ISSN 2079-3316 **ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ** **т. 10**, № 3(42), с. 59–80

ББК 32.973.202-018.2
ГРНТИ 50.07.05,50.33.04
УДК 519.68+004.272.25

А. А. Рыбаков, С. С. Шумилин

Векторизация римановского решателя с использованием набора инструкций AVX-512

Аннотация. Численные методы, базирующиеся на решении задачи Римана о распаде произвольного разрыва, крайне требовательны к вычислительным ресурсам. Для применения данных численных методов на современных расчетных сетках требуется использование суперкомпьютера. Среди различных инструментов повышения производительности суперкомпьютерных приложений можно выделить векторизацию программного кода. Набор инструкций AVX-512 обладает рядом уникальных возможностей, позволяющих применить векторизацию к программному контексту римановского решателя, что ведет к значительному ускорению решателя. На примере точного римановского решателя рассматривается практический подход к векторизации разнообразного программного контекста, включая простые линейные участки, регионы со сложным управлением, а также вложенные циклы. В основе рассматриваемого подхода лежит возможность одновременного выполнения на одном процессорном ядре нескольких экземпляров некоторой чистой функции. Данная возможность достигается путем перевода программного кода в предикатную форму и использования векторных инструкций. При этом количество одновременно выполняющихся экземпляров равно ширине вектора. Показано, что использование возможностей набора команд AVX-512 позволяет успешно векторизовать рассматриваемый программный контекст. Предложенный подход может быть применен для векторизации широкого спектра приложений.

Ключевые слова и фразы: задача Римана о распаде произвольного разрыва, римановский решатель, AVX-512, KNL, векторизация, функции-интринсики.

Работа выполнена в МСЦ РАН в рамках государственного задания по теме 0065-2019-0016 при поддержке гранта РФФИ № 18-07-00638_а.

10.25209/2079-3316-2019-10-3-59-80

Введение

Решение задачи Римана используется в численных методах для нестационарных задач с большими разрывами [1]. При этом данный подход может применяться как для обычных однокомпонентных газодинамических уравнений, так и для уравнений многокомпонентной газовой динамики [2]. На точном или приближенном решении задачи Римана основывается метод Годунова решения систем нестационарных уравнений газовой динамики [3]. В данном методе на каждой итерации вычислений задача Римана решается на каждой грани каждой ячейки расчетной сетки для определения потоков через эти грани. Ввиду того, что размеры расчетных сеток, которые используются в наши дни для проведения вычислений, составляют десятки, сотни миллионов ячеек и более, то для эффективного использования численных методов с римановскими решателями необходимо использование суперкомпьютеров и применение всего спектра подходов по распараллеливанию вычислений. Для повышения эффективности суперкомпьютерных вычислений используются такие инструменты, как MPI, OpenMP, OpenACC и другие.

Наиболее низкоуровневым подходом, применяемым для повышения производительности суперкомпьютерных приложений, является векторизация кода. Использование специального набора инструкций AVX-512 обладает уникальными возможностями, благодаря которым возможно создание эффективного параллельного кода, что приводит к кратному ускорению суперкомпьютерных приложений.

Повышение эффективности работы римановских решателей на параллельных архитектурах является предметом многих исследований, которые продолжаются до сих пор, так как постоянно появляются новые архитектуры, обладающие новыми особенностями. В работе [4] продемонстрирован алгоритм распараллеливания численных методов, использующих приближенный римановский решатель для задач магнитогидродинамики, приводящий к параллельной работе на многопроцессорной машине с эффективностью распараллеливания, близкой к единице. В работе [5] приведено комплексное исследование по распараллеливанию вычислительных методов, базирующихся на римановском решателе, с использованием MPI, OpenMP и графических ускорителей. В результате продемонстрированы хорошие показатели по масштабируемости, в частности, показатель сильной масштабируемости на отметке 85%. Работа [6] освещает использование

инструмента PetClaw, позволяющего применять численные методы на базе римановского решателя на нескольких тысячах вычислительных процессов с эффективностью распараллеливания, близкой к единице. В работах [7], [8] освещены вопросы параллельной реализации программы моделирования астрофизических течений AstroPhi. Показано, что текущая реализация позволяет достичь почти 50% от скалярной пиковой производительности сопроцессоров Intel Xeon Phi KNC и приводится оценка ожидаемого эффекта от будущей векторизации (более 80% суммарной пиковой производительности). В качестве одного из самых подробных исследований по векторизации римановских решателей можно отметить работу [9], в которой рассматриваются вопросы оптимизации приближенного римановского решателя для решения уравнений мелкой воды. Примененные подходы позволили добиться ускорения в 6-7 раз на сопросессорах Intel Xeon Phi KNC на вещественных операциях с одинарной точностью. В дальнейшем это исследование получило продолжение и в работе [10] описаны методы, в помощью которых было получено ускорение в диапазоне 2.4-6.5 раз для операций с двойной точностью (вычисления проводились на микропроцессорах Intel Xeon Phi KNL).

В данной статье рассматривается векторизация точного римановского решателя, содержащего сложные конструкции, в том числе решение нелинейного уравнения и сложное управление. В статье показано, что использование особенностей набора инструкций AVX-512 позволяет успешно векторизовать данный программный контекст.

Главным вкладом данной статьи является разработка подходов к векторизации сложного программного контекста с использованием уникального набора инструкций AVX-512. Точный римановский решатель подходит для этой цели весьма удачно, так как имеет компактную реализацию и в то же время вмещает в себя целый ряд особенностей программного контекста, требующих своих приемов при проведении векторизации.

В первом разделе статьи приводится краткое описание особенностей набора инструкций AVX-512, позволящих выполнять векторизацию сложного программного контекста с помощью предикатного представления кода. В разделе 2 представлена общая схема точного римановского решателя, а разделы 3– 45 раскрывают особенности векторизации его отдельных частей (простого линейного контекста, участков со сложным управлением и вложенного цикла). В разделе 6 представлены

результаты численных экспериментов на микропроцессоре Intel Xeon Phi KNL, а в разделе 7 приведено сравнение с близкими работами.

1. Особенности набора инструкций AVX-512

Набор инструкций AVX-512 представляет собой расширение 256-битных инструкций AVX архитектуры Intel x86. Данный набор инструкций поддержан в семействах микропроцессоров Intel Xeon Phi второго поколения (Knight Landing, KNL) и Intel Xeon Skylake.

Инструкции AVX-512 работают с 512-битными векторными регистрами (`zmm`), которые могут содержать целочисленные или вещественные данные. Каждый `zmm` регистр способен вместить, например, 8 вещественных значений двойной точности (`double`) или 16 вещественных значений одинарной точности (`float`). Набор инструкций AVX-512 реализует множество операций с векторными аргументами, среди которых арифметические операции, операции сравнения, операции чтения из памяти и записи в память, трансцендентные операции, комбинированные операции вида $\pm a \cdot b \pm c$, операции перестановки элементов векторов и другие.

Для поддержки выборочного применения операций над упакованными данными к конкретным элементам векторов большинство инструкций AVX-512 использует специальные регистры-маски в качестве аргументов. Всего таких регистров 8 (`k0-k7`). При выполнении векторной операции элемент результирующего вектора будет вычислен только если бит маски с соответствующим номером выставлен в единицу, в противном случае выполнение операции для данных элементов будет проигнорировано. Данная уникальная возможность набора инструкций AVX-512 обеспечивает реализацию предикатного режима исполнения [11], который поддержан в таких архитектурах, как ARM или «Эльбрус» [12]. Наличие предикатного режима исполнения позволяет применять оптимизацию слияния ветвей исполнения и, таким образом, избавляться от лишних операций передачи управления, что помогает создавать высокоэффективный параллельный код.

Из других важных особенностей набора инструкций AVX-512 можно отметить операции множественного чтения элементов векторов, расположенных в памяти с произвольными смещениями от базового адреса, а также аналогичные операции записи элементов векторов в память с произвольными смещениями (операции `gather/scatter`). Хотя

данные операции крайне медленные, они в некоторых случаях помогают существенно упростить логику векторизованного кода. Также следует отметить большое разнообразие различных операций перестановки, перемешивания, дублирования, пересылки элементов векторов, что позволяет произвольным образом менять порядок обработки данных. Также существенное ускорение способны принести комбинированные операции, объединяющие операцию умножения и сложения в одну операцию.

Для упрощения применения векторных инструкций при оптимизации программного кода для компилятора icc разработаны специальные функции-интринсики (они определены в заголовочном файле `immintrin.h`) [13]. Данные функции покрывают не все множество инструкций AVX-512, однако избавляют от необходимости вручную писать ассемблерный код. Вместо этого предоставляется возможность оперировать встроенными типами данных для 512-битных векторов и использовать их при работе с функциями-инстринсиками как обычные базовые типы (при построении компилятором исполняемого кода для этих типов данных будут использованы регистры `zmm`). Некоторые функции-инстринсики соответствуют не одной отдельной команде, а целой последовательности, как например группа функций `reduce`, другие же просто раскрываются в вызов библиотечной функции (например, тригонометрические функции или функция `hypot`).

В данной статье будут рассмотрены особенности римановского решателя в классической реализации E. F. Toro [14] и описаны подходы, позволяющие векторизовать данный решатель для параллельного решения сразу 16 экземпляров задачи Римана о распаде произвольного разрыва.

2. Описание римановского решателя

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

$$(1) \qquad U_l = \begin{pmatrix} d_l \\ u_l \\ p_l \end{pmatrix}, U_r = \begin{pmatrix} d_r \\ u_r \\ p_r \end{pmatrix}, U = \begin{pmatrix} d \\ u \\ p \end{pmatrix} = riem(U_l, U_r).$$

В формуле (1) через d_l, u_l, p_l обозначены плотность, скорость и давление газа слева от разрыва (они объединены в структуру U_l – состояние газа слева от разрыва). Аналогично через d_r, u_r, p_r обозначены плотность, скорость и давление газа справа от разрыва, объединенные в состояние газа U_r. Переменными d, u, p обозначены плотность, скорость и давление газа, полученные в результате решения задачи Римана.

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

На рисунке 1 показана схема работы римановского решателя с обозначенными потоками данных и вызовами всех входящих в реализацию функций: Функция `riemann` осуществляет вычисление скорости звука справа и слева, выполняет проверку на образование вакуума и последовательно вызывает функции `starpu` и `sample`. Функция `starpu` вычисляет значения скорости и давления в среднем регионе между левой и правой волнами (star region), при этом функция содержит цикл с неизвестным количеством итераций для решения нелинейного уравнения итерационным методом Ньютона, внутри которого расположены вызовы других функций (`prefun`). Функции `guessp` и `prefun` содержат только арифметические вычисления и простые условия и являются наиболее простыми с точки зрения векторизации. Наконец последняя функция sample определяет окончательную конфигурацию разрыва путем вычисления множества условий. Данная функция содержит очень разветвленное управление, вложенность условий в ней достигает четырех, что затрудняет применение векторизации.

В процессе счета с помощью численных методов, базирующихся на римановском решателе, выполняется множество вызовов функции `riemann` с различными наборами входных данных (на каждой итерации счета выполняется один вызов для каждой грани каждой ячейки расчетной сетки). Так как функция `riemann` является чистой, то вызовы для разных наборов входных данных (`dl`, `ul`, `pl`, `dr`, `ur`, `pr`) являются независимыми и возникает желание объединения вызовов с целью

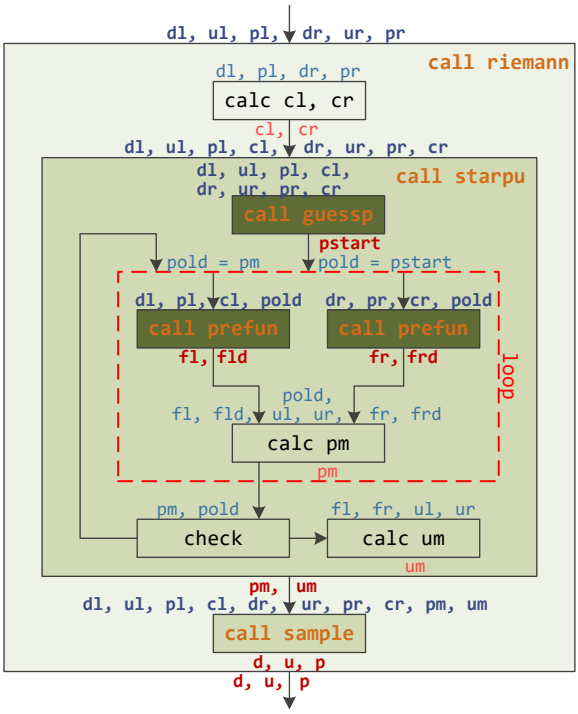

Рисунок 1. Схема потока данных в римановском решателе

эффективного задействования векторных (поэлементных) инструкций. В качестве такого объединенного вызова будем рассматривать функцию, в которую вместо атомарных данных типа `float` будут подаваться соответствующие векторы, содержащие по 16 элементов:

$$(2) \qquad \overline{U_l} = \begin{pmatrix} \overline{d_l} \\ \overline{u_l} \\ \overline{p_l} \end{pmatrix}, \overline{U_r} = \begin{pmatrix} \overline{d_r} \\ \overline{u_r} \\ \overline{p_r} \end{pmatrix}, \overline{U} = \begin{pmatrix} \overline{d} \\ \overline{u} \\ \overline{p} \end{pmatrix} = riem(\overline{U_l}, \overline{U_r}).$$

В формуле (2) все переменные $\overline{d_l}$, $\overline{u_l}$, $\overline{p_l}$, $\overline{d_r}$, $\overline{u_r}$, $\overline{p_r}$, $\overline{d}$, $\overline{u}$, $\overline{p}$ являются векторами длины 16. Например, вектор $\overline{d}$ содержит 16 значений плотности газа, полученных при решении 16 задач Римана, объединенных в один вызов. Аналогично с другими переменными.

При этом с векторными данными можно производить те же действия, что и с базовыми типами – выполнять вычисления, передавать в

функции, возвращать в качестве результата. В процессе оптимизации для наглядности будем избегать подстановки тела функции в точку вызова. Таким образом, в результате векторизации нашей целью является получение векторных аналогов всех используемых в римановском решателе описанных выше функций.

3. Векторизация простого контекста

Наиболее простым контекстом для векторизации вычислений при объединении вызовов является функция `prefun` (функция `guessp` обладает похожими свойствами и не рассматривается). Невекторизованный вариант функции представлен на рисунке 2:

```
01 void prefun(float &f, float &fd, float &p,
02              float &dk, float &pk, float &ck)
03 {
04     float ak, bk, pratio, qrt;
05
06     if (p <= pk)
07     {
08         pratio = p / pk;
09         f = G4 * ck * (pow(pratio, G1) - 1.0);
10         fd = (1.0 / (dk * ck)) * pow(pratio, -G2);
11     }
12     else
13     {
14         ak = G5 / dk;
15         bk = G6 * pk;
16         qrt = sqrt(ak / (bk + p));
17         f = (p - pk) * qrt;
18         fd = (1.0 - 0.5 * (p - pk) / (bk + p)) * qrt;
19     }
20 }
```

Рисунок 2. Оригинальная версия функции `prefun`

Данный код содержит только простые арифметические операции, извлечение квадратного корня, возведение в степень и сравнение. Для всех этих действий в наборе команд AVX-512 предусмотрены векторные аналоги. Код функции векторизуется путем замены арифметических операций на векторные аналоги. Вычисление значений, находящихся под условием (`if-else`) векторизуется путем использования векторных предикатов, полученных с помощью векторного сравнения (см. рисунок 3). В проведенной векторизации функции `prefun` стоит отметить три момента.

Так как векторная операция деления является медленной, то использовались следующие тождества для сокращения количества

```
01 void prefun_16(__m512 *f, __m512 *fd, __m512 p,
02                __m512 dk, __m512 pk, __m512 ck,
03                __mmask16 m)
04 {
05     __m512 pratio, ak, bkp, ppk, qrt;
06     __mmask16 cond, ncond;
07
08     // Conditions.
09     cond = _mm512_mask_cmp_ps_mask(m, p, pk, _MM_CMPINT_LE);
10     ncond = m & ~cond;
11
12     // The first branch.
13     if (cond != 0x0)
14     {
15         pratio = _mm512_mask_div_ps(z, cond, p, pk);
16         *f = _mm512_mask_mul_ps(*f, cond, MUL(g4, ck),
17                 SUB(_mm512_mask_pow_ps(z, cond, pratio, g1), one));
18         *fd = _mm512_mask_div_ps(*fd, cond,
19                   _mm512_mask_pow_ps(z, cond, pratio, SUB(z, g2)),
20                   MUL(dk, ck));
21     }
22
23     // The second branch.
24     if (ncond != 0x0)
25     {
26         ak = _mm512_mask_div_ps(z, ncond, g5, dk);
27         bkp = FMADD(g6, pk, p);
28         ppk = SUB(p, pk);
29         qrt = _mm512_mask_sqrt_ps(z, ncond,
30                                   _mm512_mask_div_ps(z, ncond, ak, bkp));
31         *f = _mm512_mask_mul_ps(*f, ncond, ppk, qrt);
32         *fd = _mm512_mask_mul_ps(*fd, ncond, qrt,
33                   FNMADD(_mm512_mask_div_ps(z, ncond, ppk, bkp),
34                          SET1(0.5), one));
35     }
36 }
```

Рисунок 3. Векторизованная версия функции `prefun`

делений:

$$\frac{a}{\frac{b}{c}} = \frac{ac}{b}, \quad \frac{\frac{a}{b}}{c} = \frac{a}{bc}. \tag{3}$$

Так как функция `prefun` вызывается внутри цикла в функции `starpu`, то в ее реализацию необходимо добавить дополнительный аргумент-маску, по которой выбираются элементы, обрабатываемые в данном вызове (рисунок 3, строка 03). Более подробно это будет описано ниже, когда будет обсуждаться векторизация функции `starpu`.

Еще одним важным моментом векторизации является оптимизация работы с условиями. В процессе выполнения векторизации у нас сформировались две группы векторных инструкций, выполняемых под предикатами `cond` (рисунок 3, строки 15–20) и `ncond` (рисунок 3, строки 26–34). Так как для физических задач характерно непрерывное изменение физических величин, то можно утверждать, что наборы данных, обрабатываемых в векторизованном вызове, близки. Другими словами, все векторы, фигурирующие в вычислениях, содержат

близкие значения. Это же утверждение относится и к векторам предикатов. Действительно, сбор статистики исполнения показал, что для физических задач наиболее частыми значениями векторов предикатов (в данном случае `cond` и `ncond`) являются значения `0x0` и `0xFFFF`. Таким образом, проверки предикатных векторных регистров на пустоту (рисунок 3, строки 13 и 24) сразу отсекает целые блоки лишних операций, выполняемых с нулевой маской.

4. Векторизация сильно разветвленных условий

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

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

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

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

После выполнения сокращения кода количество листовых узлов дерева условий сократилось до трех. Однако даже в этом случае прямое слияние кода с использованием векторных предикатов оказалось неэффективным. Виной тому послужил маловероятный участок кода, тяжелые операции, среди которых присутствуют вызовы функции возведения в степень (рисунок 4, строки 033–036). Векторный предикат данного участка кода более чем в 95% случаев имеет значение `0x0`,

```
001 void sample(float dl, float ul, float pl, float cl,
002              float dr, float ur, float pr, float cr,
003              const float pm, const float um,
004              float &d, float &u, float &p)
005 {
006     float c, cml, cmr, pml, pmr, shl, shr, sl, sr, stl, str;
007
008     if (0.0 <= um)
009     {
010         if (pm <= pl)
011         {
012             shl = ul - cl;
013
014             if (0.0 <= shl)
015             {
                    < d, u, p = dl, ul, pl >
019             }
020             else
021             {
022                 cml = cl * pow(pm / pl, G1);
023                 stl = um - cml;
024
025                 if (0.0 > stl)
026                 {
027                     d = dl * pow(pm / pl, 1.0 / GAMA);
028                     u = um;
029                     p = pm;
030                 }
031                 else
032                 {
                        < high-density code, low prob >
037                 }
038             }
039         }
040         else
041         {
042             pml = pm / pl;
043             sl = ul - cl * sqrt(G2 * pml + G1);
044
045             if (0.0 <= sl)
046             {
                    < d, u, p = dl, ul, pl >
050             }
051             else
052             {
053                 d = dl * (pml + G6) / (pml * G6 + 1.0);
054                 u = um;
055                 p = pm;
056             }
057         }
058     }
059     else
060     {
            < symmetrical branch >
109     }
110 }
```

Рисунок 4. Оригинальная версия функции `sample`

поэтому перед выполнением данного участка кода целесообразно выполнить проверку данного предиката на пустоту (что соответствует выносу маловероятной ветви исполнения из тела функции). Вынос маловероятной ветки исполнения из основного программного контекста способен существенно ускорить исполняемый код, так как наличие большого количества подобных редких вычислений может служить причиной к отказу от векторизации [15].

```
01 void sample_16(__m512 dl, __m512 ul, __m512 pl, __m512 cl,
02                __m512 dr, __m512 ur, __m512 pr, __m512 cr,
03                __m512 pm, __m512 um,
04                __m512 *d, __m512 *u, __m512 *p)
05 {
06     __m512 c, ums, pms, sh, st, s, uc;
07     __mmask16 cond_um, cond_pm, cond_sh, cond_st, cond_s, cond_sh_st;
08
09     // d/u/p/c/ums
10     cond_um = _mm512_cmp_ps_mask(um, z, _MM_CMPINT_LT);
11     *d = _mm512_mask_blend_ps(cond_um, dl, dr);
12     *u = _mm512_mask_blend_ps(cond_um, ul, ur);
13     *p = _mm512_mask_blend_ps(cond_um, pl, pr);
14     c = _mm512_mask_blend_ps(cond_um, cl, cr);
15     ums = um;
16     *u = _mm512_mask_sub_ps(*u, cond_um, z, *u);
17     ums = _mm512_mask_sub_ps(ums, cond_um, z, ums);
18
19     // Calculate main values.
20     pms = DIV(pm, *p);
21     sh = SUB(*u, c);
22     st = FNMADD(POW(pms, g1), c, ums);
23     s = FNMADD(c, SQRT(FMADD(g2, pms, g1)), *u);
24
25     // Conditions.
26     cond_pm = _mm512_cmp_ps_mask(pm, *p, _MM_CMPINT_LE);
27     cond_sh = _mm512_mask_cmp_ps_mask(cond_pm, sh, z, _MM_CMPINT_LT);
28     cond_st = _mm512_mask_cmp_ps_mask(cond_sh, st, z, _MM_CMPINT_LT);
29     cond_s = _mm512_mask_cmp_ps_mask(~cond_pm, s, z, _MM_CMPINT_LT);
30
31     // Store.
32     *d = _mm512_mask_mov_ps(*d, cond_st,
33                             MUL(*d, POW(pms, SET1(1.0 / GAMA))));
34     *d = _mm512_mask_mov_ps(*d, cond_s MUL(*d, DIV(ADD(pms, g6),
35                                                    FMADD(pms, g6, one))));
36     *u = _mm512_mask_mov_ps(*u, cond_st | cond_s, ums);
37     *p = _mm512_mask_mov_ps(*p, cond_st | cond_s, pm);
38
39     // Low prob - ignore it.
40     cond_sh_st = cond_sh & ~cond_st;
41     if (cond_sh_st != 0x0)
42     {
43         *u = _mm512_mask_mov_ps(*u, cond_sh_st, MUL(g5, FMADD(g7, *u, c)));
44         uc = DIV(*u, c);
45         *d = _mm512_mask_mov_ps(*d, cond_sh_st, MUL(*d, POW(uc, g4)));
46         *p = _mm512_mask_mov_ps(*p, cond_sh_st, MUL(*p, POW(uc, g3)));
47     }
48
49     // Final store.
50     *u = _mm512_mask_sub_ps(*u, cond_um, z, *u);
51 }
```

Рисунок 5. Векторизованная версия функции `sample`

Для остального кода можно производить слияние с учетом замечаний, описанных в предыдущем разделе. В результате векторизованная функция `sample` была ускорена более чем в 10 раз. Итоговый векторный код представлен на рисунке 5. В данном коде строки 11–14 соответствуют начальной инициализации, строки 16, 17 и 50 отвечают за замену переменных для слияния симметричных участков кода, а маловероятная ветка выделена в блоке, расположенном в строках 41–47.

5. Векторизация гнезда циклов

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

```
01 void starpu(float dl, float ul, float pl, float cl,
02              float dr, float ur, float pr, float cr,
03              float &p, float &u)
04 {
05     const int nriter = 20;
06     const float tolpre = 1.0e-6;
07     float change, fl, fld, fr, frd, pold, pstart, udiff;
08
09     guessp(dl, ul, pl, cl, dr, ur, pr, cr, pstart);
10     pold = pstart;
11     udiff = ur - ul;
12
13     int i = 1;
14
15     for ( ; i <= nriter; i++)
16     {
17         prefun(fl, fld, pold, dl, pl, cl);
18         prefun(fr, frd, pold, dr, pr, cr);
19         p = pold - (fl + fr + udiff) / (fld + frd);
20         change = 2.0 * abs((p - pold) / (p + pold));
21
22         if (change <= tolpre)
23         {
24             break;
25         }
26
27         if (p < 0.0)
28         {
29             p = tolpre;
30         }
31
32         pold = p;
33     }
34
35     if (i > nriter)
36     {
37         cout << "divergence in Newton-Raphson iteration" << endl;
38         exit(1);
39     }
40
41     u = 0.5 * (ul + ur + fr - fl);
42 }
```

Рисунок 6. Оригинальная версия функции `starpu`

итераций (рисунок 6). Цикл, расположенный в данной функции в строках 15–33 кроме неизвестного количества итераций содержит также условные переходы (`if, break`) и вызовы функций `prefun`, что также усложняет его векторизацию.

Перед выполнением векторизации данный цикл необходимо преобразовать в предикатную форму, в которой тело не должно содержать операций перехода. Все инструкции цикла выполняются под своими предикатами, а выполнение цикла прерывается при условии обнуления всех предикатов. Данный механизм описан в работе [16] применительно к векторизации сортировки Шелла, а также в [17] применительно к построению множества Мандельброта. При этом стоит заметить, что вызовы функций `prefun` также должны обладать соответствующими предикатами. После преобразования тела цикла в предикатную форму, он может быть векторизован, после чего предикаты инструкций заменятся на векторные регистры-маски (именно в этом месте появляется дополнительный параметр векторизованной функции `prefun` в

виде маски). Результат векторизации функции `starpu` представлен на рисунке 7:

```
01 void starpu_16(__m512 dl, __m512 ul, __m512 pl, __m512 cl,
02                __m512 dr, __m512 ur, __m512 pr, __m512 cr,
03                __m512 *p, __m512 *u)
04 {
05     __m512 two, tolpre, tolpre2, udiff, pold, fl, fld, fr, frd, change;
06     __mmask16 cond_break, cond_neg, m;
07     const int nriter = 20;
08     int iter = 1;
09
10     two = SET1(2.0);
11     tolpre = SET1(1.0e-6);
12     tolpre2 = SET1(5.0e-7);
13     udiff = SUB(ur, ul);
14
15     guessp_16(dl, ul, pl, cl, dr, ur, pr, cr, &pold);
16
17     // Start with full mask.
18     m = 0xFFFF;
19
20     for (; (iter <= nriter) && (m != 0x0); iter++)
21     {
22         prefun_16(&fl, &fld, pold, dl, pl, cl, m);
23         prefun_16(&fr, &frd, pold, dr, pr, cr, m);
24         *p = _mm512_mask_sub_ps(*p, m, pold,
25                                 _mm512_mask_div_ps(z, m,
26                                                    ADD(ADD(fl, fr), udiff),
27                                                    ADD(fld, frd))));
28         change = ABS(_mm512_mask_div_ps(z, m, SUB(*p, pold),
29                                         ADD(*p, pold)));
30         cond_break = _mm512_mask_cmp_ps_mask(m, change,
31                                              tolpre2, _MM_CMPINT_LE);
32         m &= ~cond_break;
33         cond_neg = _mm512_mask_cmp_ps_mask(m, *p, z, _MM_CMPINT_LT);
34         *p = _mm512_mask_mov_ps(*p, cond_neg, tolpre);
35         pold = _mm512_mask_mov_ps(pold, m, *p);
36     }
37
38     // Check for divergence.
39     if (iter > nriter)
40     {
41         cout << "divergence in Newton-Raphson iteration" << endl;
42         exit(1);
43     }
44
45     *u = MUL(SET1(0.5), ADD(ADD(ul, ur), SUB(fr, fl)));
46 }
```

Рисунок 7. Векторизованная версия функции `starpu`

В строке 18 видна изначальная инициализация полной маски выполнения векторизованных итераций цикла. По мере работы цикла маска истощается (строка 32), и при полном ее обнулении цикл завершает работу.

Стоит отметить, что векторизация цикла с неизвестным числом итераций может быть довольно опасной, так как количество итераций векторизованного цикла равно максимуму из количеств итераций циклов из 16 объединяемых вызовов оригинальной невекторизованной функции. При большой разнице в количестве итераций оригинального кода возникает падение эффективности, связанное с низкой плотностью масок исполняемых инструкций, как это показано в работе [**16**].

6. Анализ результатов

Перед началом оптимизации программного кода римановского решателя был выполнен сбор профиля исполнения, который показал, что время исполнения распределено между отдельными функциями согласно диаграмме, представленной на рисунке 8.

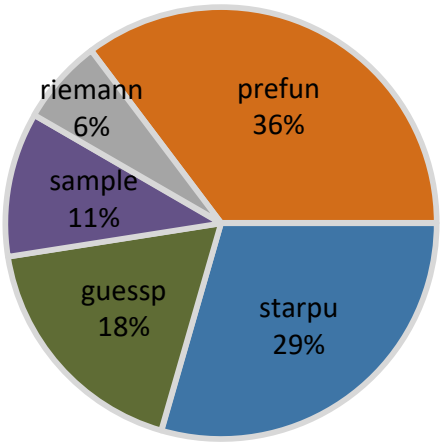

Рисунок 8. Распределение времени выполнения римановского решателя между отдельными функциями

Для сбора профиля исполнения исходная программа была скомпилирована с запретом оптимизации подстановки тела функции в точку вызова (inline). Таким образом, на диаграмме отмечено чистое время выполнения функций без учета вложенных вызовов. Из диаграммы видно, что наибольшая доля времени исполнения приходится на функцию `prefun` (36%), содержащую простой программный контекст с одним условием. Также значительная часть времени исполнения приходится на функцию `starpu` (29%), содержащую гнездо циклов с неизвестным числом итераций. Оставшееся время делится между тремя другими функциями `guessp` (18%), `sample` (11%), `riemann` (6%).

Описанные в статье подходы к векторизации функций римановского решателя были реализованы на языке программирования C с использо-

ванием функций-инстринсиков и опробованы на микропроцессорах Intel Xeon Phi 7290, входящих в состав вычислительного сегмента knl суперкомпьютера МВС-10П, находящегося в МСЦ РАН.

Тестирование производительности выполнялось на массивах входных данных, собранных при решении стандартных тестовых задач: задача Сода, задача Лакса, задача о слабой ударной волне, задача Эйнфельдта, задача Вудворда-Колелла, задача Шу-Ошера и других [18].

На диаграмме рисунка 9 показан эффект от применения различных оптимизаций к каждой из рассматриваемых функций, а также суммарное ускорение, полученное вследствие векторизации.

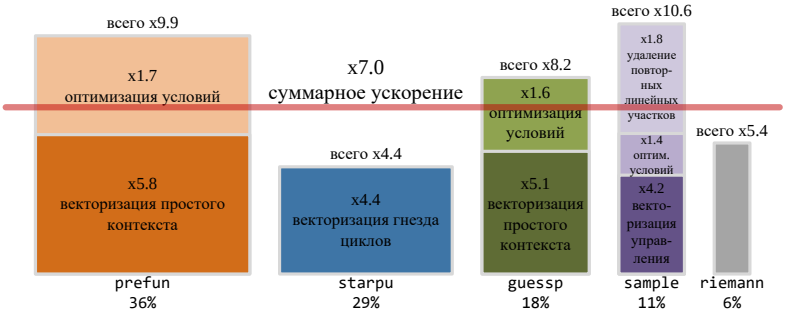

Рисунок 9. Диаграмма ускорения отдельных функций и суммарного ускорения римановского решателя

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

Отдельно на диаграмме выделен эффект от применения оптимизации замены переменных, позволившей в 1.8 раз ускорить функцию `sample` путем слияния двух поддеревьев графа потока управления (то

есть было выполнено удаление дублирующих линейных участков).

В результате применения всех описанных оптимизаций удалось достичь ускорения отдельных участков выполнения программы в 10 и более раз, а суммарное ускорение всего римановского решателя составило 7 раз (отмечено красной линией на рисунке 9).

7. Близкие работы

Приведем краткое сравнение полученных в данной статье результатов с результатами других работ, направленных на повышение эффективности работы римановских решателей на параллельных архитектурах. При этом отметим, что хотя в работе и не рассматривалось распараллеливание базирующихся на римановском решателе численных методов с помощью MPI, OpenMP, а также с помощью графических ускорителей, было бы неправильно не упомянуть работы, касающиеся данных аспектов распараллеливания.

В работе [4] рассматривается алгоритм численного решения уравнений магнитной гидродинамики, базирующийся на приближенном римановском решателе Рое. При этом вычисления проводятся на блочно-структурированной расчетной сетке, и для распараллеливания с помощью MPI выполняется декомпозиция блоков сетки (разрезание вдоль одного или нескольких направлений). Приводятся результаты измерения показателя слабой масштабируемости при использовании до 16 процессоров. Данный показатель оказывается близким к единице и даже в некоторых случаях превышает ее. Такое явление, называемое сверхлинейной масштабируемостью, можно также встретить и в других работах, посвященных распараллеливанию численных методов газовой динамики, например в [19].

Работа [5] посвящена распараллеливанию численных методов, базирующихся на различных приближенных римановских решателях, для выполнения на графических ускорителях NVIDIA Tesla C2050. При этом кроме решателя Рое используются также решатели HLLE и HLLC [20]. Продемонстрированы методы, позволяющие достигнуть ускорения на графическом ускорителе по сравнению с процессором Intel Xeon E5530 в 101 раз при использовании равномерной сетки. Также приведены показатели слабой и сильной масштабируемости распараллеленного приложения при использовании 32 GPU, они составили 98.8% и 85.0% соответственно.

В работе [6] приведены результаты применения инструмента PetClaw (Python-оболочка, интегрирующая вычислительное ядро

Clawpack, написанное на Fortran, и библиотеку PETSc в единое параллельное приложение) для распараллеливания гидродинамических расчетов с помощью MPI на большое количество ядер, вплоть до 16 тысяч. При этом продемонстрирован близкий к единице показатель эффективности распараллеливания.

В работах [7], [8] описаны подходы к распараллеливанию вычислений при моделировании астрофизических течений на гибридных суперкомпьютерах, оснащенных ускорителями Intel Xeon Phi. В частности было продемонстрировано 134-кратное ускорение расчетных кодов на одном ускорителе при использовании многопоточного распараллеливания, а также 75% масштабируемость при использовании до 224 ускорителей. В статье отмечено, что для дальнейшего ускорения приложения нужна векторизация вычислительного ядра и приводится прогноз по достижению 80% пиковой производительности при условии применения векторизации.

Наиболее близкими к рассматриваемой работе являются [9], [10], в которых особое внимание уделяется векторизации римановского решателя для численного решения уравнений мелкой воды. В работе [9] приведены результаты по ускорению 6-7 раз при векторизации римановского решателя на данный одинарной точности на ускорителях Intel Xeon Phi KNC. В работе [10] метод векторизации был доработан и эффект составил уже 2.4-6.5 на операциях с двойной точностью на микропроцессорах Intel Xeon Phi KNL. Однако в данных работах используется специально разработанный приближенный римановский решатель для уравнений мелкой воды [21]. Реализация этого решателя содержит более простой вычислительный контекст по сравнению с точным решателем, векторизация которого рассматривается в текущей работе. Приближенный решатель из [21] содержит только арифметические операции и поддается автоматической векторизации после специальной подготовки входных параметров (группировка нескольких вызовов решателя в один вызов с векторными параметрами).

Проведенное сравнение данной работы с работами схожими по тематике позволяет заключить, что полученное с помощью набора инструкций AVX-512 значение ускорения точного римановского решателя, равное 7, является хорошим результатом.

Заключение

В работе были рассмотрены подходы к векторизации сложного программного контекста с помощью использования набора инструкций

AVX-512. Данный набор инструкций появился в микропроцессорах Intel начиная с ускорителей Intel Xeon Phi KNC, а затем вошел в микропроцессоры Intel Xeon Phi KNL, Intel Xeon Skylake и далее. Инструкции AVX-512 поддерживают предикатную обработку элементов данных, что позволяет векторизовать с помощью них сложный разветвленный программный код.

В качестве целевой задачи был использован точный римановский решатель, так как он обладает компактной реализацией но в то же время содержит особенности, препятствующие автоматической векторизации средствами компилятора (вызовы функций, сложное управление, вложенные циклы). Для векторизации использовался подход, при котором несколько последовательных вызовов функции решателя заменялись на один вызов с векторными параметрами (и с соответствующими изменениями тела функции). Это позволило выполнять одновременно на одном процессорном ядре несколько экземпляров задачи (количество экземпляров равняется ширине вектора, в данном случае 16).

Были проанализированы составляющие части римановского решателя, выделены их особенности и для каждой из них предложен способ эффективной векторизации. В результате векторизации было достигнуто ускорение более 10 раз на отдельных участках решателя, а итоговое ускорение составило 7 раз на данных с одинарной точностью.

Разработанные методы векторизации сложного программного контекста могут быть использованы для оптимизации других вычислительных задач. В частности в настоящее время коллективом авторов данной статьи разрабатывается библиотека, направленная на векторизацию произвольных плоских циклов, то есть таких циклов, в которых отсутствуют межитерационные зависимости. При этом тело цикла может включать такие элементы как сложное управление, гнезда циклов, вызовы чистых функций, операторы goto и другие. Реализация такого инструмента позволит существенно повысить производительность расчетных кодов, автоматическая векторизация которых компилятором невозможна.

Список литературы

[1] А. Г. Куликовский, Н. В. Погорелов, А. Ю. Семенов. *Математические вопросы численного решения гиперболических систем уравнений*, Физматлит, М., 2001, 608 с. ↑60

[2] В. Е. Борисов, Ю. Г. Рыков. «Точный римановский солвер в алгоритмах

решения задач многокомпонентной газовой динамики», *Препринты ИПМ им. М. В. Келдыша*, 2018, 096, 28 с. ↑60

[3] С. К. Годунов, А. В. Забродин, М. Я. Иванов, А. Н. Крайко, Г. П. Прокопов. *Численное решение многомерных задач газовой динамики*, Наука, М., 1976, 400 с. ↑60

[4] U. Shumlak, B. Udrea. *An approximate Riemann solver for MHD computations on parallel architectures*, AIAA-2001-2591, 15th AIAA Computational Fluid Dynamics Conference (11 June 2001–14 June 2001, Anaheim, CA, USA), 2001, 8 pp. ↑60,75

[5] H.-Y. Schive, U.-H. Zhang, T. Chiueh. "Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in AMR", *International Journal of High Performance Computing Applications*, **26**:4 (2011), pp. 367–377. ↑60,75

[6] K. T. Mandly, A. Alghamdi, A. Ahmadia, D. I. Ketcheson, W. Scullin. "Using Python to construct a scalable parallel nonlinear wave solver", *Proceedings of the 10th Python in Science Conference*, SCIPY 2011 (11–16 Jule 2011, Austin, Texas, USA), 2011, pp. 61–66. ↑60,75

[7] И. М. Куликов, И. Г. Черных, Э. И. Воробьев, А. В. Снытников, Д. В. Винс и др. «Численное гидродинамическое моделирование астрофизических течений на гибридных СуперЭВМ, оснащенных ускорителями Intel Xeon Phi», *Вестн. ЮУрГУ. Сер. Выч. матем. информ.*, **5**:4 (2016), с. 77–97. ↑61,76

[8] I. Kulikov, I. Chernykh, V. Vshivkov, V. Prigarin, V. Mironov, A. Tatukov. "The parallel hydrodynamic code for astrophysical flow with stellar equation of state", *RuSCDays 2018: Supercomputing*, Communications in Computer and Information Science, vol. **965**, Springer, Cham, 2018, pp. 414–426. ↑61,76

[9] M. Bader, A. Breuer, W. Höltz, S. Rettenberger. "Vectorization of an augmented Riemann solver for the shallow water equations", *Proceedings of the 2014 International Conference on High Performance Computing and Simulation*, HPCS 2014 (21–25 July 2014, Bologna, Italy), 2014, pp. 193–201. ↑61,76

[10] C. R. Ferreira, K. T. Mandli, M. Bader. "Vectorization of Riemann solvers for the single- ans multi-layer shallow water equations", *Proceedings of the 2018 International Conference on High Performance Computing and Simulation*, HPCS 2018 (16–20 July 2018, Orleans, France), 2018, pp. 415-422. ↑61,76

[11] В. Ю. Волконский, С. К. Окунев. «Предикатное представление как основа оптимизации программы для архитектур с явно выраженной параллельностью», *Информационные технологии*, 2003, №4, с. 36–45. ↑62

[12] А. К. Ким, В. И. Перекатов, С. Г. Ермаков. *Микропроцессоры и вычислительные комплексы семейства «Эльбрус»*, Питер, СПб., 2013, 273 с. ↑62

[13] Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/. ↑63

[14] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, 2nd Edition, Springer, Berlin–Heidelberg, 1999, 645 pp. ↑63

[15] А. А. Рыбаков. «Оптимизация задачи об определении конфликтов с опасными зонами движения летательных аппаратов для выполнения на Intel Xeon Phi», *Программные продукты и системы*, **30**:3 (2017), с. 524–528. doi ↑69

[16] А. А. Рыбаков, П. Н. Телегин, Б. М. Шабанов. «Проблемы векторизации гнезд циклов с использованием инструкций AVX-512», *Программные продукты, системы и алгоритмы*, 2018, №3, 11 с. doi ↑71,72

[17] O. Krzikalla, F. Wende, M. Hohnerbach. "Dynamic SIMD vector lane scheduling", *ISC High Performance 2016: High Performance Computing*, Lect. Notes Comput. Sci., vol. **9945**, Springer, Cham, 2016, pp. 354–365. doi ↑71

[18] П. В. Булат, К. Н. Волков. «Одномерные задачи газовой динамики и их решение при помощи разностных схем высокой разрешающей способности», *Научно-технический вестник информационных технологий, механики и оптики*, **15**:4 (2015), с. 731–740. doi ↑74

[19] Л. А. Бендерский, Д. А. Любимов, А. А. Рыбаков. «Анализ эффективности масштабирования при расчетах высокоскоростных турбулентных течений на суперкомпьютере RANS/ILES методом высокого разрешения», *Труды НИИСИ РАН*, **7**:4 (2017), с. 32–40. URL doi ↑75

[20] C. Kong. *Comparison of Approximate Riemann Solvers*, A dissertation of the degree of Master of Science in Mathematical and Numerical Modeling of the Atmoshere and Oceans, Department of Mathematics, University of Reading, 2011. ↑75

[21] D. L. George. "Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation", *Journal of Computational Physics*, **227**:6 (2008), pp. 3089–3113. doi ↑76

Поступила в редакцию 19.02.2019
Переработана 10.09.2019
Опубликована 30.09.2019

Рекомендовал к публикации *д.ф.-м.н. С. М. Абрамов*

Пример ссылки на эту публикацию:

А. А. Рыбаков, С. С. Шумилин. «Векторизация римановского решателя с использованием набора инструкций AVX-512». *Программные системы: теория и приложения*, 2019, **10**:3(42), с. 59–80.
10.25209/2079-3316-2019-10-3-59-80
http://psta.psiras.ru/read/psta2019_3_59-80.pdf

Эта же статья по-английски: 10.25209/2079-3316-2019-10-3-41-58

Об авторах:

Алексей Анатольевич Рыбаков

Рыбаков Алексей Анатольевич – к ф.-м. н., внс МСЦ РАН – филиала ФГУ ФНЦ НИИСИ РАН. Области научных интересов – математическое моделирование задач газовой динамики с использованием суперкомпьютеров, методы построения и управления расчетными сетками, дискретная математика, теория графов, модели случайных графов, параллельное программирование, функциональное программирование.

0000-0002-9755-8830
e-mail: rybakov@jscc.ru

Сергей Сергеевич Шумилин

Шумилин Сергей Сергеевич – ведущий инженер МСЦ РАН – филиала ФГУ ФНЦ НИИСИ РАН. Области научных интересов – машинное обучение, анализ данных, алгоритмы, параллельное программирование.

0000-0002-3953-7054
e-mail: shumilin@jscc.ru

Sample citation of this publication:

Alexey A. Rybakov, Sergey S. Shumilin. “Vectorization of the Riemann solver using the AVX-512 instruction set”. *Program Systems: Theory and Applications*, 2019, **10**:3(42), pp. 59–80. (*In Russian*).
10.25209/2079-3316-2019-10-3-59-80
http://psta.psiras.ru/read/psta2019_3_59-80.pdf

The same article in English: 10.25209/2079-3316-2019-10-3-41-58