Повышение производительности векторного кода с помощью мониторинга плотности масок в векторных инструкциях

А.А. Рыбаков[1], А.Д. Чопорняк[2]

[1]МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН, Москва, Россия, rybakov@jscc.ru;

[2]МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН, Москва, Россия, adc@jscc.ru

Аннотация. Одним из ключевых факторов, влияющих на эффективность векторного кода, создаваемого с помощью AVX-512, является недостаточная плотность масок в векторных инструкциях. Это свидетельствует о том, что возможности векторных инструкций использованы не в полной мере, и во время выполнения кода обрабатывается только часть элементов векторов. Данная статья посвящена работе по созданию библиотеки многоверсионного кода, с помощью которой можно генерировать как векторный код, так и обычный код с эмуляцией векторных инструкций и возможностью мониторинга их отдельных характеристик, и в частности плотности масок. Результаты такого мониторинга можно использовать для повышения производительности векторного кода.

Ключевые слова: векторизация, повышение производительности, векторная маска, функции-интринсики, AVX-512.

1. Введение

Векторизация программного кода с использованием инструкций AVX-512 является низкоуровневой оптимизацией, с помощью которой можно добиться кратного увеличения производительности приложений [1]. Этому способствуют многочисленные особенности набора инструкций AVX-512 [2], и одной из основных особенностей является наличие масочных векторных инструкций, позволяющих применять операцию не ко всем элементам векторов-аргументов, а только к некоторым из них. Данная особенность является уникальной и позволяет применять векторизацию для сложного программного контекста, содержащего разветвленное управление, гнезда циклов и вызовы функций [3]. Однако использование масочных векторных инструкций может стать причиной снижения производительности кода, если плотность масок слишком низкая (то есть когда векторная инструкция обрабатывает не все элементы векторов, а только малую их часть, то смысл векторизации кода пропадает). Данный эффект снижения производительности особенно явно проявляется при использовании векторизации для циклов с нерегулярным числом итераций [4]. Для сглаживания негативных эффектов от снижения плотности масок в векторных инструкциях требуются возможности по мониторингу использования масок в процессе выполнения результирующего кода. При этом недостаточно просто численного показателя о средней плотности векторного кода или загрузке масок (в качестве примеров можно рассматривать показатели efficiency of vectorization, SIMD instructions per cycle, mask usage инструментов Intel VTune Amplifier и Intel Advisor [5,6]), требуется именно сбор статистики, по которой можно провести анализ плотности масок в векторных инструкциях в зависимости от внешних условий. Сбор такой статистики может быть обеспечен путем использования эмуляции векторных инструкций с параллельным накоплением произвольной информации, которая далее может быть проанализирована. Для реализации этой задачи выполняется разработка библиотеки создания многоверсионного кода, с помощью которой можно создавать как векторный код, использующий инструкции AVX-512, так и псевдовекторный код, в котором векторные инструкции эмулируются с помощью специальных классов, которые могут накапливать статистику времени выполнения.

Другой причиной создания библиотеки многоверсионного кода являются вопросы переносимости векторного кода. Во время оптимизации программ зачастую приходится сталкиваться с невозможностью компилятора автоматически векторизовать тот или иной фрагмент. Это может быть связано с разными причинами, в частности с недостатком у компилятора информации об отсутствии зависимостей между операциями. В этом случае на помощь приходят специальные функции-интринсики, которые могут быть использованы путем подключения библиотеки Intel <immintrin.h> [7]. Данные

функции-инстринсики являются обертками над векторными инструкциями и в процессе компиляции раскрываются в частности в инструкции AVX-512 (некоторые интринсики могут раскрываться в последовательности команд или даже в библиотечные вызовы). Конечно, если в коде программы используется такой подход, то программа сразу же становится непереносима на аппаратные платформы, в которых отсутствует поддержка тех инструкций, интринсики для которых были использованы.

Таким образом, создаваемая библиотека предназначена для генерации текста программы в трех режимах: режим генерации векторного кода для целевой аппаратной платформы с возможностью выбора допустимого набора векторных инструкций, режим эмуляции векторных инструкций для переносимости программы, режим сбора статистики для анализа производительности (что также выполняется с помощью эмуляции векторных инструкций). Схема получения результирующего исполняемого файла из исходного текста программы и библиотеки генерации многоверсионного кода показана на рис. 1.

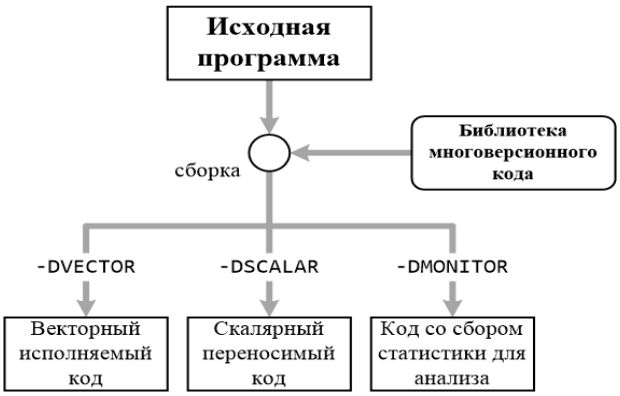

Рис. 1. Схема использования библиотеки многоверсионного кода

.

В данной статье на примере векторизации фрагмента кода из практической задачи рассмотрим влияние анализа плотности масок в векторных инструкциях на эффективность результирующего кода. В качестве примера будем использовать одну небольшую функцию из состава римановского решателя. Вопросы векторизации римановского решателя, в том числе и с помощью инструкций AVX-512, подробно изложены в работах [8,9].

2. Векторизация плоского цикла

Вначале кратко остановимся на понятии плоского цикла и подходе к его векторизации. Плоским циклом будем называть конструкцию следующего вида:

```
for (int i = 0; i < N; i++)
{
    <block(i)>
}
```

При этом тело цикла представляет собой блок вычислений со следующими свойствами. Во-первых, все обращения в память имеют вид x[i], то есть это обращения к некоторым массивам, при этом индекс массива совпадает с номером итерации плоского цикла, а также все массивы, встречающиеся внутри тела плоского цикла, не пересекаются по памяти. Данное требование к обращениям в память приводит к тому, что итерации плоского цикла становятся независимыми друг от друга, они могут выполняться в любом порядке, а значит и параллельно. Во-вторых, будем рассматривать только работу с вещественными числами одинарной точности (тип данных float). В один 512-битный векторный регистр помещается 16 таких элементов данных. Наконец, в-третьих, без ограничения общности будем считать, что количество итераций плоского цикла равно 16, в противном случае такой цикл всегда можно разбить на несколько более мелких циклов.

Такие плоские циклы представляют собой удобный контекст для векторизации, и в большинстве случаев они могут быть векторизованы с помощью векторных инструкций AVX-512 с помощью перевода тела цикла в предикатное представление и замены скалярных инструкций векторными аналогами, реализованными с помощью функций-интринсиков [10].

Многие практические вычислительные задачи состоят из выполнения однотипных вычислений, применяемых к разным наборам данных. Условно одну такую задачу можно описать как $F(a, b, c, ...) \rightarrow (x, y, z, ...)$, то есть к

некоторому набору входных параметров применяется функция F, выдающая в качестве результата набор выходных данных. Как уже говорилось, вызов данной функции F происходит многократно, при этом каждый раз используются различные наборы входных и выходных данных. Последовательность проведения вычислений может быть изменена, и вместо многих вызовов функции, работающей с набором параметров, можно рассмотреть другую функцию, работающую с массивами параметров в виде: G(a[], b[], c[], ...) → (x[], y[], z[], ...). Конечно внутри данной функции G должен быть организован цикл, на каждой итерации (с номером i) которого будет вызываться функция F со следующими наборами параметров: F(a[i], b[i], c[i], ...) → (x[i], y[i], z[i], ...). Нетрудно видеть, что такой цикл является плоским, а значит может быть легко векторизован.

Векторизация точного римановского решателя выполняется по описанной схеме. Детали реализации можно найти в [8], а в данной статье мы в качестве иллюстрации рассмотрим одну из функций, векторизация которой является наглядным примером важности анализа плотности масок векторных инструкций.

3. Мониторинг плотности масок

Итак, в качестве примера рассмотрим простую функцию prefun, входящую в состав римановского решателя (см. рис. 2). Данная функция содержит один оператор if-else, ветками которого являются блоки вычислений, содержащие простые арифметические операции, а также библиотечные функции pow и sqrt.

```
01 void prefun(float *f, float *fd,
02              float p, float dk, float pk, float ck)
03 {
04     if (p <= pk)
05     {
06         *f = G4 * ck * (pow(p/pk, G1) - 1.0);
07         *fd = (1.0 / (dk * ck)) * pow(p/pk, -G2);
08     }
09     else
10     {
11         float ak = G5 / dk;
12         float bk = G6 * pk;
13         float qrt = sqrt(ak / (bk + p));
14
15         *f = (p - pk) * qrt;
16         *fd = (1.0 - 0.5 * (p - pk) / (bk + p)) * qrt;
17     }
18 }
```

Рис. 2. Исходный вид функции prefun.

Таким образом, в функции содержится два линейных участка, которые должны выполняться под противоположными предикатами (p ≤ pk) и !(p ≤ pk). Внутри рассматриваемых линейных участков производятся арифметические вычисления, использующие входные параметры и глобальные константы (схематически граф потока управления функции prefun, а также графы зависимостей линейных участков данной функции изображены на рис. 3).

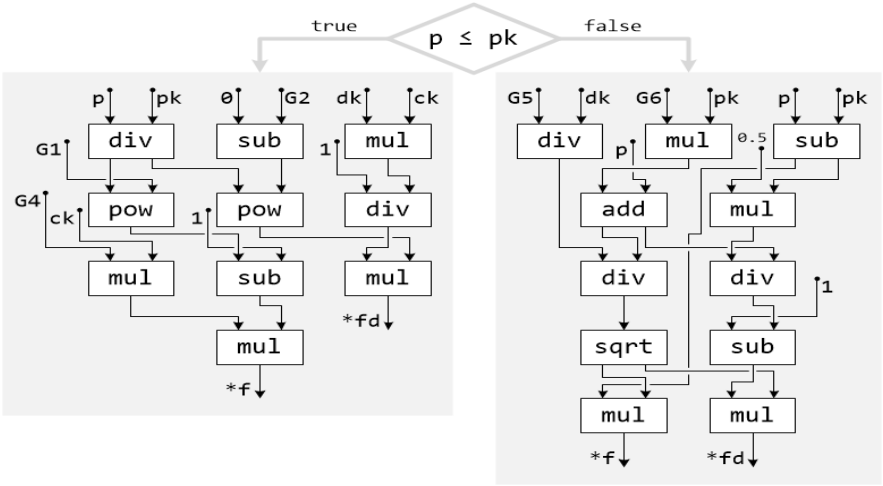

Рис. 3. Представление тела функции prefun в виде графа.

Для векторизации рассматриваемого программного контекста выполним объединение 16 вызовов функции prefun в один вызов функции prefun_16, которая вместо скалярных аргументов f, fd, p, dk, pk, ck оперирует с массивами соответствующих параметров fs, fds, ps, dks, pks, cks. При векторизации кода скалярные операции заменяются векторными аналогами, а скалярное условие заменяется на получение и использование векторной маски. При этом происходит слияние обеих ветвей исполнения под противоположными предикатами, векторные инструкции оперируют с векторами, содержащими по 16 элементов типа float, и с векторными масками, содержащими по 16 логических значений. На рис. 4 представлены две версии результирующего кода, полученные в двух разных режимах. Слева показан код, сгенерированный в режиме -DMONITOR, в котором 512-битные вектора и векторные маски представлены специальными библиотечными классами VectorF и Mask16, выполняющими операции над векторами с помощью эмуляции. Справа на рисунке показан результирующий код, сгенерированный в режиме -DVECTOR и использующий встроенные типы __m512 и __mmask16 и функции-интринсики для реализации векторных операций. Из рис. 4 можно заметить, что обе версии кода логически соответствуют друг другу.

```
01 void prefun_16(float *fs, float *fds,
02                 float *ps, float *dks, float *pks, float *cks)
03 {
04     VectorF f, fd, p, dk, pk, ck;
05     Mask16 cond, ncond;
06     VectorF z, g1, g2, g4, g5, g6, one, half;
07     VectorF t1, t2, t3, t4, t5;
08
09     // Инициализация необходимых векторов
10     ...
11
12     VectorF::CmpLE(nullptr, &p, &pk, &cond);
13     Mask16::Not(&cond, &ncond);
14
15     VectorF::Div(nullptr, &cond, &p, &pk, &t1);
16     VectorF::Pow(nullptr, &cond, &t1, &g1, &t2);
17     VectorF::Sub(nullptr, &cond, &t2, &one, &t3);
18     VectorF::Mul(nullptr, &cond, &g4, &ck, &t4);
19     VectorF::Mul(&f, &cond, &t4, &t3, &f);
20     VectorF::Sub(nullptr, &cond, &z, &g2, &t2);
21     VectorF::Pow(nullptr, &cond, &t1, &t2, &t3);
22     VectorF::Mul(nullptr, &cond, &dk, &ck, &t4);
23     VectorF::Div(nullptr, &cond, &one, &t4, &t5);
24     VectorF::Mul(&fd, &cond, &t5, &t3, &fd);
25
26     VectorF::Div(nullptr, &ncond, &g5, &dk, &t1);
27     VectorF::Mul(nullptr, &ncond, &g6, &pk, &t2);
28     VectorF::Add(nullptr, &ncond, &t2, &p, &t3);
29     VectorF::Div(nullptr, &ncond, &t1, &t3, &t4);
30     VectorF::Sqrt(nullptr, &ncond, &t4, &t4);
31     VectorF::Sub(nullptr, &ncond, &p, &pk, &t1);
32     VectorF::Mul(&f, &ncond, &t1, &t4, &f);
33     VectorF::Mul(nullptr, &ncond, &half, &t1, &t2);
34     VectorF::Div(nullptr, &ncond, &t2, &t3, &t1);
35     VectorF::Sub(nullptr, &ncond, &one, &t1, &t2);
36     VectorF::Mul(&fd, &ncond, &t2, &t4, &fd);
37
38     f.Store(fs);
39     fd.Store(fds);
40 }
```

```
01 void prefun_16(float *fs, float *fds,
02                 float *ps, float *dks, float *pks, float *cks)
03 {
04     __m512 f, fd, p, dk, pk, ck;
05     __mmask16 cond, ncond;
06     __m512 z, g1, g2, g4, g5, g6, one, half;
07     __m512 t1, t2, t3, t4, t5;
08
09     // Инициализация необходимых векторов
10     ...
11
12     cond = _mm512_cmp_ps_mask(p, pk, _MM_CMPINT_LE);
13     ncond = ~cond;
14
15     t1 = _mm512_mask_div_ps(z, cond, p, pk);
16     t2 = _mm512_mask_pow_ps(z, cond, t1, g1);
17     t3 = _mm512_mask_sub_ps(z, cond, t2, one);
18     t4 = _mm512_mask_mul_ps(z, cond, g4, ck);
19     f = _mm512_mask_mul_ps(f, cond, t4, t3);
20     t2 = _mm512_mask_sub_ps(z, cond, z, g2);
21     t3 = _mm512_mask_pow_ps(z, cont, t1, t2);
22     t4 = _mm512_mask_mul_ps(z, cond, dk, ck);
23     t5 = _mm512_mask_div_ps(z, cond, one, t4);
24     fd = _mm512_mask_mul_ps(fd, cond, t5, t3);
25
26     t1 = _mm512_mask_div_ps(z, ncond, g5, dk);
27     t2 = _mm512_mask_mul_ps(z, ncond, g6, pk);
28     t3 = _mm512_mask_add_ps(z, ncond, t2, p);
29     t4 = _mm512_mask_div_ps(z, ncond, t1, t3);
30     t4 = _mm512_mask_sqrt_ps(z, ncond, t4);
31     t1 = _mm512_mask_sub_ps(z, ncond, p, pk);
32     f = _mm512_mask_mul_ps(f, ncond, t1, t4);
33     t2 = _mm512_mask_mul_ps(z, ncond, half, t1);
34     t1 = _mm512_mask_div_ps(z, ncond, t2, t3);
35     t2 = _mm512_mask_sub_ps(z, ncond, one, t1);
36     fd = _mm512_mask_mul_ps(fd, ncond, t2, t4);
37
38     _mm512_store_ps(fs, f);
39     _mm512_store_ps(fds, fd);
40 }
```

Рис. 4. Генерация исполняемого кода в режимах -DMONITOR (слева) и -DVECTOR (справа).

Описанное представление функции prefun_16 в векторном виде приводит к ее ускорению в 5,8 раз на микропроцессорах с поддержкой AVX-512 по сравнению с невекторизованным оригиналом. Однако заметим, что в векторизованной версии содержится один общий линейный участок, являющийся объединением двух линейных участков, выполняющихся с использованием противоположных векторных масок cond и ncond. Это приводит к потере производительности в силу низкой плотности масок (в среднем каждая из них заполнена на 50%).

К сожалению данный факт является принципиально узким местом при использовании векторизации программного контекста с разветвленным управлением. Без использования дополнительной информации о профиле исполнения улучшить ситуацию не представляется возможным, и при слиянии разных ветвей исполнения производительность результирующего кода падает тем сильнее, чем выше вложенность условий [11].

4. Результаты мониторинга

Для анализа возможности повышения производительности получившегося кода в режиме эмуляции векторных инструкций внутри классов VectorF и Mask16 реализован функционал по сбору статистики плотности использованных масок (в данном случае анализировалась плотность маски cond). Вначале был осуществлен сбор статистики для случайных наборов входных данных, результат этой статистики приведен на рис. 5. На диаграмме видно распределение плотности маски, которое имеет нормальный характер. Это вполне ожидаемо, так как при случайных входных данных значение предиката ($p \leq pk$) также является случайным (0 или 1 с равной вероятностью), а плотность маски это количество всех ее истинных элементов, что можно трактовать как сумму ее единичных битов.

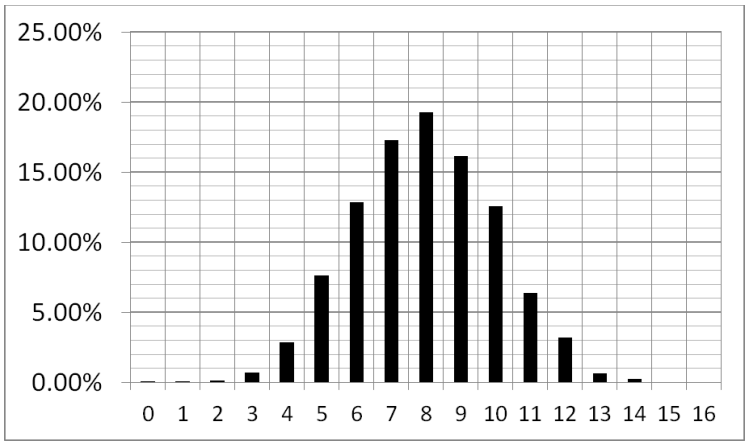

Рис. 5. Распределение плотности маски cond для случайно сгенерированных входных данных функции prefund

.

Однако наборы входных данных для физических задач имеют другую природу, и нельзя полагать, что эти наборы данных являются полностью независимыми и случайными. Если рассматривать задачу Римана о распаде произвольного разрыва, то в ней для каждого набора скалярных параметров рассматривается пара соседних ячеек расчетной сетки, для которых производится расчет газодинамических параметров на их общей границе [12]. При объединении нескольких вызовов функции римановского решателя в один вызов, обрабатывающий массивы входных параметров, с целью векторизации, наборы параметров, обрабатываемых на соседних итерациях векторизуемого плоского цикла, оказываются достаточно близкими. Можно условно считать, что каждый параметр внутри векторизуемого цикла изменяется достаточно медленно. Если рассматривать наш пример с функцией prefun_16, то можно сделать предположение, что значения массива ps изменяются медленно при изменении индекса i, аналогично значения массива pks изменяются медленно при изменении индекса i. Из этого можно сделать вывод, что также медленно изменяется и значение предиката ($ps[i] \leq pks[i]$). Но предикат это логическая величина, которая может принимать только значения 0 или 1. Это означает только то, что с большой вероятностью на продолжительных интервалах изменения индекса i значение предиката ($ps[i] \leq pks[i]$) остается константным. При векторизации кода предикаты объединяются в маски, таким образом, интервалы с константными предикатами будут формировать маски со значениями 0x0 или 0xFFFFFF (на рис. 6 представлена иллюстрация данного свойства задач физических расчетов). Конечно, если линейный участок в векторизованном коде часто попадает под маску со значением 0x0, то проверка маски на пустоту перед выполнением имеет смысл. Описанное свойство повышенной частоты нулевых и полных масок в векторизованном коде характерно для физических расчетных задач, таких как задачи газовой динамики, механики твердого тела, молекулярной динамики. И наоборот, это свойство не выполняется для дискретных задач, например задач сортировки, кодирования, задач комбинаторики и других.

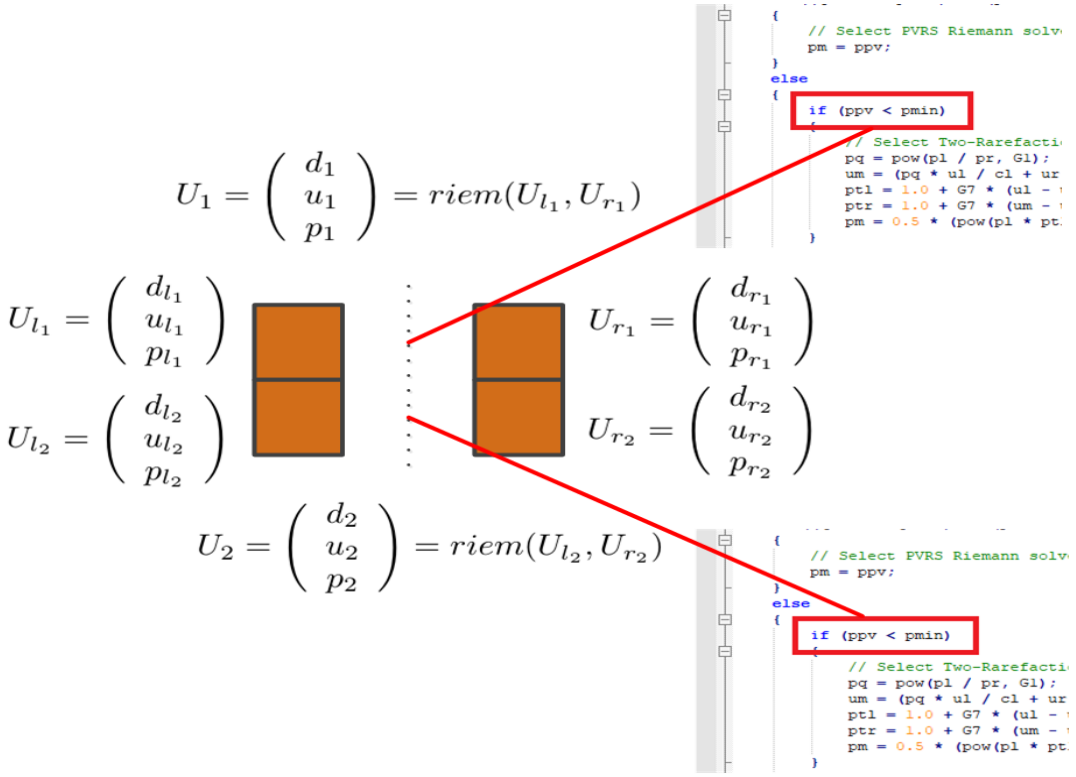

Рис. 6. Иллюстрация высокой вероятности совпадения предикатов для соседних итераций плоского цикла в физических расчетах.

Для рассматриваемой в данной статье функции prefun был осуществлен мониторинг плотности маски cond при использовании реальных входных данных, полученных при расчетах течения газа. Диаграмма с распределением плотности маски представлена на рис. 7, и данное распределение не имеет ничего общего с распределением, полученным на случайных входных данных. На диаграмме можно наблюдать почти 50% долю нулевых масок и превышающую 10% долю полных масок.

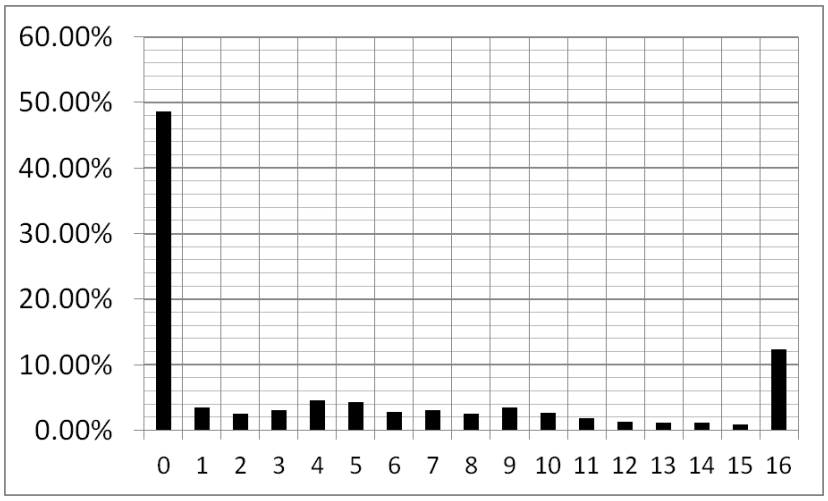

Рис. 7. Распределение плотности маски cond для реальных расчетных данных.

По результатам мониторинга соответствующие участки кода, выполняемые под масками cond и ncond, были помещены под условия if (cond) и if (ncond) соответственно, что позволило добиться дополнительного ускорения результирующего кода в 1,7 раз. Итоговое ускорение функции prefun в результате векторизации составило 9,9 раз.

Заметим, что рассматриваемый пример является тривиальным случаем, содержащим все-

го один условный оператор. Как правило, векторизуемые функции содержат множество условий. Добавление проверок для всех линейных участков в таких функциях приводит к деградации производительности, поэтому с помощью мониторинга плотности масок необходимо отыскивать те ветки, помещение под условие которых действительно имеет смысл.

5. Заключение

В рамках работы над проектом РФФИ № 20-07-00594 выполняется разработка библиотеки генерации многоверсионного кода, с помощью которой можно создавать как векторный код, предназначенный для целевой аппаратной платформы, так и код с эмуляцией векторных инструкций. В режиме эмуляции векторных инструкций реализованы возможности по сбору статистики времени выполнения программы, в частности собирается статистика по плотности векторных масок. Данная информация может быть использована для повышения производительности векторизуемых приложений.

В данной статье показано применение разрабатываемой библиотеки генерации многоверсионного кода на примере отдельной функции в составе римановского решателя и продемонстрирован положительный эффект от мониторинга плотности векторных масок, проявляющийся для расчетных физических задач.

Работа выполнена при поддержке гранта РФФИ № 20-07-00594.

Improving vector code performance by monitoring masks density in vector instructions

A.A. Rybakov, A.D. Chopornyak

Abstract. One of the key factors affecting the effectiveness of vector code created using the AVX-512 is the insufficient density of masks in vector instructions. This indicates that the capabilities of vector instructions are not fully used, and only part of the vector elements are processed during code execution. This article is devoted to the creation of a library of multi-version code, with which you can generate both vector code and regular code with emulation of vector instructions and the ability to monitor their individual characteristics, and in particular the density of the masks. The results of such monitoring can be used to improve the performance of vector code.

Keywords: vectorization, performance increase, vector mask, intrinsic functions, AVX-512.

Литература

1. Vectorization opportunities for improved performance with Intel AVX-512. https://techdecoded.intel.io/resources/vectorization-opportunities-for-improved-performance-with-intel-avx-512, дата обращения 18.06.2020.

2. 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, October 2019.

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

4. А.А. Рыбаков, С.С. Шумилин. Исследование эффективности векторизации гнезд циклов с нерегулярным числом итераций. // Программные системы: Теория и алгоритмы, 2019, Т. 10, № 4 (43), с. 77–96.

5. Analyze vector instruction set with Intel VTune Amplifier. https://software.intel.com/content/www/us/en/develop/documentation/itac-vtune-mpi-openmp-tutorial-lin/top/analyze-vector-instruction-set-with-intel-vtune-amplifier.html, дата обращения 18.06.2020.

6. K. O'Leary. Intel VTune Amplifier and Intel Advisor. Optimization workshop, 2019, https://www.nccs.nasa.gov/sites/default/files/Optimization.pdf, дата обращения 18.06.2020.

7. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide, дата обращения 18.06.2020.

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

9. C.R. Ferreira, K.T. Mandli, M. Bader. Vectorization of Riemann solvers for the single- and multi-layer shallow water equations. Proceedings of the 2018 International Conference on High Performance

Computing and Simulation, HPCS 2018 (16-20 July 2018, Orleans, France), 2018, p. 415–422.

10. А.А. Рыбаков. Векторизация нахождения пересечения объемной и поверхностной сеток для микропроцессоров с поддержкой AVX-512. Труды НИИСИ РАН, 2019, Т. 9, № 5, с. 5–14.

11. А.А. Рыбаков, С.С. Шумилин. Векторизация сильно разветвленного управления с помощью инструкций AVX-512. Труды НИИСИ РАН, 2018, Т. 8, № 4, с. 114–126.

12. E.F. Toro. Riemann solvers and numerical methods for fluid dynamics: A practical introduction. 2nd edition, Springer, Berlin-Heidelberg, 1999, 645 p.