с другой – отслеживать пути исполнения программы с низкой вероятностью для их локализации или выноса из тела цикла. С помощью созданного инструментария удалось выделить горячие плоские циклы ряда реальных приложений и повысить эффективность их векторизации.

Ключевые слова: векторизация, плоский цикл, AVX-512, низкоуровневая оптимизация, компьютерные приложения
Благодарности. Работа выполнена в МСЦ РАН в рамках государственного задания по теме FNEF-2022-0016. В исследованиях использован суперкомпьютер МВС-10П

Введение. Векторизация является основной низкоуровневой оптимизацией, с помощью которой можно достичь кратного ускорения компьютерных приложений. Использование векторных инструкций позволяет применять одинаковые операции сразу к нескольким наборам входных данных, упакованных в векторные регистры. На сегодняшний день наиболее продвинутым набором векторных инструкций является AVX-512, поддержку которого можно встретить в микропроцессорах Intel и AMD. Инструкции AVX-512 работают с векторными регистрами размера 512 бит, каждый из которых вмещает в себя 8 элементов в формате вещественных чисел двойной точности, что с учетом комбинированных операций приводит к возможности выполнения 16 операций сложения и умножения за одну векторную команду. Из этого следует, что на микропроцессорах с поддержкой AVX-512 без использования этих инструкций даже теоретически невозможно добиться производительности более 6,25 % от пиковой. Особенности векторных инструкций AVX-512 позволяют применять их для векторизации достаточно сложного программного контекста, однако оптимизирующий компилятор не всегда успешно справляется с этой задачей. Для обеспечения программисту возможности прямого использования векторных инструкций существует набор функций-интринсиков, которые в дальнейшем заменяются компилятором на конкретные векторные инструкции или последовательности инструкций. Использование интринсиков значительно упрощает разработку векторизованного кода, однако этот процесс все еще остается слишком трудоемким.

Можно считать, что впервые набор векторных инструкций AVX-512 был поддержан в 2016 году в микропроцессорах Intel Xeon Phi Knights Landing (так как более раннее поколение Intel Xeon Phi Knights Corner представляет собой ускоритель и векторный код для него не является x86 совместимым). С этого момента

вопросы векторизации приложений активно обсуждаются в научных статьях. Отметим наиболее заметные актуальные работы.

В качестве практического руководства по созданию векторизованного программного кода с помощью функций-интринсиков для языка программирования C++ и с помощью языка ассемблера для широкого спектра практических задач, включая задачи линейной алгебры, обработки изображений, реализацию сверток и другие, можно рассматривать работу [1].

Среди научно-исследовательских статей в различных областях применения можно отметить [2], посвященную разработке векторизованной версии быстрой сортировки (vqsort) с помощью наборов векторных инструкций AVX2 и AVX-512, превышающей по эффективности реализацию из стандартной библиотеки (qsort) более чем в 10 раз. В работе [3] рассматриваются реализация бессеточного метода решения задачи газовой динамики и векторизация данного метода. Использование векторных инструкций AVX-512 позволило ускорить исходный код в 6,3 раза. В статье [4] описано успешное применение векторизации для алгоритма анализа генетических вариаций. Использование векторных инструкций AVX-512 позволило добиться ускорения расчетов от 7 до 12 раз по сравнению с оригинальным алгоритмом (алгоритм базируется на обработке данных формата Byte, что объясняет такое серьезное ускорение). В работе [5] рассматривается прямое применение инструкций AVX-512 для задачи обработки временных рядов, что привело к ускорению результирующего кода в 4 раза по сравнению с автоматической векторизацией, выполняемой оптимизирующим компилятором. Статья [6] посвящена оптимизации алгоритмов шифрования с открытым ключом, в частности, рассматривается оптимизация блочного варианта умножения Монтгомери. Было достигнуто ускорение основных операций от 1,5 до 1,9 раза. В статье [7] показана реализация алгоритма блочного шифрования на различных архитектурах, включая GPU и CPU. В частности, в рамках данной работы алгоритм блочного шифрования был реализован с помощью инструкций AVX-512, что позволило ускорить его в 9,5 раза по сравнению с оригиналом. В статье [8] рассматривается применение векторных инструкций для оптимизации математических операций, используемых в протоколе обмена ключами с применением суперсингулярных изогений (SIKE). В результате выполненных усовершенствований зафиксировано ускорение отдельных операций в диапазоне от 1,5 до 3,5 раза. В работе [9] описано использование инструкций AVX-512 для оптимизации быстрого преобразования Фурье. С помощью функций-интринсиков для векторных инструкций было достигнуто ускорение примерно в полтора раза по сравнению с оригинальной версией. Работа [10] посвящена применению подмножества инструкций Integer Fused Multiply-Add (AVX-512 IFMA) для повышения эффективности реализации деления больших целых чисел. В результате оптимизации было достигнуто ускорение требуемого функционала на 25–35 %. В работе [11] продемонстрирован практический подход к векторизации расчета попарного взаимодействия множества частиц. Использование инструкций AVX-512 в явном виде привело к ускорению исполнения программы в 3,3 раза.

В статье [12] предлагается подход к развитию средств автоматической векторизации расчетных циклов, основанный на анализе и эквивалентных преобразованиях графа зависимостей между операциями, расположенными в теле цикла. Этот подход позволяет переупорядочить операции внутри цикла, снизив их количество и укоротив критический путь исполнения [13], и сгруппировать для объединения в векторные инструкции. При векторизации тела циклов с помощью набора инструкций AVX-512 количество и вид используемых операций не являются препятствием для векторизации. Основная проблема – наличие в теле цикла операций передачи управления и вызовов функций. Однако набор инструкций AVX-512 позволяет оперировать даже таким программным контекстом [14].

Векторные инструкции AVX-512

Набор инструкций AVX-512 – это 512-битное расширение 256-битных AVX-инструкций из набора команд Intel x86, поддержанное в семействах микропроцессоров, начиная с Intel Xeon Skylake и Intel Xeon Phi Knights Landing. Поддержка данного набора инструкций внедряется и для микропроцессоров AMD.

Множество инструкций AVX-512 состоит из нескольких подмножеств, их перечень расширяется с появлением новых поколений микропроцессоров (при этом в различных поколениях микропроцессоров поддержаны разные наборы подмножеств инструкций AVX-512) [15]. Перечислим только некоторые из этих подмножеств. Основной набор векторных инструкций

с поддержкой маскирования – AVX-512F (Foundation) – включает в себя векторные аналоги для большинства арифметических операций, которые используются в расчетах. Инструкции предварительной подкачки данных из памяти – AVX-512PF (PreFetch) – были поддержаны в микропроцессорах Intel Xeon Phi KNL для оптимизации загрузки данных. Команды для вычисления экспоненты и обратных значений – AVX-512ER (Exponential and Reciprocal) – крайне полезный набор, позволяющий векторизовать вычисления, содержащие, в частности, операции возведения в степень. Инструкции для определения конфликтов, которые используются для анализа пересечения адресов обращений в память, – AVX-512CD (Conflict Detection). Дополнительные инструкции для работы с данными размера Byte, Word, Double-word, Quad-word – AVX-512BW, AVX-512DQ.

Из других подмножеств можно упомянуть команды для работы с 52-битными целочисленными значениями, специальные команды для работы с нейросетями и AES-шифрованием, реализацию арифметики полей Галуа, имплементацию специальных битовых операций, а также новый класс комбинированных операций, позволяющий еще больше повысить пиковую производительность процессоров.

Для поддержки выборочного применения операций над упакованными данными к конкретным элементам вектора 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 k0–k7 используются в командах для осуществления условной операции над элементами упакованных данных (если соответствующий бит выставлен в 1, то операция выполняется, а точнее – результат операции записывается в соответствующий элемент вектора назначения) или для слияния элементов данных в регистр назначения. Также маски можно использовать для выборочного чтения из памяти и записи в память элементов векторов, для аккумулирования результатов логических операций над элементами векторов.

По схеме работы можно выделить несколько групп операций AVX-512. Упакованные операции с одним операндом zmm (512-битный вектор) и одним результатом zmm получают на вход один вектор и применяют к каждому его элементу конкретную функцию, получая результат того же размера, который по маске записывается в выходной вектор. Примерами таких операций являются получение абсолютного значения, извлечение корня, округление, операции сдвигов на фиксированное количество разрядов и другие. Упакованные операции с двумя операндами zmm и одним результатом zmm отличаются от предыдущего класса только тем, что применяемая функция является бинарной. К данной группе относятся операции поэлементного сложения, вычитания, умножения, деления, сдвига на переменное количество разрядов и другие. Упакованные операции с двумя операндами zmm и результатом маской выполняют поэлементное сравнение двух векторов и записывают полученный результат в маску. Операции конвертации предназначены для преобразования элементов вектора из одного формата в другой, к ним относятся наборы команд cvt и pack. Упакованные комбинированные операции принимают на вход сразу три zmm-вектора, a, b, c, и поэлементно вычисляют значения вида $\pm a \cdot b \pm c$, которые по маске записываются в выходной вектор. Операции перестановок не выполняют арифметические действия, а только переставляют части вектора в произвольном порядке, определяемом типом операции и дополнительными параметрами. Данная группа представлена большим набором разнообразных операций – unpck, shuf, align, blend, perm. Операции пересылок предназначены для перемещения последовательных данных между регистрами, а также между памятью и регистром. Поддержаны также операции пересылки элементов данных, расположенных не последовательно, а с произвольными смещениями от заданного базового адреса в памяти (операции gather и scatter), а также операции пересылки с дублированием элементов, позволяющие переместить одно значение сразу в несколько элементов вектора. Операции предварительной подкачки данных используются для увеличения вероятности того, что к моменту исполнения команды данные уже будут в кэше памяти. Кроме того, поддержаны другие операции с более сложной логикой, среди которых определение класса вещественного числа, реализация логических функций от трех аргументов, операции определения конфликтов и другие.

С целью упрощения векторизации исходного кода для компилятора icc разработаны специальные функции – интринсики, определенные в заголовочном файле immintrin.h. Они покрывают практически все инструкции AVX-512, избавляют от необходимости вруч-

ную писать ассемблерный код и позволяют использовать встроенные типы данных для 512-битных векторов (_m512, _m512i, _m512d) и масок (_mmask8, _mmask16 и другие). Некоторые интринсики соответствуют не отдельной команде, а целой последовательности. Примером могут служить функция сложения всех элементов вектора (_mm512_reduce_add_ps) и другие reduce-операции. Из множества интринсиков можно выделить различные группы функций, схожих по структуре. Для большинства операций AVX-512 реализованы соответствующие инстринсики, раскрывающиеся в одну конкретную операцию. Среди них арифметические и 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 интринсики, особенно предназначенные для выполнения упакованных трансцендентных операций, раскрываются просто в вызов библиотечной функции (например, _mm512_log_ps, _mm512_hypot_ps, тригонометрические функции). Полный перечень всех доступных функций-интринсиков размещена по адресу https://www.laruence.com/sse/.

Понятие плоского цикла

Векторизация программного контекста автоматически не может быть применена к коду произвольного вида. Определим подходящий для векторизации программный контекст специального вида – плоский цикл – и опишем его свойства.

На рисунке 1 слева продемонстрирована функции fun, которая получает на вход три аргумента – a, b и c и на их основании формирует результат, состоящий из трех значений, – *x*, *y*, *z*. Будем считать, что результат выполнения этой функции зависит только от значений a, b и c (например, отсутствуют побочные эффекты через глобальную память или операции ввода-вывода). Если рассмотреть вместо одного вызова функции fun несколько вызовов (*w* штук) с разными наборами входных параметров, то их можно трактовать как вызов некоторой функции FUN, входными параметрами которой являются векторы A, B и C длины *w*, а результатом – набор из векторов X, Y и Z также длины *w*. В данном случае можно говорить, что функция FUN представляет собой реализацию векторизованной функции fun при ширине векторизации *w* (рис. 1 справа). В этом случае семантику функции FUN можно записать следующим образом:

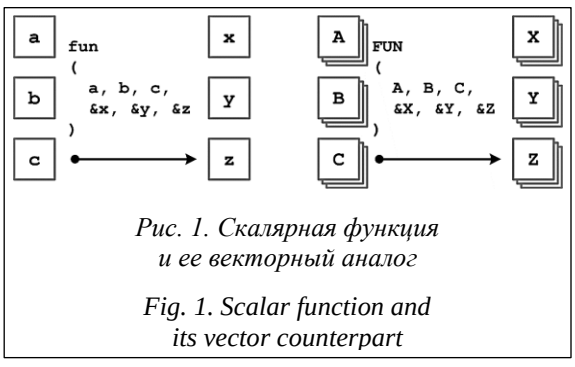

Рис. 1. Скалярная функция и ее векторный аналог

Fig. 1. Scalar function and its vector counterpart

```
// Плоский цикл в скалярном виде.
for (int i = 0; i < w; i++)
{
  fun(A[i], B[i], C[i], X[i], Y[i], Z[i];
}

// Плоский цикл в векторном виде.
FUN(A, B, C, X, Y, Z);
```

Цикл такого вида будем называть плоским. Определим характеристики, присущие плоскому циклу. Будем считать, что плоский цикл – это цикл for, индуктивная переменная которого меняется от 0 до $w - 1$, где w – ширина векторизации (например, при работе с набором инструкций AVX-512 и с вещественными значениями формата single precision ширина векторизации равна 16, то есть один zmm-регистр вмещает 16 значений), а также, что внутри плоского цикла на i-й итерации все обращения в память (и вообще все обращения к глобальным данным) имеют вид a[i]. Такое ограничение гарантирует отсутствие конфликтов между итерациями при работе с памятью. Таким образом, итерации плоского цикла становятся независимыми между собой, то есть могут выполняться в произвольном порядке, в том числе и одновременно.

В случае нарушения требования на вид обращения в память будем называть цикл квазиплоским. Для таких циклов все еще доступны некоторые приемы векторизации. Так, в [16] рассматривается подход к векторизации сортировки Шелла, реализованной в виде гнезда из трех циклов. В данной реализации сортировки запись в память имеет вид a[i + off]. Это приводит к необходимости использовать векторную операцию множественной записи в память scatter, что в совокупности с нерегулярным количеством итераций вложенных циклов приводит к очень скромным результатам по ускоре-

нию (примерно в 2 раза). В данной работе такие квазиплоские циклы не рассматриваются.

Плоские циклы, обладающие описанными свойствами, представляют собой удобный контекст для векторизации, и в большинстве случаев они могут быть векторизованы с помощью векторных инструкций AVX-512 путем перевода тела цикла в предикатное представление [17] и замены скалярных инструкций векторными аналогами, реализованными с помощью интринсиков. Многие практические вычислительные задачи состоят из выполнения однотипных вычислений, применяемых к разным наборам данных, которые можно сгруппировать, трансформировав в плоский цикл, как это продемонстрировано на рисунке 1 на примере функции fun.

Слияние ветвей исполнения при векторизации

Основной причиной потери производительности при векторизации плоского цикла является наличие в теле цикла операций передачи управления (конструкции if, else, switch и другие). Наличие команд передачи управления означает, что внутри итерации цикла некоторые инструкции, до которых управление не дойдет, не должны исполняться. В векторизованной версии цикла это означает, что некоторые векторные инструкции должны обрабатывать не все свои элементы, а только часть их, что регулируется масками. Если назвать плотностью маски отношение количества в ней единичных битов к общей длине маски, то можно отметить, что эффективность векторизации коррелирует с плотностью маски (при низкой плотности масок векторных операций эффективность векторного кода не может быть высокой, обратное не всегда верно).

Рассмотрим векторизацию простого программного контекста – тела плоского цикла, состоящего из двух линейных участков (left и right), выполняющихся под противоположными условиями (предикатами). Пусть длины линейных участков равны t и αt, где под длиной линейного участка можно понимать количество операций в нем, стоящих на критическом пути исполнения (рис. 2). Пусть вероятность выполнения условия для перехода на первый линейный участок равна p, тогда вероятность условия для перехода на второй линейный участок равна $(1 - p)$ (рис. 2а). Если количество итераций рассматриваемого цикла равно w, то общее время выполнения скалярного варианта цикла $T_1 = (p + (1 - p)\alpha)tw$.

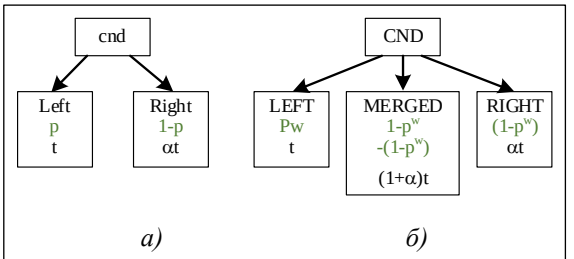

Рис. 2. Слияние двух линейных участков при векторизации

Fig. 2. Merging two linear sections during vectorization

Пусть необходимо векторизовать рассматриваемый цикл с длиной векторизации w. В общем случае в блоке CND должна быть сформирована маска длины w, каждый бит которой соответствует выполнению блока left или right в скалярной версии цикла. После этого для всех скалярных инструкций из блоков left и right должны быть взяты векторные аналоги под противоположными масками и помещены в единый блок MERGED. Тогда длина блока MERGED, а значит, и время выполнения векторизованного цикла, будет $T_w = (1 + \alpha)t$. В этом случае теоретическая эффективность векторизации составит

$$e = \frac{T_1}{T_w w} = \frac{p + (1 - p)\alpha}{1 + \alpha}.$$

Заметим, что в случае одинаковых по длине участков left и right ($\alpha = 1$) эффективность векторизации равна 0,5 даже при стремлении вероятности выполнения одного из этих участков к нулю ($p = 0$ или $p = 1$).

Если присутствует дополнительная информация об условии выбора линейного участка в скалярной версии цикла, то при векторизации можно произвести дополнительную оптимизацию. Зачастую в расчетных приложениях условия выбора линейного участка на разных итерациях плоского цикла не являются независимыми. Более того, данные условия могут меняться достаточно медленно при переходе от одной итерации к другой. Это приводит к тому, что маска, сформированная в блоке CND, нередко оказывается пустой или, наоборот, полной. Это означает, что в скалярном варианте цикла на каждой из w итераций всегда выполняется линейный участок left или right. Для повышения эффективности векторной версии цикла может быть создан векторизованный блок LEFT, который будет выполняться с вероятностью p^w. Аналогично для линейного участка right может быть создан векторизован-

ный блок RIGHT, который будет выполняться с вероятностью $(1 - p)^w$. Таким образом, вероятность выполнения векторизованного участка MERGED, содержащего все векторизованные операции тела цикла, составит $1 - p^w - (1-p)^w$. Посчитав математическое ожидание длины векторизованного тела цикла, содержащего все три участка – LEFT, MERGED и RIGHT, получим теоретическую эффективность векторизации:

$$e = \frac{T_1}{T_w w} = \frac{p + \alpha(1-p)}{\left(1-(1-p)^w\right) + \alpha\left(1-p^w\right)}.$$

В этом случае при одинаковых по длине участках left и right и при стремлении вероятности выполнения одного из них к нулю эффективность векторизации стремится к единице. С другой стороны, такое преобразование требует введения дополнительных проверок маски на пустоту или полноту (маски можно сравнивать как обычные беззнаковые целые числа), а также дублирования кода, поэтому выполнять его следует ограниченно и при наличии информации мониторинга о вероятностях переходов между линейными участками.

Отдельно следует отметить, что набор векторных инструкций AVX-512 содержит ряд инструкций, логика которых соответствует конструкциям с передачей управления. Это операции abs (реализуется с помощью обнуления знакового бита), min, max, blend (см. таблицу).

Инструкции, семантика которых отражает передачу управления

Instructions whose semantics reflect control transfer

Инструкция	Семантика
abs(x)	(x > 0) ? x : -x
min(x, y)	(x <= y) ? x : y
max(x, y)	(x >= y) ? x : y
blend(cnd, x, y)	cnd ? x : y

Указанные в таблице инструкции с семантикой, содержащей передачу управления, имеют векторные аналоги в AVX-512, поэтому с их помощью можно векторизовать другие похожие операции. Например, операция to_bounds(x, lo, hi) приведения величины в заданный диапазон может быть выражена как to_bounds(x, lo, hi) = = min(hi, max(lo, x)), а значит, и векторизована. Конечно, наиболее ценной среди этих операций является blend, так как с ее помощью можно производить слияние различных линейных участков с произвольной степенью разветвленности управления.

Пример выполнения векторизации кода с сильно разветвленным управлением

В качестве примера векторизации плоского цикла с сильно разветвленным управлением рассмотрим цикл, тело которого включает нахождение минимального положительного корня квадратного уравнения (рис. 3, слева). Данный программный контекст с обработкой всех краевых случаев содержит большое количество линейных участков при их небольшой длине, поэтому прямое слияние всех этих участков крайне неэффективно.

Для анализа возможностей векторизации тела данного плоского цикла рассмотрим его граф потока управления [18] – ориентированный граф, узлами которого являются линейные участки программы, связанные между собой операциями передачи управления. В графе потока управления выделен начальный узел (start node), в который не входит ни одна дуга, а также конечный (stop node), из которого не выходит ни одна дуга. Таким образом, граф потока управления является гамаком. На рисунке 4 слева представлен граф потока управления рассматриваемого фрагмента кода. Можно заметить, что большое количество линейных участков рассматриваемого графа может быть удалено. Например, все линейные участки, обнуляющие величину *h*, могут быть удалены при условии, что переменная *h* инициализирована нулем в стартовом узле. Также некоторые линейные участки могут быть непосредственно слиты с помощью семантики операций max и blend (на рисунке 4 сливаемые в один узел регионы отмечены пунктиром). После выполнения преобразований по уменьшению количества линейных участков получим граф потока управления в более компактном виде (рис. 4, справа).

Следующим шагом векторизации является запись полученного кода в предикатной форме, то есть без операций передачи управления, где для каждой операции указано условие ее выполнения (рис. 3, справа сверху). После записи кода в предикатной форме остается один механический шаг – замена всех скалярных инструкций векторными аналогами. При этом предикаты скалярных операций при векторизации трансформируются в векторные маски. Результирующий код представлен на рисунке 3 (справа внизу).

Векторизация плоских циклов с помощью инструмента flatvec

Подход к векторизации плоского цикла путем перевода скалярной версии в предикатную

```
// Исходный скалярный
код тела цикла.
if (a == 0.0)
{
    if (b == 0.0)
    {
        h = 0.0;
    }
    else
    {
        h = -c / b;
        if (h < 0.0)
        {
            h = 0.0;
        }
    }
}
else
{
    b /= a;
    c /= a;
    double d =
= b * b - 4.0 * c;

    if (d < 0.0)
    {
        h = 0.0;
    }
    else
    {
        double sd = sqrt(d);
        double h1 =
= 0.5 * (-b - sd);
        if (h1 > 0.0)
        {
            h = h1;
        }
        else
        {
            double h2 =
= 0.5 * (-b + sd);
            if (h2 > 0.0)
            {
                h = h2;
            }
            else
            {
                h = 0.0;
            }
        }
    }
}
```

```
// Код в предикатной форме для векторизации.
h = 0.0;
p1 = (a == 0.0);
p2 = p1 && (b != 0.0);
h = std::max(-c / b, 0.0);            [?  p2]
b /= a;                               [? !p1]
c /= a;                               [? !p1]
d = b * b - 4.0 * c;                  [? !p1]
p4 = !p1 && (d >= 0.0);
sd = sqrt(d);                         [?  p4]
h1 = 0.5 * (-b - sd);                 [?  p4]
h = (h1 > 0.0)
    ? h1
    : std::max(0.5 * (-b + sd), 0.0); [?  p4]


// Векторный код с использованием
функций-интринсиков.
h = zero;

__mmask16 p1 = _mm512_cmpeq_ps_mask(a, zero);
__mmask16 p2 = _mm512_mask_cmpneq_ps_mask(p1,
b, zero);

h = _mm512_maskz_max_ps(p2,
        _mm512_maskz_div_ps(p2,
_mm512_sub_ps(zero, c), b),
        zero);

__mmask16 np1 = _mm512_knot(p1);

b = _mm512_mask_div_ps(b, np1, b, a);
c = _mm512_mask_div_ps(c, np1, c, a);

__m512 d = _mm512_sub_ps(_mm512_mul_ps(b, b),
                _mm512_mul_ps(four, c));

__mmask16 p4 = _mm512_mask_cmpge_ps_mask(np1,
d, zero);

__m512 sd = _mm512_mask_sqrt_ps(sd, p4, d);

__m512 h1 = _mm512_mul_ps(half,
                _mm512_sub_ps(zero,
_mm512_add_ps(b, sd)));

h = _mm512_mask_mov_ps(h, p4,
        _mm512_mask_blend_ps(
            _mm512_cmpgt_ps_mask(h1, zero),
            _mm512_max_ps(
                _mm512_mul_ps(half,
_mm512_sub_ps(sd, b)),
                 zero),
            h1));
```

Рис. 3. Пример векторизации плоского цикла

Fig. 3. Flat loop vectorization example

форму с последующей заменой скалярных инструкций на векторные аналоги является достаточно трудоемкой задачей, при выполнении которой велика вероятность внесения ошибки. Для упрощения описанного процесса предлагается инструментарий, реализация которого ведется в открытом репозитории (https://github.com/r-aax/flatvec).

Репозиторий flatvec состоит из отдельных тестов по векторизации программного контекста плоских циклов (http://www.swsys.ru/uploaded/image/2023-4/16.jpg). Имя теста имеет структуру case_<имя>, в качестве параметров функция принимает количество тестируемых векторов, а также повторений теста. Внутри теста выполняется инициализация мас-

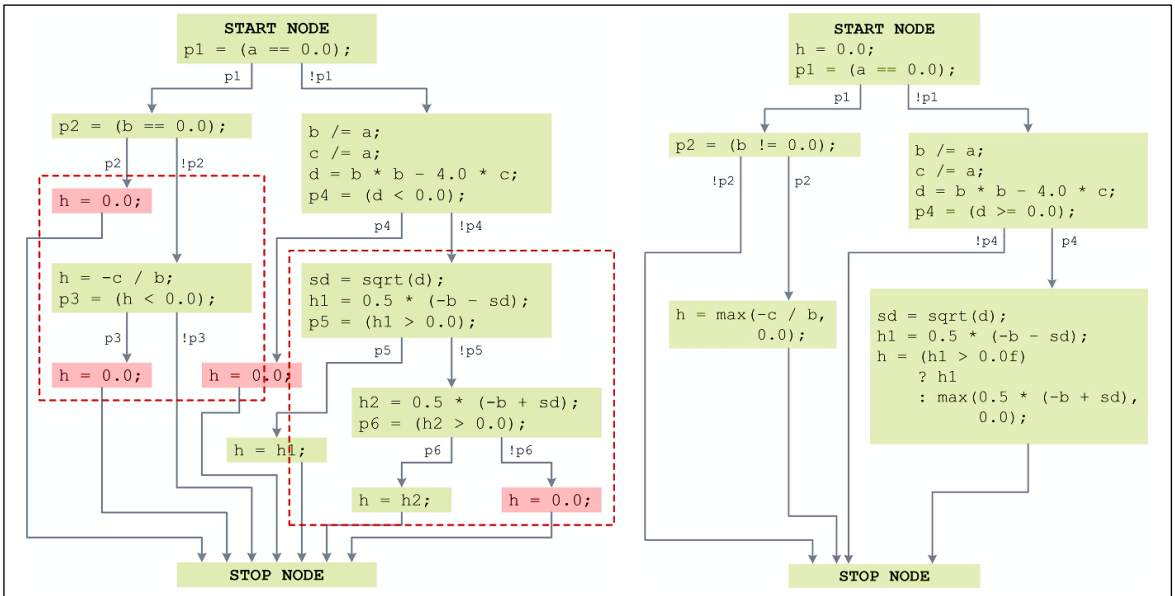

Рис. 4. Граф потока управления плоского цикла

Fig. 4. Control flow graph of flat loop

сивов входных данных, содержащих требуемое количество проверяемых векторов. Инициализация массивов входных данных может выполняться с помощью генерации псевдослучайных чисел или загрузки данных из текстовых файлов. После этого массивы данных обрабатываются с помощью скалярной версии цикла (scase_<имя>) и векторной версии того же цикла (vcase_<имя>). После выполнения вычислений скалярной и векторной версиями цикла выполняется проверка на совпадение результатов вычислений. В случае совпадения векторизация теста считается верной. Функция scase_<имя>_1 содержит скалярную реализацию одной итерации исходного цикла, в данной функции осуществляется работа с атомарными типами данных. Функция vcase_<имя>_1 содержит векторную реализацию *w* итераций исходного цикла, в данной функции работа ведется с векторными регистрами (для примера, работающего с входными данными типа float, значение *w* равно 16, так как в одном векторном 512-битном регистре помещаются 16 значений типа float).

Все помещенные в репозиторий тесты могут быть скомпилированы в двух режимах. Первый – режим исполнения и замера ускорения от векторизации. В данном режиме используемые функции-интринсики раскрываются в инструкции из набора AVX-512, на выходе получается исполняемый векторизованный код. Второй – режим эмуляции функций-интринсиков для отладки и мониторинга. В этом режиме векторные регистры и маски представлены классами fv::ZMM и fv::Mask, а функции-интринсики полностью эмулируются. В режиме эмуляции доступны мониторинг содержимого векторных регистров и масок, а также сбор информации по плотности масок и эффективности векторизации.

Сравним результаты работы некоторых тестов в двух различных режимах сборки. Результат работы в режиме исполнения:

```
guessp    : OK
Global stat:
  vector opers              : 0
  scalar opers              : 0
  mask opers                : 0
  theoretical acceleration  : -nan
  mean masks density        : -nan
  real time acceleration    : 9.354823
---------------------------
prefun    : OK
Global stat:
  vector opers              : 0
  scalar opers              : 0
  mask opers                : 0
  theoretical acceleration  : -nan
  mean masks density        : -nan
  real time acceleration    : 11.446910
---------------------------
sample    : OK
Global stat:
  vector opers              : 0
  scalar opers              : 0
  mask opers                : 0
  theoretical acceleration  : -nan
  mean masks density        : -nan
     real time acceleration   : 3.022369
```

Можно отметить, что в этом режиме недоступна никакая статистическая информация, кроме непосредственного измеренного ускорения от векторизации (real time acceleration).

Представим результат работы в режиме эмуляции:

```
guessp    : OK
Global stat:
  vector opers             : 610
  scalar opers             : 8285
  mask opers               : 60
  theoretical acceleration : 13.581967
  mean masks density       : 0.848873
  real time acceleration   : 0.058976
-------------------------
prefun    : OK
Global stat:
  vector opers             : 260
  scalar opers             : 3200
  mask opers               : 30
  theoretical acceleration : 12.307692
  mean masks density       : 0.769231
  real time acceleration   : 0.013309
-------------------------
sample    : OK
Global stat:
  vector opers             : 590
  scalar opers             : 7615
  mask opers               : 50
  theoretical acceleration : 12.906780
  mean masks density       : 0.806674
     real time acceleration   : 0.01155
```

В этом режиме, наоборот, доступна вся информация, которая собирается в процессе эмуляции, но время фактического ускорения векторизации не имеет смысла, поскольку эмуляция векторных инструкций заведомо выполняется значительно дольше исходного скалярного кода. В выводимой статистике можно отметить, что у всех тестов наблюдается достаточно высокое теоретическое ускорение от векторизации (после theoretical acceleration в режиме эмуляции), тогда как реальное изменяется в широких пределах (поле real time acceleration в режиме исполнения). Это связано с тем, что теоретическое ускорение рассчитывается через среднюю плотность масок в векторных инструкциях. Однако средняя плотность масок при векторизации кода может быть завышена. Для примера можно рассмотреть плоский цикл с телом c[i] = cnd ? (a[i] + b[i]) : (a[i] – b[i]). Теоретическое ускорение векторизованного кода, записанного в виде _mm512_mask_blend_ps(cnd, _mm512_add_ps(a, b), _mm512_sub_ps(a, b)), будет максимальным, так как все векторные инструкции обрабатывают все элементы своих аргументов. Но из полученных результатов инструкций add и sub для получения конечного результата используется только половина (вторая половина будет проигнорирована при выполнении инструкции blend). Таким образом, для получения более адекватной оценки теоретического ускорения необходимо проставлять маски во всех векторных инструкциях, хотя на реальную производительность это никак не влияет.

Рассмотренный подход к векторизации плоских циклов с помощью функций-интринсиков для инструкций AVX-512 был опробован на ряде реальных задач. В статье [19] приведено описание векторизации для матричных операций, а также для гнезд циклов с нерегулярным числом итераций на примере сортировки Шелла. В работе [20] в качестве объекта векторизации рассматривается программный контекст точного римановского решателя, в котором содержатся гнезда вложенных циклов, вызовы функций и сложное управление. Использование инструкций AVX-512 позволило добиться ускорения на данном программном коде примерно в 7 раз.

Заключение

Набор векторных инструкций AVX-512 оказывает существенное влияние на производительность суперкомпьютерных приложений. Наличие специальных векторных операций с выборочной обработкой отдельных элементов позволяет выполнять векторизацию достаточно сложного программного контекста. В работе рассмотрена специальная конструкция, называемая плоским циклом. На плоский цикл накладывается ряд ограничений, касающихся его структуры и порядка обращений в память. При выполнении этих ограничений плоский цикл может быть успешно векторизован с помощью специальных функций-интринсиков, объявленных в заголовочном файле immintrin.h. На первый взгляд может показаться, что ограничения, накладываемые на плоские циклы, слишком жесткие, однако значительная часть программного кода в расчетных приложениях может быть представлена в виде композиции плоских циклов без особых усилий. Для облегчения процесса векторизации плоского цикла путем записи программного кода в предикатном виде с последующей заменой всех операций на векторные аналоги был разработан инструмент flatvec, в котором поддержана возможность сборки тестов в двух режимах. Первый режим предназначен для по-

лучения результирующего кода для целевого микропроцессора, тогда как во втором режиме осуществляется эмуляция всех векторных инструкций, что позволяет выполнить отладку и мониторинг создаваемого векторного кода. Предложенный подход был использован для векторизации кода в ряде различных расчетных приложений для целевых микропроцессоров Intel Xeon Phi Knights Landing и Intel Xeon Cascade Lake. На сегодня существует проблема завышения показателя теоретического ускорения от векторизации в режиме эмуляции, что может приводить к излишне оптимистичным ожиданиям ускорения в реальных вычислениях. В дальнейшем планируется справиться с этой проблемой путем учета тех результатов векторных инструкций, которые фактически не участвуют в дальнейших вычислениях. Также одной из задач, которые необходимо решить для развития инструмента, является автоматическое выделение маловероятных ветвей исполнения из тела плоского цикла для выноса их из векторной версии путем проверки масок на пустоту.

Список литературы

1. Kusswurm D. Modern parallel programming with C++ and Assembly Language. X86 SIMD development using AVX, AVX2, and AVX-512. CA, Apress Berkeley Publ., 2022, 633 p. doi: 10.1007/978-1-4842-7918-2.

2. Blacher M., Giesen J., Sanders P., Wassenberg J. Vectorized and performance-portable Quicksort. ArXiv, 2022, art. 2205.05982, pp. 1–21. URL: https://arxiv.org/abs/2205.05982 (дата обращения: 14.06.2023).

3. Long S., Fan X., Chao L., Yi L., Fan S., Guo X.-W., Yang C. VecDualSPHysics: A vectorized implementation of Smoothed Particle Hydrodynamics method for simulating fluid flows on multi-core processors. J. of Computational Phys., 2022, vol. 463, art. 111234. doi: 10.1016/j.jcp.2022.111234.

4. Ponte-Fernández C., González-Domínguez J., Martín M.J. A SIMD algorithm for the detection of epistatic interactions of any order. Future Generation Comput. Sys., 2022, vol. 132, pp. 108–123. doi: 10.1016/j.future.2022.02.009.

5. Quislant R., Fernandez I. Time series analysis acceleration with advanced vectorization extensions. The J. of Supercomputing, 2023, vol. 79, no. 9, pp. 10178–10207. doi: 10.1007/s11227-023-05060-2.

6. Buhrow B., Gilbert B., Haider C. Parallel modular multiplication using 512-bit advanced vector instructions. J. of Cryptographic Engineering, 2022, vol. 12, pp. 95–105. doi: 10.1007/s13389-021-00256-9.

7. Choi H., Seo S.C. Efficient parallel implementations of PIPO Block Cipher on CPU and GPU. IEEE Access, 2022, vol. 10, pp. 85995–86007. doi: 10.1109/ACCESS.2022.3198707.

8. Cheng H., Fotiadis G., Großschädl J., Peter Y.A. Ryan. Highly vectorized SIKE for AVX-512. IACR Transactions on Cryptographic Hardware and Embedded Sys., 2022, no. 2, pp. 41–68. doi: 10.46586/tches.v2022.i2.41-68.

9. Sansone G., Cococcioni M. Experiments on speeding up the recursive fast Fourier transform by using AVX-512 SIMD instructions. Proc. ApplePies. LNEE, 2023, vol. 1036, pp. 255–263. doi: 10.1007/978-3-031-30333-3_34.

10. Edamatsu T., Takahashi D. Fast multiple-precision integer division using Intel AVX-512. IEEE Transactions on Emerging Topics in Computing, 2023, vol. 11, no. 1, pp. 224–236. doi: 10.1109/TETC.2022.3196147.

11. Медакин П.О., Никулин Р.Н., Авдеюк О.А., Королева И.Ю., Павлова Е.С., Лемешкина И.Г. Векторизация и распараллеливание метода «частица-частица» // Инженерный вестник Дона. 2021. № 1. URL: http://ivdon.ru/ru/magazine/archive/n1y2021/6800 (дата обращения: 30.06.2023).

12. Tayeb H., Paillat L., Bramas B. Autovesk: Automatic vectorization of unstructured static kernels by graph transformations. ArXiv, 2023, art. 2301.01018, pp. 1–21. URL: https://arxiv.org/abs/2301.01018 (дата обращения: 30.06.2023).

13. Laukemann J., Hammer J., Hager G., Wellein G. Automatic throughput and critical path analysis of x86 and ARM assembly kernels. Proc. IEEE/ACM PMBS, 2019, pp. 1–6. doi: 10.1109/PMBS49563.2019.00006.

14. Shabanov B.M., Rybakov A.A., Shumilin S.S. Vectorization of high-performance scientific calculations using AVX-512 instruction set. Lobachevskii J. of Math., 2019, vol. 40, no. 5, pp. 580–598. doi: 10.1134/S1995080219050196.

15. Intel 64 and IA-32 architectures software developer's manual. Combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Intel Corporation, 2017, no. 325462-079US. URL: https://archive.org/details/Intel_64_and_IA-32_Architectures_Software_Developer_s_Manual_2017_07 (дата обращения: 30.06.2023).

16. Рыбаков А.А., Шумилин С.С. Исследование эффективности векторизации гнезд циклов с нерегулярным числом итераций // Программные системы: Теория и алгоритмы. 2019. Т. 10. № 4. С. 77–96. doi: 10.25209/2079-3316-2019-10-4-77-96.

17. Carminati A., Starke R.A., de Oliveira R.S. Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. Applied Computing and Informatics, 2017, vol. 13, no. 2, pp. 184–193. doi: 10.1016/j.aci.2017.03.002.

18. Sahin V.H. Turna: A control flow graph reconstruction tool for RISC-V architecture. Computing, 2023, vol. 105, pp. 1821–1825. doi: 10.1007/s00607-023-01172-y.

19. Shabanov B.M., Rybakov A.A., Shumilin S.S. Vectorization of high-performance scientific calculations using AVX-512 instruction set. Lobachevskii J. of Math., 2019, vol. 40, no. 5, pp. 580–598. doi: 10.1134/S1995080219050196.

20. Рыбаков А.А., Шумилин С.С. Векторизация римановского решателя с использованием набора инструкций AVX-512 // Программные системы: Теория и алгоритмы. 2019. Т. 10. № 3. С. 59–80. doi: 10.25209/2079-3316-2019-10-3-59-80.

Software & Systems doi: 10.15827/0236-235X.142.561-572 2023, vol. 36, no. 4, pp. 561–572

Tools for vectorizing a flat loop body using AVX-512 vector instructions

Alexey A. Rybakov
Anthony N. Shvindt

For citation
Rybakov, A.A., Shvindt, A.N. (2023) 'Tools for vectorizing a flat loop body using AVX-512 vector instructions', *Software & Systems*, 36(4), pp. 561–572 (in Russ.). doi: 10.15827/0236-235X.142.561-572
Article info
Received: 23.08.2023 After revision: 01.09.2023 Accepted: 04.09.2023

Abstract. The paper studies approaches to reducing program execution time using instructions from AVX-512 set to increase program code vectorization efficiency. Vectorization is a basic low-level optimization that can speed up program execution by several times; AVX-512 instruction set has a number of unique features that allow applying vectorization in complex program contexts. The paper investigates a special type program context – a flat loop, which can be vectorized using AVX-512 instructions with almost arbitrary body complexity if meeting a number of requirements. However, quite often an optimizing compiler fails to automatically vectorize flat loops due to complex control, function calls, nested loops, and other constructs. This leads to manual vectorization using special intrinsic functions. Vectorization efficiency directly depends on the density of vector operation masks, which turn out to be sparse with a highly branched control inside a flat loop body. The paper proposes a toolkit that allows simultaneous creating a scalar and vector version of a flat loop and analyzing performed vectorization effectiveness. At the same time, there are two modes of assembling the code under test: a scalar version with emulation and monitoring of vector instructions and a vector version for execution on a target machine. On the one hand, this allows controlling code transformations to be correct; on the other hand, this allows tracking program execution paths with a low probability for their localization or removal from a loop body. The created tools enabled identifying hot flat loops of some real applications and increasing their vectorization efficiency.
Keywords: vectorization, flat loop, AVX-512, low-level optimization, computer applications
Acknowledgements. The work was carried out at the JSC RAS in terms of FNEF-2022-0016 state assignment. The research used MVS-10P supercomputer

References

1. Kusswurm, D. (2022) *Modern Parallel Programming with C++ and Assembly Language. X86 SIMD Development Using AVX, AVX2, and AVX-512*. CA, Apress Berkeley Publ., 633 p. doi: 10.1007/978-1-4842-7918-2.
2. Blacher, M., Giesen, J., Sanders, P., Wassenberg, J. (2022) 'Vectorized and performance-portable Quicksort', *ArXiv*, art. 2205.05982, pp. 1–21, available at: https://arxiv.org/abs/2205.05982 (accessed June 14, 2023).
3. Long, S., Fan, X., Chao, L., Yi, L., Fan, S., Guo, X.-W., Yang, C. (2022) 'VecDualSPHysics: A vectorized implementation of Smoothed Particle Hydrodynamics method for simulating fluid flows on multi-core processors', *J. of Computational Phys.*, 463, art. 111234. doi: 10.1016/j.jcp.2022.111234.
4. Ponte-Fernández, C., González-Domínguez, J., Martín, M.J. (2022) 'A SIMD algorithm for the detection of epistatic interactions of any order', *Future Generation Comput. Sys.*, 132, pp. 108–123. doi: 10.1016/j.future.2022.02.009.
5. Quislant, R., Fernandez, I. (2023) 'Time series analysis acceleration with advanced vectorization extensions', *The J. of Supercomputing*, 79(9), pp. 10178–10207. doi: 10.1007/s11227-023-05060-2.
6. Buhrow, B., Gilbert, B., Haider, C. (2022) 'Parallel modular multiplication using 512-bit advanced vector instructions', *J. of Cryptographic Engineering*, 12, pp. 95–105. doi: 10.1007/s13389-021-00256-9.
7. Choi, H., Seo, S.C. (2022) 'Efficient parallel implementations of PIPO Block Cipher on CPU and GPU', *IEEE Access*, 10, pp. 85995–86007. doi: 10.1109/ACCESS.2022.3198707.
8. Cheng, H., Fotiadis, G., Großschädl, J., Peter Y.A. Ryan. (2022) 'Highly vectorized SIKE for AVX-512', *IACR Transactions on Cryptographic Hardware and Embedded Sys.*, (2), pp. 41–68. doi: 10.46586/tches.v2022.i2.41-68.
9. Sansone, G., Cococcioni, M. (2023) 'Experiments on speeding up the recursive fast Fourier transform by using AVX-512 SIMD instructions', *Proc. ApplePies. LNEE*, 1036, pp. 255–263. doi: 10.1007/978-3-031-30333-3_34.
10. Edamatsu, T., Takahashi, D. (2023) 'Fast multiple-precision integer division using Intel AVX-512', *IEEE Transactions on Emerging Topics in Computing*, 11(1), pp. 224–236. doi: 10.1109/TETC.2022.3196147.
11. Medakin, P.O., Nikulin, R.N., Avdeyuk, O.A., Koroleva, I.Yu., Pavlova, E.S., Lemeshkina, I.G. (2021) 'Vectorization and parallelization of the particle-particle method', *Engineering J. of Don*, (1), available at: http://ivdon.ru/ru/magazine/archive/n1y2021/6800 (accessed June 30, 2023) (in Russ.).
12. Tayeb, H., Paillat, L., Bramas, B. (2023) 'Autovesk: Automatic vectorization of unstructured static kernels by graph transformations', *ArXiv*, art. 2301.01018, pp. 1–21, available at: https://arxiv.org/abs/2301.01018 (accessed June 30, 2023).
13. Laukemann, J., Hammer, J., Hager, G., Wellein, G. (2019) 'Automatic throughput and critical path analysis of x86 and ARM assembly kernels', *Proc. IEEE/ACM PMBS*, pp. 1–6. doi: 10.1109/PMBS49563.2019.00006.
14. Shabanov, B.M., Rybakov, A.A., Shumilin, S.S. (2019) 'Vectorization of high-performance scientific calculations using AVX-512 instruction set', *Lobachevskii J. of Math.*, 40(5), pp. 580–598. doi: 10.1134/S1995080219050196.
15. Intel 64 and IA-32 architectures software developer's manual. Combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. (2017) *Intel Corporation*, 325462-079US, available at: https://archive.org/details/Intel_64_and_IA-32_Architectures_Software_Developer_s_Manual_2017_07 (accessed June 30, 2023).

16. Rybakov, A.A., Shumilin, S.S. (2019) 'Study of the vectorization efficiency of loop nests with an irregular number of iterations', *Program Systems: Theory and Applications*, 10(4), pp. 77–96 (in Russ.). doi: 10.25209/2079-3316-2019-10-4-77-96.

17. Carminati, A., Starke, R.A., de Oliveira, R.S. (2017) 'Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software', *Applied Computing and Informatics*, 13(2), pp. 184–193. doi: 10.1016/j.aci.2017.03.002.

18. Sahin, V.H. (2023) 'Turna: A control flow graph reconstruction tool for RISC-V architecture', *Computing*, 105, pp. 1821–1825. doi: 10.1007/s00607-023-01172-y.

19. Shabanov, B.M., Rybakov, A.A., Shumilin, S.S. (2019) 'Vectorization of high-performance scientific calculations using AVX-512 instruction set', *Lobachevskii J. of Math.*, 40(5), pp. 580–598. doi: 10.1134/S1995080219050196.

20. Rybakov, A.A., Shumilin, S.S. (2019) 'Vectorization of the Riemann solver using the AVX-512 instruction set', *Program Systems: Theory and Applications*, 10(3), pp. 41–58. doi: 10.25209/2079-3316-2019-10-3-41-58.

Авторы

Рыбаков Алексей Анатольевич [1], к.ф.-м.н., ведущий научный сотрудник, rybakov@jscc.ru
Швиндт Антоний Николаевич [2], к.т.н., докторант, shvindt@ispras.ru

[1] МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН, г. Москва, 117218, Россия
[2] Институт системного программирования им. В.П. Иванникова РАН (ИСП РАН), г. Москва, 109004, Россия

Authors

Alexey A. Rybakov [1], Cand. of Sci. (Physics and Mathematics), Leading Researcher, rybakov@jscc.ru
Anthony N. Shvindt [2], Cand. of Sci. (Engineering), Doctoral Student, shvindt@ispras.ru

[1] JSCC RAS – branch of SRISA RAS, Moscow, 117218, Russian Federation
[2] Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS), Moscow, 109004, Russian Federation