Vol. 14, no. 1. 2018 *ISSN 2411-1473* *sitito.cs.msu.ru*

УДК 004.42
DOI: 10.25559/SITITO.14.201801.073-090

ВЕКТОРИЗАЦИЯ ОПЕРАЦИЙ НАД МАТРИЦАМИ МАЛОЙ РАЗМЕРНОСТИ ДЛЯ ПРОЦЕССОРА INTEL XEON PHI KNIGHTS LANDING

Л.А. Бендерский, С.А. Лещев, А.А. Рыбаков

Федеральный научный центр Научно-исследовательский институт системных исследований РАН, г. Москва, Россия

Аннотация

Статья посвящена вопросам векторизации вычислений для процессора Intel Xeon Phi Knights Landing (KNL). В качестве объектов для оптимизации рассматриваются операции над матрицами малой размерности. Такие операции являются распространенными в расчетных кодах из различных областей исследований, например при решении задач газовой динамики. KNL являются новейшими процессорами линейки Intel Xeon Phi, они содержат до 72 вычислительных ядер и позволяют выполнять приложения с использованием массивного параллелизма. Они обладают широкими возможностями для эффективного выполнения суперкомпьютерных вычислений, в частности поддерживают различные режимы работы с памятью и режимы кластеризации. Зачастую компилятор не справляется с задачей создания высокоэффективного параллельного векторизованного кода, что приводит к потерям производительности. Одним из резервов повышения производительности кода является осуществление ручной векторизации наиболее горячих участков кода, что приводит в итоге к ускорению работы всего приложения. При использовании процессоров KNL важным шагом оптимизации программы является задействование специальных 512-битных векторных инструкций, которые способны заметно ускорить исполняемый код. Использование 512-битных векторных инструкций позволяет обрабатывать векторы, содержащие по 16 элементов с плавающей точкой. Наличие специальных комбинированных FMA инструкций позволяет объединять операции покомпонентного умножения и сложения таких векторов. Для облегчения процесса ручной векторизации кода программы используются специальные функции-интринсики, которые являются обертками над инструкциями процессора. Применение векторизации операций над матрицами, выполненной с использованием функций-интринсиков, позволило добиться снижения времени выполнения данных операций от 23% до 70% по сравнению с версией,

Об авторах:

Бендерский Леонид Александрович, старший научный сотрудник, Межведомственный суперкомпьютерный центр Российской академии наук (филиал), Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук (119334, Россия, г. Москва, Ленинский проспект, д. 32а); ORCID: http://orcid.org/0000-0003-0529-3255, leosun.ben@gmail.com

Лещев Сергей Алексеевич, научный сотрудник, Межведомственный суперкомпьютерный центр Российской академии наук (филиал), Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук (119334, Россия, г. Москва, Ленинский проспект, д. 32а); ORCID: http://orcid.org/0000-0001-6728-1028, sergey.leshchev@jscc.ru

Рыбаков Алексей Анатольевич, кандидат физико-математических наук, ведущий научный сотрудник, Межведомственный суперкомпьютерный центр Российской академии наук (филиал), Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук (119334, Россия, г. Москва, Ленинский проспект, д. 32а); ORCID: http://orcid.org/0000-0002-9755-8830, rybakov@jscc.ru

построенной компилятором icc и с указанием максимального уровня оптимизации. Полученные результаты демонстрируют дополнительные резервы эффективности выполнения приложений, которые могут быть получены при помощи ручной оптимизации кода.

Ключевые слова

Операции над матрицами; векторизация; KNL; AVX-512; функции-интринсики.

VECTORIZATION OF OPERATIONS ON SMALL- DIMENSIONAL MATRICES FOR INTEL XEON PHI KNIGHTS LANDING PROCESSOR

Leonid A. Benderskiy, Sergey A. Leshchev, Alexey A. Rybakov

Scientific Research Institute for System Analysis of the Russian Academy of Sciences, SRISA, Moscow, Russia

Abstract

The article is devoted to the vectorization of calculations for Intel Xeon Phi Knights Landing (KNL) processor. Small-dimensional matrices are considered as objects for optimization. These operations are wide common in calculation codes in various scopes of research, for example, in calculational fluid dynamics. KNL is the latter Intel Xeon Phi processor, it contains up to 72 calculational cores and allows running applications using massive parallelism. They implement wide range of opportunities for effective performance of supercomputer calculations. In particular, they support different memory and cluster modes. In many cases the compiler isn't able to create high-performance parallel vectorized execution code. This leads to performance losses. One of the reserves of improving code performance is the manual vectorization of the hot blocks of the code. This leads to the entire application acceleration.

An important step in the program optimizing when using KNL processors is applying special 512-bit vector instructions that can significantly increase the speed of the execution code. Using of 512-bit vector instructions allows processing vectors consisting of 16 floating-point values. Special fused multiply-add instructions allow us to combine operations of componentwise multiplication and addition of these vectors. For simplification of the manual vectorization of the program code, special intrinsic functions are used. In fact these functions are just wrappers over the processor instructions. Vectorization of operations on matrices, performed with the intrinsic functions, made it possible to reduce the execution time of these operations in the range from 23% to 70% in comparison with the version compiled by the Intel compiler with the maximum level of optimization. The results received show additional hidden performance reserves of applications that can be obtained by manual optimization of the source code

Keywords

Matrix operations; vectorization; KNL; AVX-512; intrinsic functions.

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 [1, 2], и постепенно появляется потребность в использовании сеток, содержащих миллиарды ячеек. На сегодняшний день наиболее мощным суперкомпьютером является китайская машина Sunway TaihuLight, которая достигает производительности 93 PFLOPS на бенчмарке High Performance Linpack (HPL) [3]. В 2018 году ожидается появление сразу нескольких суперкомпьютеров с производительностью в диапазоне 100-200 PFLOPS, самым мощным из которых будет суперкомпьютер Summit от IBM, оснащенный процессором Power9 и графическими процессорами NVidia с архитектурой Volta. По некоторым оптимистическим прогнозам к 2024 году возможно появление первого экзафлопсного суперкомпьютера, способного выполнять 10^{18} операций с плавающей точкой в секунду [4]. 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, 6], по развитию технологий управления расчетными сетками и равномерного распределения вычислений на кластере [7, 8, 9], активно развиваются инструменты языков программирования, направленные на облегчение создания высокопроизводительного параллельного кода [10, 11].

Самым низкоуровневым направлением создания высокопроизводительного параллельного исполняемого кода является векторизация вычислений, позволяющая напрямую задействовать аппаратные возможности вычислителей [12, 13]. В данной работе рассматривается векторизация операций над матрицами малой размерности для процессоров Intel Xeon Phi Knights Landing (KNL). Данные операции часто встречаются в вычислительных задачах и могут занимать значительную часть времени выполнения расчета.

Обзор работ по теме исследования

Семейство процессоров Intel Xeon Phi x200 Knights Landing это второе поколение линейки Intel Xeon Phi и первое поколение вычислителей, выступающих в качестве самостоятельного процессора (первое поколение Knights Corner представляло собой сопроцессор [14]). Первые процессоры KNL были представлены в 2016 году. Каждый процессор содержит до 36 активных тайлов, в состав каждого из которых входят два ядра и L2 кэш размером 1 MB, являющийся общим для данных двух ядер. Каждое ядро содержит VPU (Vector Processing Unit), поддерживающий 512-битные векторные инструкции, и L1 кэш размером 64 KB, разделенный на равные по размеру кэш инструкций и кэш данных по 32 KB каждый. Каждое ядро поддерживает 4 потока, что дает суммарно 288 логических процессора на сокет [15]. Хотя частота каждого ядра KNL ниже, чем у серверных процессоров Intel Xeon, такое количество потоков исполнения и наличие 512-битных векторных инструкций обеспечивает внушительную пиковую производительность, превышающую 6 TFLOPS на операциях с одинарной точностью.

Появление такого мощного аппаратного средства открыло новые возможности для оптимизации программного обеспечения, использующегося в суперкомпьютерных расчетах. В работе [16] описаны подходы, позволившие с помощью векторизации для KNL ускорить вычислительные ядра программного кода LAMMPS вплоть до 12 раз по сравнению с невекторизованной версией, что привело к общему ускорению запусков решателя в 2-3 раза. Можно отметить успешное применение векторизации для ускорения операций с разреженными матрицами высокой размерности, давшее пятикратное ускорение на этих операциях [17]. Специальные возможности процессоров KNL, связанные с использованием маскированных векторных операций, используются для векторизации циклов даже с неизвестным количеством итераций и сложным управлением, что показано на примере векторизации кода построения множества Мандельброта [18]. В работе [19] описывается достижение 6-кратного ускорения, достигнутого с помощью применения низкоуровневой оптимизации расчетных кодов задач ядерной физики. В работе [20] освещается пример применения векторизации кода с

помощью вынесения маловероятной ветви исполнения из горячего внутреннего цикла.

Однако несмотря на высокие потенциальные возможности KNL исследования показывают, что ускорение, достигаемое по сравнению с запусками на процессорах Intel Xeon поколений Haswell и Broadwell, редко превышает 1.5-2 раза [21, 22]. Отчасти это связано с дисбалансом между пиковой пропускной способностью при работе с памятью и интенсивностью выполнения арифметических операций [23]. Ведутся активные исследования по использованию различных режимов работы с памятью и различных режимов кластеризации при запуске приложений на KNL [24].

Основным низкоуровневым средством повышения интенсивности выполнения арифметических операций для процессоров KNL является векторизация вычислений. Мы будем рассматривать три основные операции над матрицами малой размерности, которые встречаются в расчетных кодах, и скорость исполнения которых имеет важное значение: умножение матрицы на вектор, умножение матрицы на матрицу и нахождение обратной матрицы. Эффективность выполнения операций с матрицами является важной частью показателя параллелизма архитектуры и анализируется для демонстрации эффективности функционирования последней [25]. Рассматриваются операции для матриц размером 8x8 и 16x16, содержащих элементы типа float.

Теоретическая часть

Набор инструкций AVX-512 представляет собой 512-битное расширение 256-битных AVX инструкций из набора команд Intel x86, поддержанное в семействах микропроцессоров Intel Xeon Phi KNL и Intel Xeon Skylake. До появления набора инструкций AVX-512 сопроцессоры Intel Xeon Phi KNC также поддерживали 512-битные инструкции, которые схожи с AVX-512, однако не являются бинарно совместимыми с исполняемым кодом Intel Xeon.

Множество инструкций AVX-512 состоит из следующих подмножеств: AVX-512F (Foundation) - основной набор векторных инструкций с поддержкой маскирования, AVX-512PF (PreFetch) - инструкции предварительной подкачки данных из памяти, AVX-512ER (Exponential and Reciprocal) - команды для вычисления экспоненты и обратных значений, AVX-512CD (Conflict Detection) - инструкции для определения конфликтов, которые помогают эффективно применять векторизацию кода, а также наборы AVX-512BW и AVX-512DQ, поддержанные в Skylake. В следующих поколениях процессоров (Intel Xeon Phi Knights Mill, Intel Cannonlake, Intel Ice Lake) набор инструкций AVX-512 расширяется еще больше, в него входят команды для работы с 52-битными целочисленными значениями, специальные команды для работы с нейросетями и AES шифрованием, реализация арифметики полей Галуа, имплементация 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 могут использоваться для выборочного чтения из памяти и запись 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 операция является получение абсолютного значения, извлечение корня, округление, операции сдвигов и другие. Упакованные операции с двумя операндами zmm и одним результатом zmm отличаются только тем, что применяемая функция является

Vol. 14, no. 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

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. 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 арифметических действий, 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 того, чтобы увеличить вероятность того, что к моменту исполнения команды данные уже будут в кэше. 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.

Для упрощения векторизации исходного кода для компилятора icc разработаны специальные функции-интринсики, определенные в заголовочном файле immintrin.h. Они 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).

Отдельного внимания заслуживают интринсики, предназначенные для выполнения так называемых горизонтальных операций над элементами вектора. Например, сложение или перемножение всех элементов вектора, нахождение максимального или минимального элемента вектора. Для таких функций нет аналога в наборе команд, и интринсик, раскрывается в довольно длинную последовательность операций, состоящую из арифметики и перестановок.

После краткого обзора возможностей по использованию векторных инструкций для оптимизации кода можно перейти к главной цели данной статьи - поиску возможностей для ускорения операций с малоразмерными матрицами. Детально будут рассматриваться только варианты функций для работы с матрицами размера 8x8, так как реализация соответствующих функций для матриц 16x16 выполняется аналогично, и даже более просто (один 512-битный вектор содержит ровно 16 значений типа float). Рассмотрим первую операцию - умножение матрицы 8x8 на вектор. Реализация неоптимизированной версии может

Том 14, № 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

выглядеть следующим образом:

```
   void matvec8_orig(float * __restrict m,
float * __restrict v, float * __restrict r)
   {
       for (int i = 0; i < V8; i++)
       {
           float sum = 0.0;
           int ii = i * V8;

           for (int j = 0; j < V8; j++)
           {
               sum = sum + m[ii + j] * v[j];
           }
           r[i] = sum;
       }
   }
```

Рассмотрим некоторые моменты реализации. Матрица хранится в сплошной области памяти по строкам и передается в фунцию через указатель на начало этой области. Все параметры функции передаются с указанием __restrict для облегчения компилятору задачи по оптимизации. Умножение матрицы на вектор состоит в вычислении скалярного произведения каждой строки этой матрицы на вектор и составлении из результатов выходного вектора. Так как в данном случае размер строки матрицы равен 8 элементам типа float, то одной операцией в 512-битный регистр можно загрузить из памяти сразу 2 соседние строки матрицы. После чего нужно выполнить операцию упакованного умножения на регистр, содержащий две копии вектора, на который умножается матрица (выполнить запись двух копий вектора в zmm регистр можно с помощью операции gather). Сумма первых восьми элементов получившегося регистра и последних восьми элементов будут являться элементами выходного вектора r (см. рис. 1).

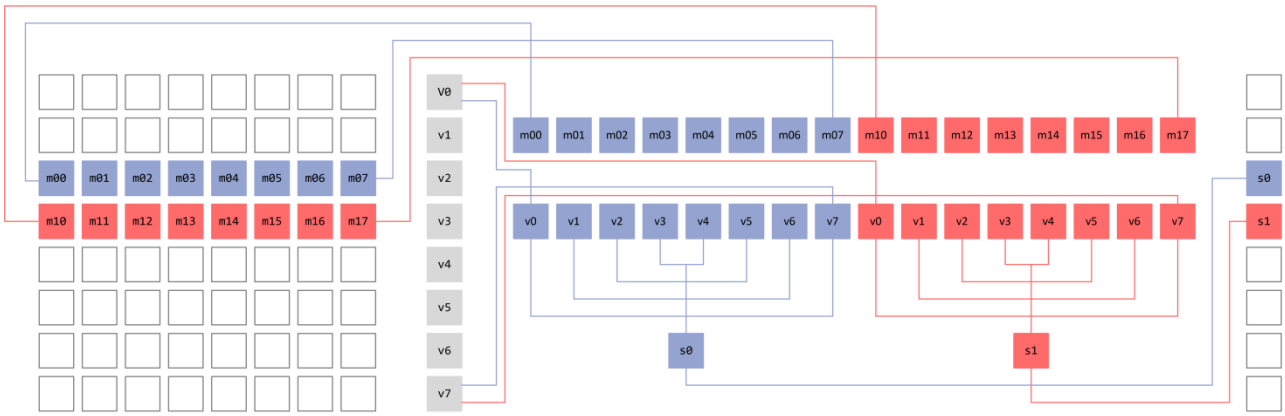

Рисунок 1. Схема вычисления результата в операции matvec8

Итак, при реализации операции умножения матрицы 8x8 на вектор мы должны загрузить всю матрицу в четыре zmm регистра. Затем выполнить четыре упакованные операции умножения этих регистров, на регистр, содержащий две копии вектора, на которые умножается матрица. После этого из каждого из получившихся четырех регистров мы должны получить сумму элементов каждой его половины, что в результате даст 8 искомых элементов выходного вектора. Получение суммы элементов половины zmm регистра представляет собой как раз ту горизонтальную операцию, реализация которой в помощью интринсика оказывается слишком дорогой. Как показали эксперименты, простое применение интринсика _mm512_mask_reduce_add_ps (и даже безмасочного _mm512_reduce_add_ps в случае умножения матрицы 16x16) не приводит к ускорению по сравнению с оригинальной версией функции, оптимизированной компилятором icc с использованием уровня оптимизации -O3. Прежде чем переходить к оптимизации данных горизонтальных операций рассмотрим второй интересующий нас пример, - перемножение двух матриц размера 8x8, - и убедимся, что в этом случае проявляется та же проблема. Как и в случае с первым примером, вначале приведем простую реализацию неоптимизированной версии перемножения двух матриц размера 8x8:

```
   void matmat8_orig(float * __restrict a,
float * __restrict b, float * __restrict r)
```

Vol. 14, no. 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

```
    {
        for (int i = 0; i < V8; i++)
        {
            int ii = i * V8;

            for (int j = 0; j < V8; j++)
            {
                float sum = 0.0;

                for (int k = 0; k < V8; k++)
                {
                    int kk = k * V8;

                    sum = sum + a[ii + k] *
b[kk + j];
                }

                r[ii + j] = sum;
            }
        }
    }
```

Логика перемножения двух матриц состоит в том, что результаты попарного скалярного произведения 8 строк матрицы a и 8 столбцов матрицы b формируют элементы результирующей матрицы. Как и в предыдущем примере, использование 512-битных команд загрузки данных из памяти позволяет за одну операцию загрузить две соседние строки матрицы a (операцией последовательного чтения) или два соседних столбца матрицы b (с помощью операции gather). После этого мы должны вычислить четыре скалярные произведения каждой половины загруженного из матрицы *a* вектора с каждой половиной загруженного из матрицы *b* вектора. Для этого выполняются две операции поэлементного перемножения двух загруженных векторов, в одном из которых старшая и младшая половины второго вектора переставлены местами, как показано на рис. 2.

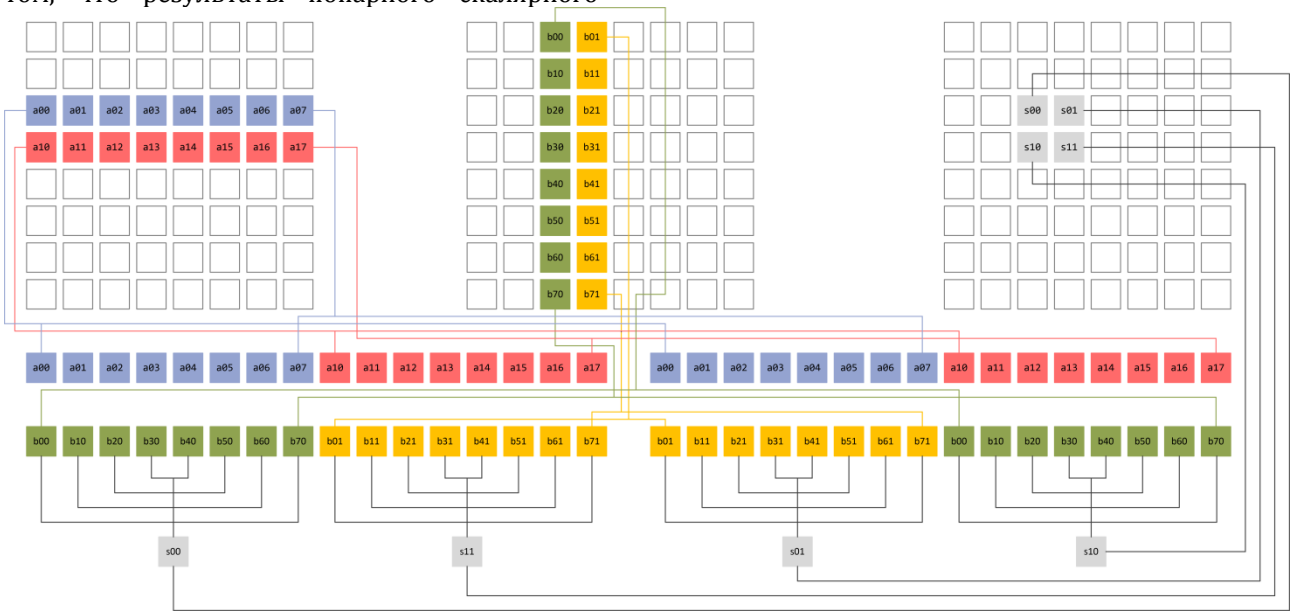

Рисунок 2. Схема вычисления результата в операции matmat8

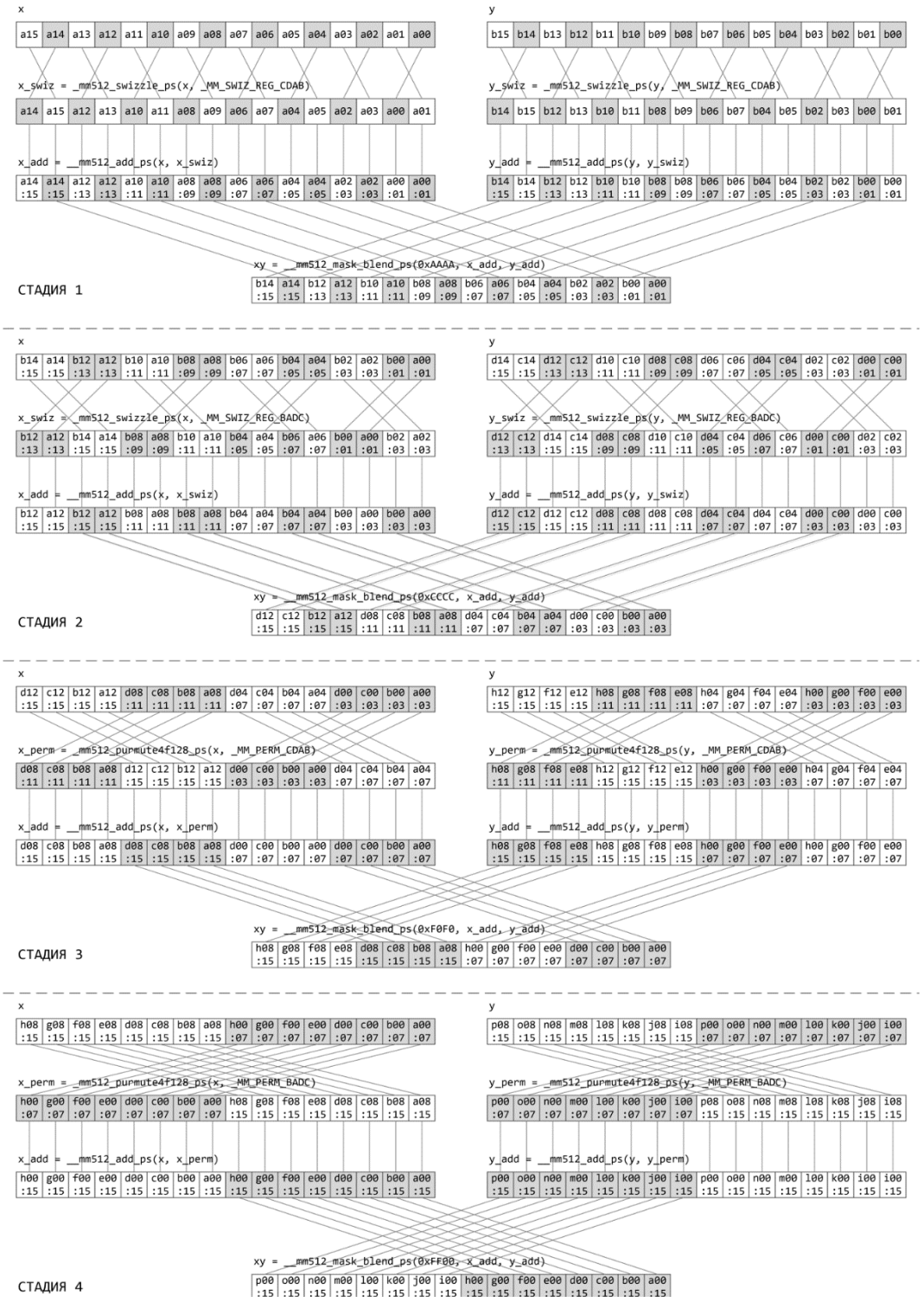

Рисунок 3. Схема суммирования элементов zmm вектора

После выполнения данных действий снова возникает потребность вычисления суммы 8

Vol. 14, no. 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

младших элементов данного вектора и 8 старших элементов этого же вектора. Таким образом, мы приходим к тому, что каждый элемент результирующей матрицы должен быть получен с помощью горизонтальной операции суммирования 8 младших или старших элементов некоторого zmm регистра.

Примеры реализаций функций matvec16_orig и matmat16_orig аналогичны, но там возникают задачи суммирования всех 16 элементов вектора. Отсюда возникает потребность объединения таких горизонтальных операций вместе. Рассмотрим задачу в общем виде: даны 16 zmm регистров (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p), каждый из которых содержит по 16 элементов типа float. Требуется посчитать суммы их элементов и записать в один регистр zmm. Набор команд AVX-512 и библиотека интринсиков содержат все необходимые возможности для эффективной реализации данного функционала. Схема вычислений состоит из четырех фаз, каждая из которых реализуется операциями перестановок с последующим сложением и слиянием векторов. На рис. 3 представлена схема суммирования элементов для пары векторов (a и b).

Набор команд AVX-512 не содержит операций горизонтального сложения элементов вектора. Таким образом, для сложения двух элементов одного и того же вектора требуется покомпонентно сложить этот вектор со своей копией, в которой элементы переставлены нужным образом. Это действие и выполняется на каждой обозначенной на рис. 3 фазе. Так как всего вектор содержит 16 элементов, то минимальное количество фаз, необходимых для суммирования его элементов равно $\log_2 16 = 4$. После первой фазы (после выполнения операции слияния blend) можно наблюдать вектор, содержащий суммы пар соседних элементов векторов a и b, после второй фазы вектор состоит уже из сумм четверок элементов векторов a, b, c и d, и т. д. На рис 4. представлен граф потока данных для осуществления суммирования всех элементов 16 векторов. При этом черными квадратами обозначены операции перестановки элементов (swizzle или purmute в терминах интринсиков), черными кругами обозначены операции сложения вектора со своей пермутированной копией, а белые круги обозначают операции слияния двух векторов по маске. Прямоугольниками с пунктирной границей очерчены фазы из рис. 3.

Можно заметить, что граф потока данных на рис. 4 состоит из схожих блоков операций, выполняющих одни и те же действия с разными входными регистрами: пермутация двух векторов, сложение пары векторов с пермутированной копией и слияние этих двух результатов по маске (2 swizzle/permute + 2 add + 1 blend). Причем на каждой фазе свои маски перестановок и слияний, но они постоянны для всех входных векторов. Для удобства можно определить макросы, реализующие обозначенные фазы. На вход макрос получает пару обрабатываемых векторов, а также параметры перестановки элементов и слияния результирующей пары. Пара макросов объясняется тем, что интринсик swizzle предназначен для перестановки элементов внутри 128-битных четвертей, а permute4f128 переставляет местами эти четверти (обе функции раскрываются в операцию perm).

```
   #define      SWIZ_2_ADD_2_BLEND_1(X,      Y,
SWIZ_TYPE, BLEND_MASK) \
       _mm512_mask_blend_ps(BLEND_MASK, \
                            _mm512_add_ps(X,
_mm512_swizzle_ps(X, SWIZ_TYPE)), \
                            _mm512_add_ps(Y,
_mm512_swizzle_ps(Y, SWIZ_TYPE)))

   #define      PERM_2_ADD_2_BLEND_1(X,      Y,
PERM_TYPE, BLEND_MASK) \
       _mm512_mask_blend_ps(BLEND_MASK, \
                            _mm512_add_ps(X,
_mm512_permute4f128_ps(X, PERM_TYPE)), \
                            _mm512_add_ps(Y,
_mm512_permute4f128_ps(Y, PERM_TYPE)))
```

Есть еще один похожий вариант выполнения одной фазы, реализующийся последовательностью операций 2 swizzle/permute + 2 blend + 1 add, но он оказался менее эффективным, поэтому не приводится.

Последний рассматриваемый в статье пример - нахождение обратной матрицы. Приведем краткое описание алгоритма Гаусса-Жордана для нахождения обратной матрицы и опишем пути оптимизации реализации соответствующей функции invmat8_orig.

Том 14, № 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

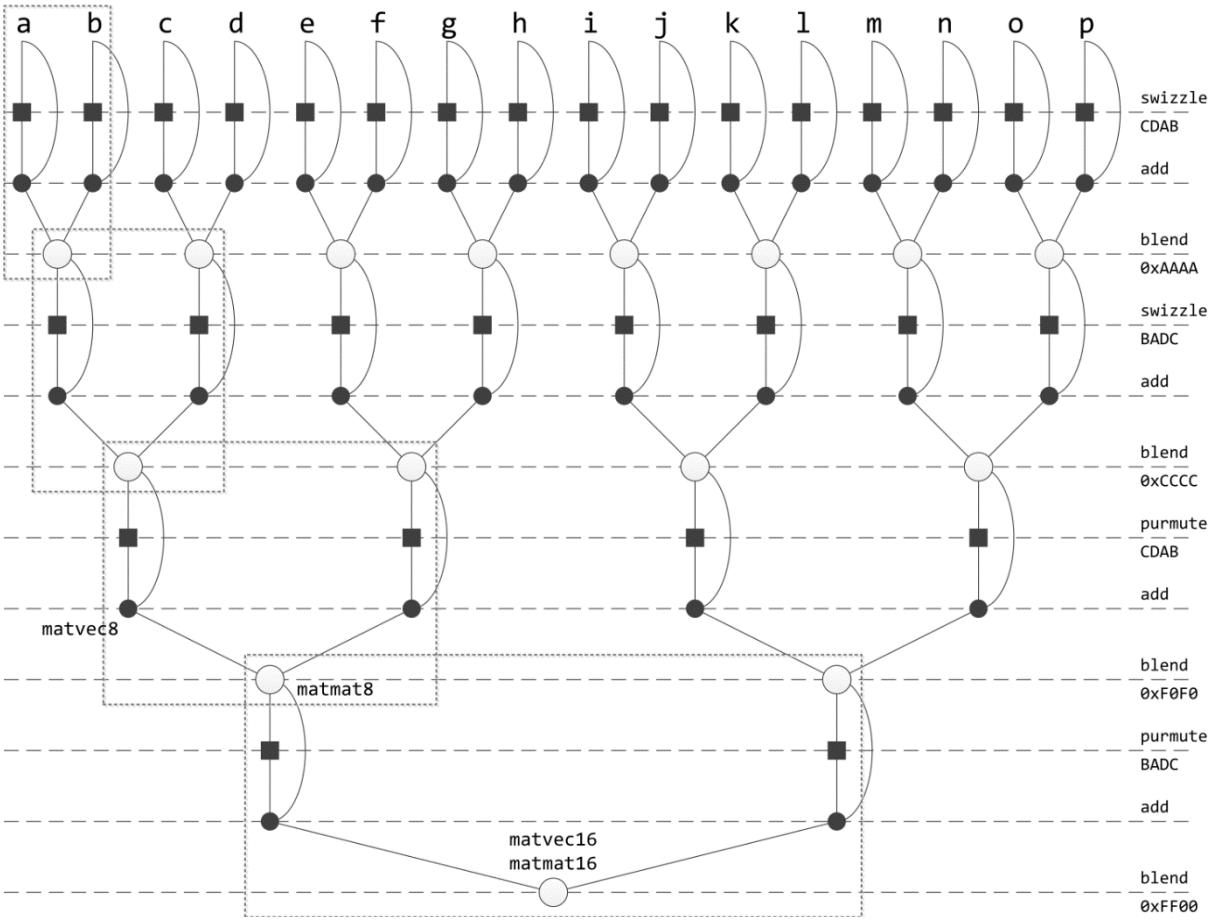

Рисунок 4. Схема суммирования элементов 16 zmm векторов

По данному алгоритму к исходной матрице m нужно приписать справа единичную матрицу e. Получим прямоугольную матрицу вида (a|e), размера 8x16. Далее над этой матрицей необходимо выполнить такие преобразования строк (перестановка строк, умножение строк на постоянное число, прибавление к одной строке другой строки, умноженной на постоянное число), чтобы исходная матрица, стоящая слева, трансформировалась в единичную. Тогда матрица, стоящая справа, из единичной трансформируется в искомую обратную матрицу. Для достижения этого эффекта выполняется следующая последовательность действий (цикл по i от 0 до 8):

Поиск ведущей строки с максимальным абсолютным значением i-го элемента (рис. 5. а). Если такое значение равно нулю, то матрица вырождена. Это единственное место алгоритма, которое плохо поддается векторизации, к тому же содержит аварийный выход. Данное действие занимает небольшую часть вычислений и не сказывается на производительности, поэтому ручная оптимизация в данном случае не применялась;

Перестановка ведущей строки и i-ой строки местами (рис. 5. б). После выполнения этого действия элемент m_{ii} является максимальным по абсолютному значению элементом i-го столбца. Действие успешно векторизуется с помощью операций пересылки zmm векторов;

Нормировка i-ой строки матрицы, то есть деление всех ее элементов на m_{ii} (рис. 5. в). После выполнения этого действия элемент m_{ii} равен единице. Реализуется одной операцией упакованного деления (или упакованного умножения на обратный элемент);

Vol. 14, no. 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

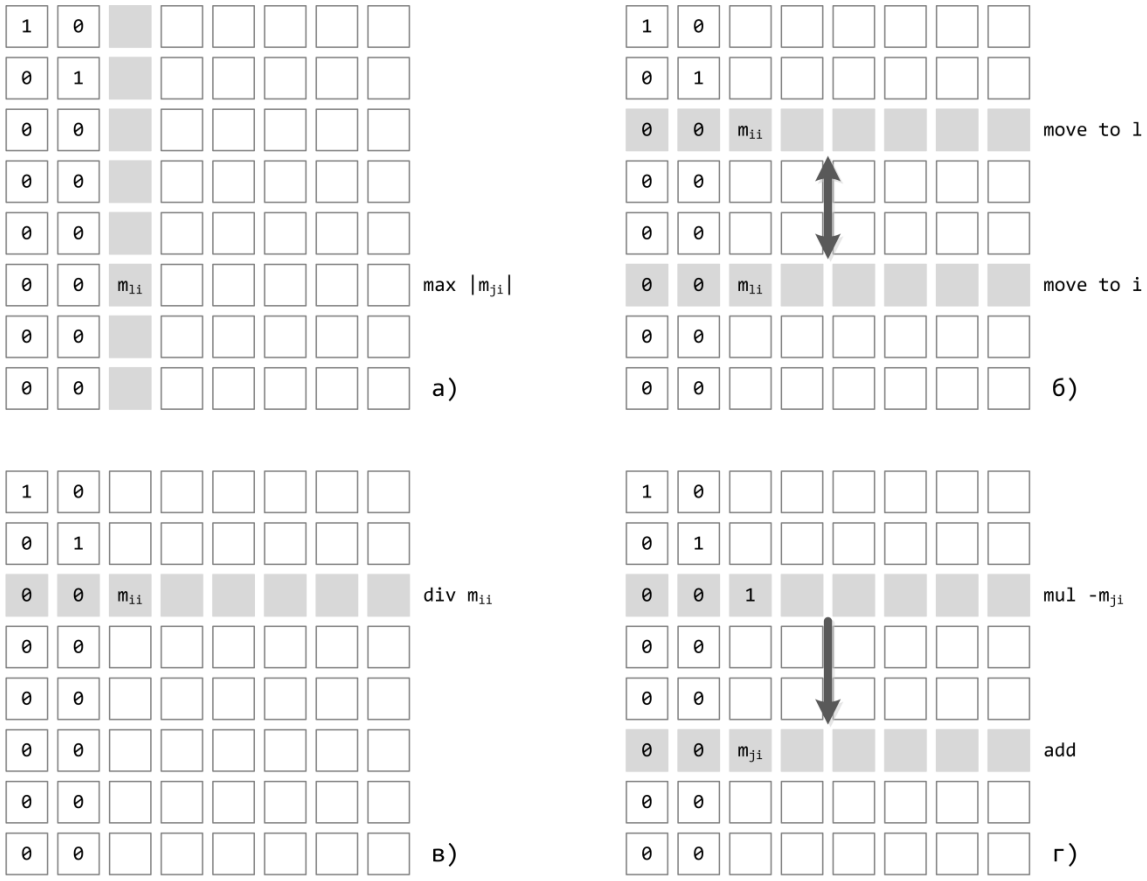

Рисунок 5. Базовые действия алгоритма Гаусса-Жордана нахождения обратной матрицы

С помощью i-ой строки обнуление всех i-ых элементов во всех остальных строках (рис. 5. г). Выполняется путем прибавления i-ой строки, умноженной на нужный коэффициент, к каждой строке матрицы. Для этого действия применяются упакованные FMA операции, которые существенно ускоряют код.

Однако нужно сделать одну оговорку. Описанные выше действия выполняются над строками матрицы размера 8x16, последовательно хранящейся в памяти. Но на вход функции подается матрица 8x8, также хранящаяся в памяти последовательно. Поэтому внутри функции возникают накладные действия, связанные с копированием элементов входной матрицы во временную матрицу 8x16 и обратным копированием из временной матрицы результата. Также требуется инициализация правой части временной матрицы элементами единичной матрицы. Все эти накладные действия реализуются с помощью операций gather/scatter. Заметим, что при реализации invmat16 накладные расходы не обязательны, так как нет смысла объединять две матрицы 16x16 в одну временную матрицу 16x32 (строка объединенной матрицы, содержащая 32 значения типа float, все равно не поместится в zmm регистр).

Реализационная часть

В данном разделе статьи приводятся оптимизированные фрагменты для описанных выше функций реализации операций с малоразмерными матрицами. В реализации функции matvec8_opt следует отметить загрузку двух копий вектора в zmm регистр с помощью операции gather. Существует возможность загрузки двух копий 256-битного участка памяти в zmm регистр, однако реализована она с помощью интринсика _mm512_extload_pd с указанием параметра _MM_BROADCAST_4X8. Так как формально в данном случае результатом

является регистр типа __m512d, то было решено оставить чтение через gather. Выполнение горизонтальных операций суммирования половинок четырех zmm регистров заканчивается на операции add третьей фазы, как показано на рис. 3.

```
   void matvec8_opt(float * __restrict m,
float * __restrict v, float * __restrict r)
   {
       .....................................
       // чтение двух копий вектора v
       vec                                 =
_mm512_i32gather_ps(_mm512_set_epi32(7,  6,  5,
4, 3, 2, 1, 0,

7, 6, 5, 4, 3, 2, 1, 0),
                                   v,
_MM_SCALE_4);
       .....................................
       // чтение и умножение на v первых двух
строк матрицы
       m0                                  =
_mm512_mul_ps(_mm512_load_ps(&m[0]), vec);
       .....................................
       // совмещенное горизонтальное сложение
       x0   =   SWIZ_2_ADD_2_BLEND_1(m0,   m2,
_MM_SWIZ_REG_CDAB, 0xAAAA);
       x2   =   SWIZ_2_ADD_2_BLEND_1(m4,   m6,
_MM_SWIZ_REG_CDAB, 0xAAAA);
       m0   =   SWIZ_2_ADD_2_BLEND_1(x0,   x2,
_MM_SWIZ_REG_BADC, 0xCCCC);
       x0          =          _mm512_add_ps(m0,
_mm512_permute4f128_ps(m0, _MM_PERM_CDAB));
       .....................................
       // сохранение результата
       _mm512_mask_i32scatter_ps(r, 0xF0F,

_mm512_set_epi32(0, 0, 0, 0, 7, 5, 3, 1,

0, 0, 0, 0, 6, 4, 2, 0),
                                   x0,
_MM_SCALE_4);
   }
```

Перемножение двух матриц изначально реализуется гнездом из трех вложенных циклов. Два внешних цикла относятся к проходу по строкам одной из перемножаемых матриц и столбцам другой (за одну итерацию обрабатывается по две строки и по два столбца соответственно). Внутренний цикл считает скалярное произведение соответствующей строки и столбца. Так как строки матрицы занимают в памяти последовательные участки, то чтение двух соседних строк выполняется обычной функцией _mm512_load_ps. Загрузку же двух соседних столбцов следует выполнять с использованием gather и заданными смещениями всех элементов. По этой причине в реализации была выполнена перестановка двух внешних циклов, чтобы чтение с помощью gather выполнялось во внешнем цикле, так как данная операция более медленная, чем чтение последовательного участка памяти. Далее выполняется полная раскрутка ставшего теперь уже внутренним цикла, проходящего по строкам матрицы a. Выполнение горизонтальных операций суммирования половинок 8 регистров zmm заканчивается на операции blend третьей фазы из рис. 3. Полученные 16 элементов результирующей матрицы записываются в память с использованием scatter с указанием смещений.

```
   void matmat8_opt(float * __restrict a,
float * __restrict b, float * __restrict r)
   {
       .....................................
       // индексы для чтения соседних столбцов
матрицы b
       ind = _mm512_set_epi32(7 * V8 + 1, 6 *
V8 + 1, 5 * V8 + 1, 4 * V8 + 1,
                              3 * V8 + 1, 2 *
V8 + 1,     V8 + 1,          1,
                                  7    *    V8,
6 * V8,     5 * V8,     4 * V8,
                                  3    *    V8,
2 * V8,          V8,          0);

       for (int j = 0; j < V8; j += 2)
       {
           ..................................
.....
           // два соседние стобца матрицы b в
прямом и обратном порядке
           bj = _mm512_i32gather_ps(ind, &b[j],
_MM_SCALE_4);
           bj2   =   _mm512_permute4f128_ps(bj,
_MM_PERM_BADC);
           ..................................
.....
           //     подготовка     данных     для
горизонтального сложения, остальные 3 блока
аналогичные
           a0 = _mm512_load_ps(&a[ii0]);
           m0 = _mm512_mul_ps(a0, bj);
           m1 = _mm512_mul_ps(a0, bj2);


           ..................................
.....
           //    совмещенное    горизонтальное
сложения
           x0  =  SWIZ_2_ADD_2_BLEND_1(m0,  m1,
_MM_SWIZ_REG_CDAB, 0xAAAA);
           x1  =  SWIZ_2_ADD_2_BLEND_1(m2,  m3,
_MM_SWIZ_REG_CDAB, 0xAAAA);
           x2  =  SWIZ_2_ADD_2_BLEND_1(m4,  m5,
_MM_SWIZ_REG_CDAB, 0xAAAA);
           x3  =  SWIZ_2_ADD_2_BLEND_1(m6,  m7,
_MM_SWIZ_REG_CDAB, 0xAAAA);
           m0  =  SWIZ_2_ADD_2_BLEND_1(x0,  x1,
_MM_SWIZ_REG_BADC, 0xCCCC);
           m1  =  SWIZ_2_ADD_2_BLEND_1(x2,  x3,
_MM_SWIZ_REG_BADC, 0xCCCC);
           x0  =  PERM_2_ADD_2_BLEND_1(m0,  m1,
_MM_PERM_CDAB, 0xF0F0);
```

Vol. 14, no. 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

```
            ..................................
.....
            // сохранение результата
            ind1 = _mm512_set_epi32(ii3 + V8 +
j, ii3 + V8 + j + 1,
                                    ii2 + V8 +
j, ii2 + V8 + j + 1,
                                    ii1 + V8 +
j, ii1 + V8 + j + 1,
                                    ii0 + V8 +
j, ii0 + V8 + j + 1,
                                     ii3 + j +
1,          ii3 + j,
                                     ii2 + j +
1,          ii2 + j,
                                     ii1 + j +
1,          ii1 + j,
                                     ii0 + j +
1,          ii0 + j),
            _mm512_i32scatter_ps(r,  ind1,  x0,
_MM_SCALE_4);
        }
    }
```

Реализация функции invmat8_opt начинается с формирования временной матрицы размера 8x16, левой частью которой становится входная матрица, а в правой считается результат. Входная матрица копируется во временную с помощью четырех пар операций load - scatter, одновременно обнуляется правая часть матрицы. После этого для завершения инициализации единичной матрицы в правой части временной матрицы в нее с помощью операции scatter записывается вектор, состоящий из единиц, с указанием расстояния между записями V16 + 1. После инициализации временной матрицы выполняется тело алгоритма Гаусса-Жордана нахождения обратной матрицы. Действие по нахождению ведущей строки не подвергалось векторизации. Прежде всего там присутствует горизонтальная операция, связанная с нахождением максимального по модулю элемента некоторого столбца временной матрицы. Кроме того, во время поиска ведущей строки возможен аварийный выход из функции, поэтому попытки оптимизации данного фрагмента нецелесообразны. Три остальные действия реализуются без каких бы то ни было особенностей с помощью применения пересылок и покомпонентных операций сложения, умножения и комбинированных FMA операций.

```
    int invmat8_opt(float * __restrict m, float
* __restrict r)
    {
        ....................................
        // подготовка временной матрицы 8x16
        __m512i ind = _mm512_set_epi32(V16 + 7,
V16 + 6, V16 + 5, V16 + 4,
                                       V16 + 3,
V16 + 2, V16 + 1,     V16,
                                             7,
6,       5,       4,
                                             3,
2,       1,       0);

        for (int i = 0; i < V8; i += 2)
        {
            _mm512_i32scatter_ps(&t[i  *  V16],
ind, _mm512_load_ps(&m[i * V8]), _MM_SCALE_4);
            _mm512_i32scatter_ps(&t[i  *  V16  +
V8], ind, _mm512_setzero_ps(), _MM_SCALE_4);
        }

        __m512i ind1 = _mm512_set_epi32(0, 0, 0,
0, 0, 0, 0, 0,
                                        7 * V16
+ 7, 6 * V16 + 6,
                                        5 * V16
+ 5, 4 * V16 + 4,
                                        3 * V16
+ 3, 2 * V16 + 2,
                                           V16
+ 1,           0);
        _mm512_mask_i32scatter_ps(&t[V8],  0xFF,
ind1, _mm512_set1_ps(1.0), _MM_SCALE_4);

        for (int i = 0; i < V8; i++)
        {
            // поиск ведущей строки (опущено)
            ....................................
.....
            // перемена местами двух строк
            if (lead_i != i)
            {
                int ll = lead_i * V16;

                vi = _mm512_load_ps(&t[ii]);
                vl = _mm512_load_ps(&t[ll]);
                _mm512_store_ps(&t[ll], vi);
                _mm512_store_ps(&t[ii], vl);
            }
            ....................................
.....
            // нормализация строки
            vd  =  _mm512_set1_ps(1.0  /  t[ii  +
i]);
            vi = _mm512_load_ps(&t[ii]);
            vi = _mm512_mul_ps(vi, vd);
            _mm512_store_ps(&t[ii], vi);
            ....................................
.....
            // обнуление i-ых элементов других
строк
            for (int j = 0; j < V8; j++)
            {
                int jj = j * V16;

                if (j != i)
                {
                    vd = _mm512_set1_ps(-t[jj +
i]);
                    vj = _mm512_load_ps(&t[jj]);
                    vi = _mm512_load_ps(&t[ii]);
                    vj = _mm512_fmadd_ps(vi, vd,
vj);
```

```
                    _mm512_store_ps(&t[jj], vj);
                }
            }
        }

        .....................................
        // копирования результата из временной
матрицы
        for (int i = 0; i < V8; i += 2)
        {
            vi = _mm512_i32gather_ps(ind, &t[i
* V16 + V8], _MM_SCALE_4);
            _mm512_store_ps(&r[i * V8], vi);
        }
    }
```

После завершения работы алгоритма Гаусса-Жордана результат из временной матрицы перемещается в область памяти результата с помощью пар операций gather - store.

Реализация функций по работе с матрицами 16x16 не приводится, так как данные функции имплементируются аналогично, только код получается более громоздкий. Выполнение горизонтальных операций сложения для matvec16 и matmat16 охватывает все четыре фазы из рис. 3, логика же вычислений более примитивна, так как не требуется объединения итераций при обходе матриц - одна строка матрицы в точности заполняет zmm регистр.

Экспериментальная часть

Реализованные в рамках данной работы оптимизированные с помощью инструкций AVX-512 функции для работы с малоразмерными матрицами были опробованы на процессоре Intel Xeon Phi KNL 7290. Данные процессоры входят в состав суперкомпьютера МВС-10П, находящегося в МСЦ РАН. Был выполнен анализ эффективности примененных оптимизаций. Для этого рассматривались два варианта исполняемого кода. В первом варианте функции были реализованы без использования интринсиков, компилировались с использованием компилятора icc с указанием -xmic-avx512 и уровнем оптимизации -O3. Во втором варианте использовались те же опции компиляции, однако функции были оптимизированы вручную, как это было описано в предыдущих разделах. В обоих вариантах был запрещен инлайн рассматриваемых функций в место вызова. Рассматривались функции умножения матрицы на вектор, перемножения двух матриц и нахождения обратной матрицы. Размеры матриц рассматривались в двух вариантах: 8x8 и 16x16. Всего было рассмотрено 6 функций: matvec8, matvec16, matmat8, matmat16, invmat8, invmat16.

На рис. 6 приведены данные о сокращении времени работы оптимизированных функций, реализующих операции над малоразмерными матрицами. За 100% принято время исполнения функций, реализованных без использования интринсиков. Темным цветом на гистограмме отмечено время исполнения вариантов функций, реализованных с явным использованием 512-битных векторных операций. Наибольшее ускорение продемонстрировали функции нахождения обратной матрицы, так как алгоритм нахождения обратной матрицы естественным образом формулируется в терминах работы со строками объединенной матрицы, что находит свое отражение в 512-битных векторных операциях. При этом функция invmat16 ускорилась больше, чем invmat8, потому что, в отличие от последней, она не содержит накладных действий, связанных с копированием входной матрицы во временную матрицу, а затем чтением результата из временной матрицы. Функции matvec и matmat демонстрируют умеренное ускорение. Причиной тому служит наличие горизонтальных операций сложения, которые порождают длинный критический путь исполнения внутри функции. Объединение горизонтальных операций позволяет сгладить этот эффект, однако для дальнейшего ускорения требуется реализация объединенных функций, выполняющих одновременно сразу несколько операций или применяющих одну операцию к более широкому набору данных (например, функция перемножение двух пар матриц).

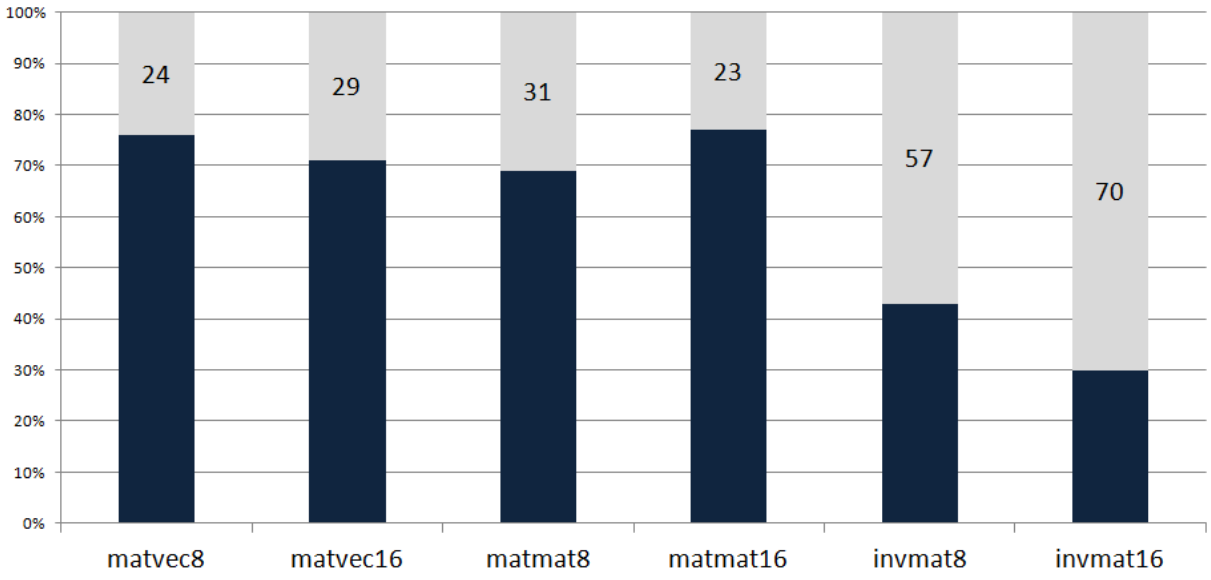

Рисунок 6. Сокращение времени работы оптимизированных версий операций над малоразмерными матрицами

Заключение

Проведенные исследования и экспериментальные расчеты показали, что применение ручной оптимизации исходного кода программы с помощью низкоуровневых конструкций является оправданным, так как компилятор icc не всегда способен добиться наилучшей производительности кода даже с применением максимального уровня оптимизации. Также компилятору не всегда удается скомпоновать код таким образом, чтобы применение векторных инструкций AVX-512 оказалось возможным и оправданным. Отчасти этому способствуют модификаторы, сигнализирующие об отсутствии зависимостей или о правильном выравнивании данных, но это не может полностью решить проблему. Отметим, что ручная оптимизация целесообразна только для самых горячих участков программы. Это могут быть часто используемые библиотечные вызовы (матричные операции, FFT и другие), ядра решателей и другие критичные по производительности части кода. Применение ручной оптимизации для функций работы с малоразмерными матрицами, рассматриваемых в данной работе, привело к снижению времени исполнения последних в диапазоне от 23% до 70%, хотя они по сути состоят из типичных шаблонов, с которыми компилятор должен успешно справляться. В данном свете видится, что другие горячие участки исполняемых кодов, предназначенных для суперкомпьютерных расчетов, обладающие менее шаблонной реализацией, таят в себе еще больший потенциал для оптимизации под процессоры, поддерживающие работу с широкими 512-битными векторами.

Благодарность

Работа выполнена при поддержке гранта Российского фонда фундаментальных исследований № 18-07-00638.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[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. DOI: https://doi.org/10.1007/978-3-319-58667-0_1

[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. DOI: https://doi.org/10.1007/978-3-319-47066-5_34

[3] *Fu H.H., Liao J.F., Yang J.Z.*, et al. The Sunway TaihuLight supercomputer: system and applications // Science China Information Sciences. 2016. Vol. 59, issue 7, id. 072001. DOI: https://doi.org/10.1007/s11432-016-5588-7

[4] *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. DOI: https://doi.org/10.1007/978-3-319-46079-6_5

[5] *Klenk B., Fröning 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. DOI: https://doi.org/10.1007/978-3-319-58667-0_12

[6] *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. DOI: https://doi.org/10.1007/978-3-319-58667-0_5

[7] *Рыбаков А.А.* Внутреннее представление и механизм межпроцессного обмена для блочно-структурированной сетки при выполнении расчетов на суперкомпьютере // Программные системы: теория и приложения. 2017. Т. 8, Вып. 1. С. 121-134. DOI: https://doi.org/10.25209/2079-3316-2017-8-1-121-134

[8] *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. DOI: https://doi.org/10.1007/978-3-319-58667-0_14

[9] *Бендерский Л.А., Любимов Д.А., Рыбаков А.А.* Анализ эффективности масштабирования при расчетах высокоскоростных турбулентных течений на суперкомпьютере RANS/ILES методов высокого разрешения // Труды НИИСИ РАН. 2017. Т. 7, № 4. С. 32-40. URL: https://elibrary.ru/item.asp?id=32294100 (дата обращения: 10.02.2018).

[10] *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. DOI: https://doi.org/10.1007/978-3-319-46079-6_2

[11] *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. Т. 7, Вып. 3. С. 99-107. DOI: https://doi.org/10.25209/2079-3316-2016-7-3-99-107

[12] *Bramas B.* Fast sorting algorithms using AVX-512 on Intel Knights Landing // arXiv: 1704.08579 [cs.MS]. URL: https://arxiv.org/abs/1704.08579 (дата обращения: 10.02.2018).

[13] *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. Т. 7, Вып. 2. С. 3-25. DOI: https://doi.org/10.25209/2079-3316-2016-7-2-3-25

[14] *Dorris J., Kurzak J., Luszczek P.* Task-Based Cholesky Decomposition on Knights Corner Using OpenMP // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS. 2016. Vol. 9945. Pp. 544–562. DOI: https://doi.org/10.1007/978-3-319-46079-6_37

[15] *Tobin J., Breuer A., Heinecke A.* et al. Accelerating Seismic Simulations Using the Intel Xeon Phi Knights Landing Processor // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS. 2017. Vol. 10266. Pp. 139–157. DOI: https://doi.org/10.1007/978-3-319-58667-0_8

[16] *McDoniel W., Höhnerbach M., Canales R.* et al. LAMMPS' PPPM Long-Range Solver for the Second Generation Xeon Phi // J. M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS. 2017. Vol. 10266. Pp. 61–78. DOI: https://doi.org/10.1007/978-3-319-58667-0_4

[17] *Malas T., Kurth T., Deslippe J.* Optimization of the Sparse Matrix-Vector Products of an IDR Krylov Iterative Solver in EMGeo for the Intel KNL Manycore Processor // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS. 2016. Vol. 9945. Pp. 378–389. DOI: https://doi.org/10.1007/978-3-319-46079-6_27

[18] *Krzikalla O., Wende F., Höhnerbach M.* Dynamic SIMD Vector Lane Scheduling // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS. 2016. Vol. 9945. Pp. 354–365. DOI: https://doi.org/10.1007/978-3-319-46079-6_25

[19] *Cook B., Maris P., Shao M.* High Performance Optimizations for Nuclear Physics Code MFDn on KNL // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS. 2016. Vol. 9945. Pp. 366–377. DOI: https://doi.org/10.1007/978-3-319-46079-6_26

[20] *Рыбаков А.А.* Оптимизация задачи об определении конфликтов с опасными зонами движения летательных аппаратов для выполнения на Intel Xeon Phi // Программные продукты и системы. 2017. Т. 30, № 3. С. 524-528. DOI: https://doi.org/10.15827/0236-235.X.030.3.524-528

[21] *Sengupta D., Wang Y. Sundaram N.* et al. High-Performance Incremental SVM Learning on Intel Xeon Phi Processors // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS. 2017. Vol. 10266. Pp. 120–138. DOI: https://doi.org/10.1007/978-3-319-58667-0_7

[22] *Kronbichler M., Kormann K., Pasichnyk I.* Fast Matrix-Free Discontinuous Galerkin Kernels on Modern Computer Architectures // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS. 2017. Vol. 10266. Pp. 237–255. DOI: https://doi.org/10.1007/978-3-319-58667-0_13

[23] *Doerfler D., Deslippe J., Williams S.* et al. Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS. 2016. Vol. 9945. Pp. 339–353. DOI: https://doi.org/10.1007/978-3-319-46079-6_24

[24] *Rosales C., Cazes J., Milfeld K.* A Comparative Study of Application Performance and Scalability on the Intel Knights Landing Processor // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS. 2016. Vol. 9945. Pp. 307–318. DOI: https://doi.org/10.1007/978-3-319-46079-6_22

[25] *Дикарев Н. И., Шабанов Б. М., Шмелев А. С.* Использование «сдвоенного» умножителя и сумматора в векторном процессоре с архитектурой управления потоком данных // Программные системы: теория и приложения. 2015. Т. 6, Вып. 4. С. 227-241. DOI: https://doi.org/10.25209/2079-3316-2015-6-4-227-241

Поступила 20.12.2017; принята к публикации 10.02.2018; опубликована онлайн 30.03.2018.

REFERENCES

[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, p. 3–21. DOI: https://doi.org/10.1007/978-3-319-58667-0_1

[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. p. 499-510. DOI: https://doi.org/10.1007/978-3-319-47066-5_34

[3] Fu H.H., Liao J.F., Yang J.Z., et al. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences*. 2016; 59(7), id. 072001. DOI: https://doi.org/10.1007/s11432-016-5588-7

[4] 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. 994, p. 56–68. DOI: https://doi.org/10.1007/978-3-319-46079-6_5

[5] Klenk B., Fröning H. An Overview of MPI Characteristics of Exascale Proxy Applications // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017. Vol. 10266, p. 217–236. DOI: https://doi.org/10.1007/978-3-319-58667-0_12

[6] 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, p. 79–96. DOI: https://doi.org/10.1007/978-3-319-58667-0_5

[7] Rybakov A. Inner representation and crossprocess exchange mechanism for block-structured grid for supercomputer calculations. *Program systems: Theory and applications*. 2017; 8:1(32):121–134. (In Russian) DOI: https://doi.org/10.25209/2079-3316-2017-8-1-121-134

[8] 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, p. 256–274. DOI: https://doi.org/10.1007/978-3-319-58667-0_14

[9] Benderskij L.A., Ljubimov D.A., Rybakov A.A. Scaling of fluid dynamic calculations using the RANS/ILES method on supercomputer. *Trudy NIISI RAN*. 2017; 7(4):32-40. Available at: https://elibrary.ru/item.asp?id=32294100 (accessed 10.02.2018). (In Russian)

[10] 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, p. 18–31. DOI: https://doi.org/10.1007/978-3-319-46079-6_2

[11] Roganov V.A., Osipov V.I., Matveev G.A. Solving the 2D Poisson PDE by Gauss-Seidel method with parallel programming system OpenTS. *Program Systems: Theory and Applications*. 2016; 7(3):99-107. (In Russian) DOI: https://doi.org/10.25209/2079-3316-2016-7-3-99-107

[12] Bramas B. Fast sorting algorithms using AVX-512 on Intel Knights Landing // arXiv: 1704.08579 [cs.MS]. Available at: https://arxiv.org/abs/1704.08579 (accessed 10.02.2018).

[13] Sokolov A.P., Shhetinin V.N., Sapelkin A.S. Strength surface reconstruction using special parallel algorithm based on Intel MIC (Intel Many Integrated Core) architecture. *Program Systems: Theory and Applications*. 2016; 7(2):3-25. (In Russian) DOI: https://doi.org/10.25209/2079-3316-2016-7-2-3-25

[14] Dorris J., Kurzak J., Luszczek P. Task-Based Cholesky Decomposition on Knights Corner Using OpenMP. // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016. Vol. 9945, p. 544–562. DOI: https://doi.org/10.1007/978-3-319-46079-6_37

[15] Tobin J., Breuer A., Heinecke A. et al. Accelerating Seismic Simulations Using the Intel Xeon Phi Knights Landing Processor // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017. Vol. 10266, p. 139–157. DOI: https://doi.org/10.1007/978-3-319-58667-0_8

[16] McDoniel W., Höhnerbach M., Canales R. et al. LAMMPS' PPPM Long-Range Solver for the Second Generation Xeon Phi // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017. Vol. 10266, p. 61–78. DOI: https://doi.org/10.1007/978-3-319-58667-0_4

[17] Malas T., Kurth T., Deslippe J. Optimization of the Sparse Matrix-Vector Products of an IDR Krylov Iterative Solver in EMGeo for the Intel KNL Manycore Processor // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016. Vol. 9945, p. 378–389. DOI: https://doi.org/10.1007/978-3-319-46079-6_27

[18] Krzikalla O., Wende F., Höhnerbach M. Dynamic SIMD Vector Lane Scheduling // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016. Vol. 9945, p. 354–365. DOI: https://doi.org/10.1007/978-3-319-46079-6_25

[19] Cook B., Maris P., Shao M. High Performance Optimizations for Nuclear Physics Code MFDn on KNL // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016. Vol. 9945, p. 366–377. DOI: https://doi.org/10.1007/978-3-319-46079-6_26

[20] Rybakov A.A. Optimization of the problem of conflict detection with dangerous aircraft movement areas to execute on Intel Xeon Phi. *Programmnye produkty i sistemy* [Software & Systems]. 2017; 30(3):524-528. (In Russian) DOI: https://doi.org/10.15827/0236-235.X.030.3.524-528

[21] Sengupta D., Wang Y. Sundaram N. et al. High-Performance Incremental SVM Learning on Intel Xeon Phi Processors // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017. Vol. 10266, p. 120–138. DOI: https://doi.org/10.1007/978-3-319-58667-0_7

[22] Kronbichler M., Kormann K., Pasichnyk I. Fast Matrix-Free Discontinuous Galerkin Kernels on Modern Computer Architectures // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017. Vol. 10266, p. 237–255. DOI: https://doi.org/10.1007/978-3-319-58667-0_13

[23] Doerfler D., Deslippe J., Williams S. et al. Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016. Vol. 9945, p. 339–353. DOI: https://doi.org/10.1007/978-3-319-46079-6_24

[24] Rosales C., Cazes J., Milfeld K. A Comparative Study of Application Performance and Scalability on the Intel Knights Landing Processor // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016. Vol. 9945, p. 307–318. DOI: https://doi.org/10.1007/978-3-319-46079-6_22

[25] Dikarev N.I., Shabanov B.M., Shmelev A.S. Fused MultiplyAdders Using in Vector Dataflow Processor. *Program Systems: Theory and Applications*. 2015; 6(4):227-241. (In Russian) DOI: 10.25209/2079-3316-2015-6-4-227-241

Том 14, № 1. 2018 ISSN 2411-1473 sitito.cs.msu.ru

Submitted 20.12.2017; Revised 10.02.2018; Published 30.03.2018.

About the authors:

Leonid A. Benderskiy, Senior Researcher, Joint Supercomputer Center of the Russian Academy of Sciences, Scientific Research Institute for System Analysis of the Russian Academy of Sciences, SRISA (32a Leninsky prospect, Moscow 119334, Russia); ORCID: http://orcid.org/0000-0003-0529-3255, leosun.ben@gmail.com

Sergey A. Leshchev, Research associate, Joint Supercomputer Center of the Russian Academy of Sciences, Scientific Research Institute for System Analysis of the Russian Academy of Sciences, SRISA (32a Leninsky prospect, Moscow 119334, Russia); ORCID: http://orcid.org/0000-0001-6728-1028, sergey.leshchev@jscc.ru

Alexey A. Rybakov, Candidate of Physical and Mathematical Sciences, Lead researcher, Joint Supercomputer Center of the Russian Academy of Sciences, Scientific Research Institute for System Analysis of the Russian Academy of Sciences, SRISA (32a Leninsky prospect, Moscow 119334, Russia); ORCID: http://orcid.org/0000-0002-9755-8830, rybakov@jscc.ru