

Векторизация сильно разветвленного управления с помощью инструкций AVX-512

А.А. Рыбаков¹, С.С. Шумилин²

Межведомственный суперкомпьютерный центр РАН - филиал ФГУ ФНЦ НИИСИ РАН, Москва, Россия,

E-mails: ¹rybakov.aax@gmail.com, ²shumilin@jssc.ru

Аннотация: Векторизация вычислений является важнейшей оптимизацией, с помощью которой может быть достигнуто кратное ускорение расчетных кодов. По мере развития современных микропроцессоров длина вектора в векторных операциях постоянно увеличивается. В современных линейках микропроцессоров Intel x86 (Xeon Phi KNL и Xeon Skylake) длина вектора уже достигает 512 бит. Однако зачастую программный код написан таким образом, что автоматическое применение векторизации невозможно. Причинами отказа от применения векторизации может стать наличие зависимостей между операциями в коде, вызовы функций или непостоянное количество итераций циклов в векторизуемых гнездах. В статье рассматривается актуальная проблема векторизации циклов, тела которых содержат сложное управление. Предложены два подхода к применению векторизации для циклов с условием и рассмотрено применение данных подходов на практической задаче с сильно разветвленным управлением.

Ключевые слова: оптимизация, векторизация цикла, предикатный режим исполнения, AVX-512, функции-интринсики, сильно разветвленное управление.

Введение

В настоящее время одновременно с развитием вычислительных систем и возрастанием их вычислительной мощности возникают вопросы об эффективности их применения. Использование векторных инструкций является наиболее низкоуровневым направлением создания высокопроизводительного кода. При выполнении векторизации программного кода необходимо учитывать факторы, которые могут негативно повлиять на итоговую производительность.

Самый простой вид циклов для векторизации – плоские циклы. Это циклы без побочных эффектов и зависимостей между итерациями, в которых эти итерации могут выполняться параллельно. Такие циклы хорошо поддаются векторизации, и при выполнении необходимых условий по выравниванию компилятор применяет векторизацию.

При добавлении в плоский цикл команд ветвления (операторы if, switch) компиляторicc также без труда справляется с векторизацией, используя масочные операции и операции слияния по условию. Сложнее дело обстоит, если внутри векторизуемого цикла появляется другой

цикл с непостоянным количеством итераций. Такие конструкции не поддаются автоматической векторизации. Однако в случае независимости итераций вложенного цикла друг от друга охватывающий цикл все же можно векторизовать вручную, заменив все операции на векторные аналоги, использующие в качестве операнда вектор предикатов продолжения выполнения каждой из объединенных итерации исходного вложенного цикла [1].

В случае наличия в векторизуемом цикле маловероятных ветвей исполнения можно выполнить расщепление цикла по условию, получив два цикла, в первом из которых будет сохраняться только условие ухода на маловероятную ветку, во второй попадет ее реализация. После преобразования первый цикл, избавившись от маловероятного кода, может быть эффективно векторизован [2]. Другие эквивалентные преобразования циклов, в результате которых цикл может быть приведен к виду, пригодному для автоматической векторизации, можно найти в фундаментальных трудах [3, 4].

В данной статье рассматривается проблема векторизации кода для процессоров с поддержкой набора инструкций AVX-512 применительно к

циклам, тела которых содержат сложное управление, и приводится описание двух различных подходов к векторизации циклов, содержащих условие. Также описано объединение данных подходов в комбинированный метод векторизации цикла со сложным управлением и рассмотрено применение этого метода к практической задаче.

Особенности инструкций AVX-512

Множество инструкций AVX-512 представляет собой 512-битное расширение 256-битных инструкций AVX архитектуры Intel x86 [5], которое поддерживается в семействах микропроцессоров Intel Xeon Phi Knights Landing (KNL) [6] и Intel Xeon Skylake.

В микропроцессорах KNL поддерживаются следующие подмножества инструкций AVX-512: AVX-512F – основной набор векторных инструкций с поддержкой маскирования, AVX-512PF – инструкции предварительной подкачки данных из памяти, AVX-512ER – команды для вычисления экспоненты и обратных значений, AVX-512CD – инструкции для определения совпадения элементов вектора друг с другом, предназначенные для динамического определения конфликтов по доступу в память.

Для поддержания выборочного применения операций над упакованными данными к конкретным элементам аргументов-векторов используются маски. Большинство инструкций из набора AVX-512 могут использовать специальные регистры масок (k0 - k7).

Длина каждой маски составляет 64 бита, она может быть использована для маскирования векторов любых типов, содержащих от 8 до 64 элементов. Маски k0 - k7 используются в командах для осуществления условной операции над элементами упакованных данных (если бит маски выставлен в 1, то результат операции записывается в соответствующий элемент вектора назначения), для слияния элементов данных в регистр назначения, для выборочного чтения и записи в память

элементов векторов и для аккумулирования результатов логических операций над элементами векторов.

Операции AVX-512 можно условно разделить на несколько групп согласно количеству и виду их аргументов и результата. Упакованные операции с одним операндом zmm (512-битный вектор) и одним результатом zmm (извлечение корня, округление, операции сдвигов и другие), упакованные операции с двумя операндами zmm и одним результатом zmm (поэлементные арифметические операции, операции сдвига на переменное количество разрядов и другие), упакованные операции с двумя операндами zmm и результатом маской (поэлементное сравнение двух векторов), операции конвертации, предназначенные для преобразования элементов вектора из одного формата в другой (cvt, pack), упакованные комбинированные операции (поэлементное вычисление значений вида $\pm a \cdot b \pm c$), операции перестановок, операции пересылок (в частности операции пересылки элементов данных расположенных с произвольными смещениями от заданного базового адреса в памяти, а также операции пересылки с дублированием элементов), операции предварительной подкачки данных, и другие операции с более сложной логикой, среди которых можно отметить определение класса вещественного числа, реализацию логических функций от трех аргументов, операции определения конфликтов и другие.

Для упрощения векторизации исходного кода для компилятора icc разработаны специальные функции-интринсики [7], 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, например для операции сложения всех элементов вектора (группа функций-интринсиков `reduce`). Из множества интринсиков можно выделить следующие группы функций, схожие по структуре. Функции `swizzle`, `shuffle` и

permute осуществляют перестановку элементов вектора и раскрываются в последовательность операций, в которой присутствует операция shuf и пересылка по маске. Для большего числа операций AVX-512 реализованы соответствующие инстринсики, раскрывающиеся в одну конкретную операцию. Некоторые инстринсики, особенно предназначенные для выполнения упакованных трансцендентных операций, раскрываются просто в вызов библиотечной функции (например _mm512_log_ps, _mm512_hypot_ps, а также тригонометрические функции).

Исследование эффективности векторизации цикла с одним условием

Рассмотрим простой цикл for с счетчиком, принимающим значения от 0 до $n - 1$. Тело цикла состоит из одного условного оператора if и двух линейных участков block 1 и block 2, которые выполняются в зависимости от данного условия. Будем считать, что цикл не содержит межитерационных зависимостей, все итерации цикла могут выполняться независимо друг от друга. Как вычисление условия, так и выполнение всех операций в блоках block 1 и block 2 зависят только от значения i на данной итерации цикла, возможны обращения в память только вида $a[i]$.

```
for ( $i = 0$ ;  $i < n$ ;  $i++$ )
{
    if (<cond( $i$ )>
    {
        <block 1> //  $t_1$ ,  $p_1$ 
    }
    else
    {
        <block 2> //  $t_2$ ,  $p_2$ 
    }
}
```

Пусть длина линейного участка block 1 равна t_1 , а вероятность его выполнения – p_1 . Аналогично длина линейного участка block 2 равняется t_2 , а вероятность его выполнения – p_2 . Единицы измерения длины участка условны, будем

понимать под ними количество операций в результирующем коде без учета задержек между ними. В нашем случае длины участком обозначены через t_1 и t_2 , так как длина линейного участка пропорциональна времени его исполнения. В дальнейшем условимся считать длину участка и время его исполнения тождественными понятиями. Время исполнения одной итерации цикла совпадает с математическим ожиданием длины исполненного линейного участка, тогда общее время выполнения цикла равно

$$T^{orig} = (\delta + p_1 t_1 + p_2 t_2) n,$$

где через δ обозначено время вычисления условия, и им, вообще говоря можно, пренебречь.

Рассмотрим возможности по векторизации данного цикла. Одной из важнейших особенностей некоторых микропроцессорных архитектур является наличие предикатного (условного) режима исполнения отдельных операций. Особенно это характерно для архитектур с явно выраженной параллельностью [8]. Наличие предикатного режима исполнения операций позволяет избежать создания избыточных операций передачи управления путем планирования в одном линейном участке команд под различными предикатами. Предикатных режим исполнения поддержан в таких архитектурах, как ARM [9] или «Эльбрус» [10]. В архитектуре x86 предикатный режим поддержан частично, в ней присутствуют специальные команды условной пересылки (CMOVcc, conditional move). Так как в языке программирования С отсутствуют явные синтаксические конструкции для применения предикатного режима исполнения, то необходима поддержка со стороны компилятора, способного создавать предикатный код. Примерами таких оптимизаций могут послужить преобразование if nodes conversion для архитектуры «Эльбрус» [11] или оптимизация условных пересылок для архитектур с поддержкой таких операций. В предикатной форме исходный цикл может быть переписан в следующем виде:

```
for ( $i = 0$ ;  $i < n$ ;  $i++$ )
{
    p = <cond( $i$ )>;
```

```

<block 1> // p
<block 2> // ~p
}

```

В данном случае под обозначением $\sim p$ понимается предикат, обратный предикату p . В результате данного преобразования тело цикла содержит один линейный участок, и общее время выполнения цикла равно

$$T_1^{pred} = (\delta + t_1 + t_2)n.$$

Набор инструкций AVX-512 обладает важным достоинством, заключающимся в наличии масочных аргументов в подавляющем большинстве векторных операций. Использование масок позволяет производить вычисления выборочно, над отдельными элементами векторов. При использовании векторизации с длиной вектора v суммарное время исполнения векторизованного кода будет равно

$$T_1^{vect} = (\delta + t_1 + t_2)nv^{-1},$$

при этом v равняется 8 в случае использования вещественных вычислений с типом `double`, v равняется 16 при использовании вещественных вычислений с типом `float` (именно данный случай мы будем в дальнейшем анализировать).

Конечно при оценке эффективности векторизации необходимо учитывать показатели латентности отдельных операций (у векторных инструкций они выше, чем у скалярных), это имеет важное значение при наличии длинных критических путей исполнения в программном коде.

Также важно учитывать пропускную способность для этих операций (количество операций данного вида, которые могут быть исполнены за один такт).

Некоторые операции AVX-512 имеют крайне низкую пропускную способность (например, операции деления, а также инструкции `gather/scatter`), и их присутствие в коде может послужить причиной возникновения узких мест и полностью нивелировать всю выгоду от оптимизации.

В данной статье мы не будем учитывать эти факторы, ограничившись простым числом операций в качестве времени выполнения кода (в том случае, когда код состоит из простых арифметических операций и не содержит длинных критических путей исполнения, это предположение можно считать верным).

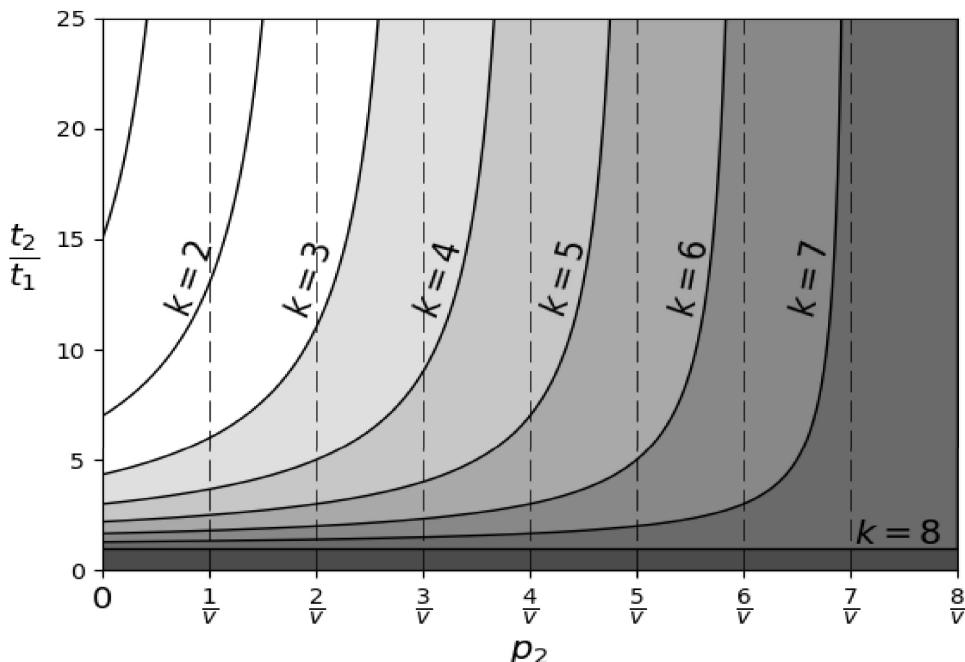


Рис. 1. Оценка эффективности векторизации при слиянии ветвей исполнения в теле цикла.

Для анализа эффекта от применения векторизации рассмотрим условие, при котором в наших предположениях эффективность применения векторизации не превышает k раз, то есть векторизованный код быстрее оригинального не более, чем в k раз (величину δ в данном случае опустим):

$$T^{orig} \leq k T_1^{vect}$$

$$p_1 t_1 + p_2 t_2 \leq \frac{(t_1 + t_2)k}{v}$$

Введем обозначение $q = k/v$, тогда последнее неравенство может быть переписано в виде

$$p_1 t_1 + p_2 t_2 \leq (t_1 + t_2)q,$$

что после преобразования может быть переписано в виде

$$t_1(p_1 - q) + t_2(p_2 - q) \leq 0$$

Для того, чтобы это условие выполнялось, необходимо, чтобы вероятность хотя бы одного из линейных участков block 1 или block 2 была достаточно мала. Пусть маловероятной веткой является block 2, тогда имеем $p_2 \leq q$ (в данном случае q выступает в роли верхней границы маловероятной ветки, при которой эффективность векторизованного кода мала). Вычислим, во сколько раз длина маловероятного линейного участка должна превышать длину основной ветки, чтобы положительный эффект от применения векторизации пропал.

$$d_1(p_2) = \frac{t_2}{t_1} \geq \frac{p_1 - q}{q - p_2} = \frac{1 - p_2 - q}{q - p_2} = \frac{(1 - q) - p_2}{q - p_2}$$

Будем рассматривать значения k от 1 до 16 (значение v у нас равняется 16) при $p_2 \leq 0.5$.

Проанализируем графики, которые приведены на рис. 1. Для $k < 8$ график зависимости $d_1(p_2)$ представляет собой возрастающую ветвь гиперболы. При увеличении значения p_2 величина d_1 значительно возрастает, то есть чем выше вероятность исполнения маловероятного участка, тем длиннее он должен быть для того, чтобы применение векторизации оказалось неэффективно, а начиная со значения $p_2 = q$ векторизованный вариант более эффективен при любом соотношении длин линейных участков block 1 и block 2.

При $k = 8$ график представляет собой константу, равную единице. Это означает, что при условии $t_2 > t_1$ невозможно получить теоретическое ускорение векторизованного кода в 8 раз. Для оставшихся случаев $k > 8$ график также представляет собой ветку гиперболы, однако загибающуюся вниз и уходящую в область отрицательных значений. Это означает, что при использовании векторизации предикатного кода, полученного вследствие слияния отдельных ветвей исполнения, сложно или невозможно достигнуть ускорения в k раз.

Теперь рассмотрим другой подход к векторизации цикла, основанный на вынос маловероятного условия в отдельный цикл. Снова будем считать, что маловероятной веткой является линейный участок block 2. Модифицируем цикл таким образом, чтобы полностью избавиться от выполнения маловероятного кода block 2 в цикле. Вместо этого заведем временный массив с булевыми переменными, в котором будем сохранять признаки необходимости исполнения маловероятного кода. Обработку же маловероятного кода вынесем в отдельный цикл.

```
for (i = 0; i < n; i++)
{
    tmp[i] = <cond(i)>;
    if (tmp[i])
    {
        <block 1> // t1, p1
    }
}

for (i = 0; i < n; i++)
{
    if (!tmp[i])
    {
        <block 2> // t2, p2
    }
}
```

Такой подход оправдывается тем, что после преобразования первый цикл содержит только вероятную ветку и небольшой объем вспомогательных вычислений, а второй цикл содержит только маловероятные вычисления, которыми можно пренебречь. Общее время работы данного модифицированного кода равно

$$T_2^{\text{mod}} = (\delta + \alpha + p_1 t_1) n + (\beta + p_2 t_2) n.$$

Запишем первый цикл в предикатной форме. При этом второй цикл не требуется векторизовать, поэтому переводить его в предикатную форму не нужно, поэтому приведем представление только первого цикла, который теперь выглядит следующим образом:

```
for (i = 0; i < n; i++)
{
    p = <cond(i)>;
    tmp[i] = p;
    <block 1> // p
}
```

Общее время работы данного участка кода в предикатной форме равно

$$T_2^{\text{pred}} = (\delta + \alpha + t_1) n + (\beta + p_2 t_2) n,$$

однако теперь первый цикл поддается векторизации, что в итоге дает время исполнения векторизованного кода, равное

$$T_2^{\text{vect}} = (\delta + \alpha + t_1) n v^{-1} + (\beta + p_2 t_2) n.$$

Аналогично предыдущему подходу, проанализируем ситуацию, когда векторизованный код быстрее оригинального не более, чем в k раз.

$$T^{\text{orig}} \leq k T_2^{\text{vect}}$$

$$p_1 t_1 + p_2 t_2 \leq k \left(\frac{t_1}{v} + p_2 t_2 \right)$$

$$t_1(p_1 - q) - t_2 p_2(k - 1) \leq 0$$

Проанализируем получившееся неравенство. В отличие от предыдущего подхода, в данном случае чем ниже вероятность маловероятной ветви, тем выгоднее применение векторизации. Получим аналогичную оценку на отношение длины маловероятной ветки к длине вероятной ветви.

$$d_2(p_2) = \frac{t_2}{t_1} \geq \frac{p_1 - q}{p_2(k - 1)} = \frac{(1 - q) - p_2}{p_2(k - 1)}$$

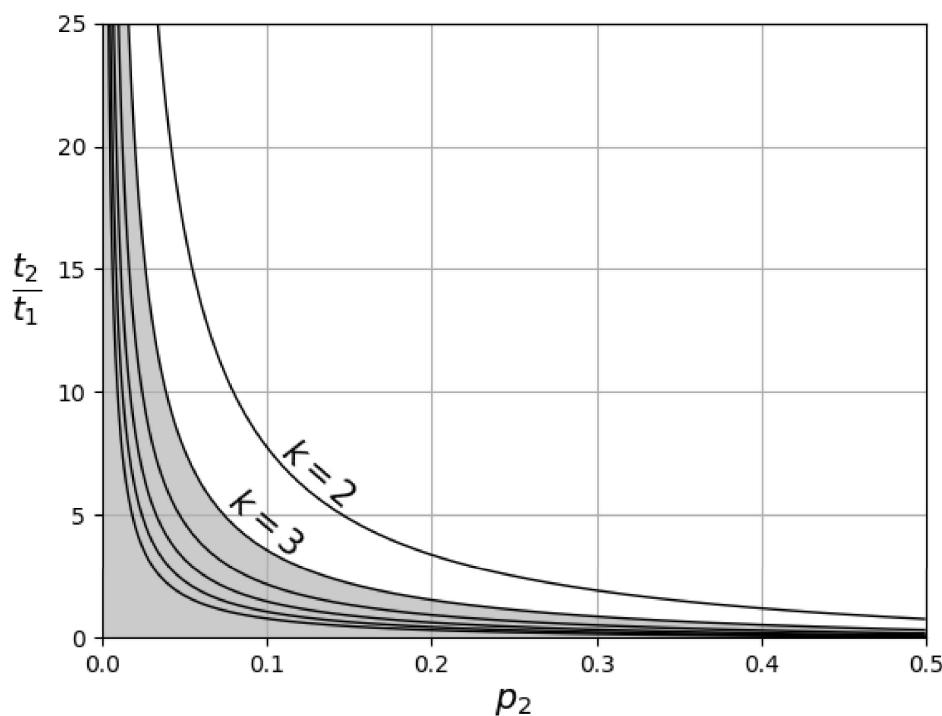


Рис. 2. Оценка эффективности векторизации при выносе маловероятной ветви исполнения из цикла.

Из графиков на рис. 2 видно, что применение выноса маловероятной ветви

исполнения из цикла выгодно при очень низких значениях p_2 .

Таким образом, описаны два подхода к векторизации циклов, состоящих из одного простого условия с двумя линейными участками: слияние двух ветвей исполнения и вынос маловероятной ветви из цикла. При $k < 8$ эффективность векторизации со слиянием двух ветвей повышается при росте вероятности маловероятной ветви, а эффективность векторизации с выносом маловероятной ветви из цикла наоборот понижается. То есть, в зависимости от вероятности маловероятной ветви исполнения нужно выбирать конкретный

подход к векторизации. Значение вероятности p_2 для выбора одного из двух подходов определяется из соотношения

$$\frac{(1-q) - p_2}{q - p_2} = \frac{(1-q) - p_2}{p_2(k-1)},$$

откуда находим $p_2 = v^{-1}$. На рис. 3 представлены комбинированные графики оценки эффективности векторизации цикла с одним условием с учетом выбора конкретного подхода к векторизации.

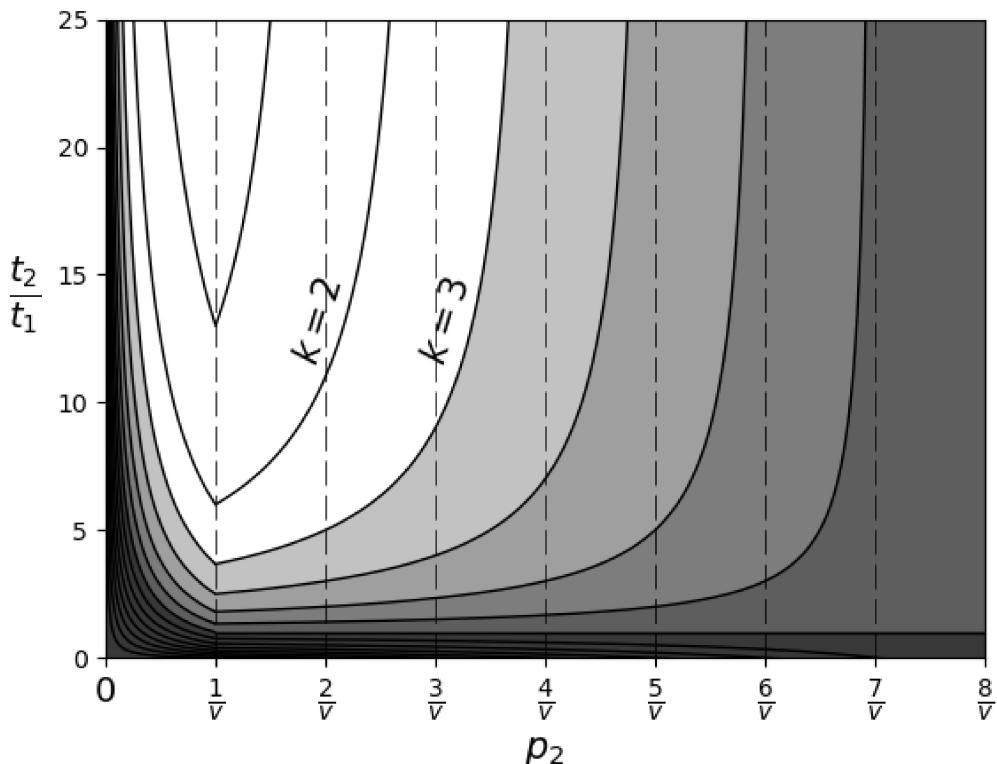


Рис. 3. Оценка эффективности векторизации цикла с одним условием с учетом выбора подхода к векторизации.

Практическое применение

Рассмотрим применение описанных выше методов векторизации кода со сложным управлением. Подходящий контекст для оптимизации будем искать в реализации точного решения задачи о распаде произвольного разрыва, которая является одной из классических фундаментальных задач математической физики [12].

Ввиду широкой известности программной реализации различных вариаций решения данной задачи не будем детально описывать ее, полную информацию касательно ее решения можно найти в [13], реализация на языке программирования FORTRAN находится в открытом доступе в сети Интернет [14].

Основная функция решения задачи о распаде произвольного разрыва имеет следующую сигнатуру:

```
void riemann
  (float dl, float ul, float pl,
   float dr, float ur, float pr,
   float &d, float &u, float &p)
```

Функция принимает на вход значение газодинамических параметров слева от разрыва (dl, ul, pl) и справа от него (dr, ur, pr) и определяет значения параметров в нулевой момент времени на самом разрыве (d, u, p). В процессе математического моделирования процессов газовой динамики данная задача решается множество раз на каждой грани каждой ячейки расчетной сетки, поэтому эффективная реализация данного кода имеет важное значение. Решение каждой отдельной задачи о распаде разрыва является независимым, решение всех таких задач может выполняться параллельно. На каждой временной итерации проведения расчетов вызовы функции `riemann` выполняются над отдельными элементами входных данных, объединенных в массивы. Условно это можно изобразить в следующем виде:

```
void riemann_multy
  (float *dl, float *ul,
   float *pl, float *dr,
   float *ur, float *pr,
   float *d, float *u,
   float *p)
{
  .....
  for (i = 0; i < count; i++)
  {
    riemann(dl[i], ul[i], pl[i],
            dr[i], ur[i], pr[i],
            d_, u_, p_);
    d[i] = d_;
    u[i] = u_;
    p[i] = p_;
  }
  .....
}
```

Для повышения эффективности целесообразно произвести inline-подстановку тела функции `riemann` в место вызова. Составной частью функции `riemann` является функция `sample`, которая на основании данных о газодинамических параметрах слева и справа от разрыва, а также вычисленных скорости и давлении газа в центральном регионе (star region),

получает характер и газодинамические характеристики на разрыве в нулевой момент времени. Функция `sample` содержит сильно разветвленное управление, что является подходящим контекстом для применения описанных в данной статье подходов. Для повышения эффективности применения оптимизации произведем расщепление основного цикла функции `riemann_multy` таким образом, чтобы один из результирующих циклов содержал только тело функции `sample`. Данный цикл и представляет интересующий нас контекст для применения оптимизации, в дальнейшем будем рассматривать только этот участок кода.

```
for (i = 0; i < count; i++)
{
  .....
  // should be inlined...
  sample(dl[i],ul[i],pl[i],cl[i],
         dr[i],ur[i],pr[i],cr[i],
         pm[i], um[i],
         d_, u_, p_);
  d[i] = d_;
  u[i] = u_;
  p[i] = p_;
  .....
}
```

С учетом того, что тело рассматриваемого цикла содержит около полутора сотен строк кода, мы не будем приводить текст данного цикла, а обозначим только его граф потока управления. Тело рассматриваемого цикла содержит 19 линейных участков и 9 условий (максимальная вложенность условий равна четырем). При этом все итерации данного цикла независимы и могут быть выполнены параллельно. Также отметим, что на итерации с номером i осуществляются обращения в память только в элементах массивов вида $a[i]$, из операций используются арифметические операции (сложение, вычитание, умножение, деление), сравнение, возведение в степень, извлечение квадратного корня, вычисление обратного значения. Все эти операции имеют векторные аналоги в множестве команд AVX-512, в тому же все эти векторные операции поддерживают предикатный режим исполнения.

Для определения лучшей стратегии векторизации для тела рассматриваемого цикла был собран профиль исполнения полученный на стандартных тестовых задачах: задача Сода, задача Лакса, задача о слабой ударной волне, задача Эйнфельдта, задача Вудвуда-Колелла, задача Шу-Ошера

[15, 16] и другие. На рис. 4 представлен граф потока управления (control flow graph, CFG) для тела рассматриваемого цикла. Узлами данного графа являются линейные участки, а ребрами – переходы между ними.

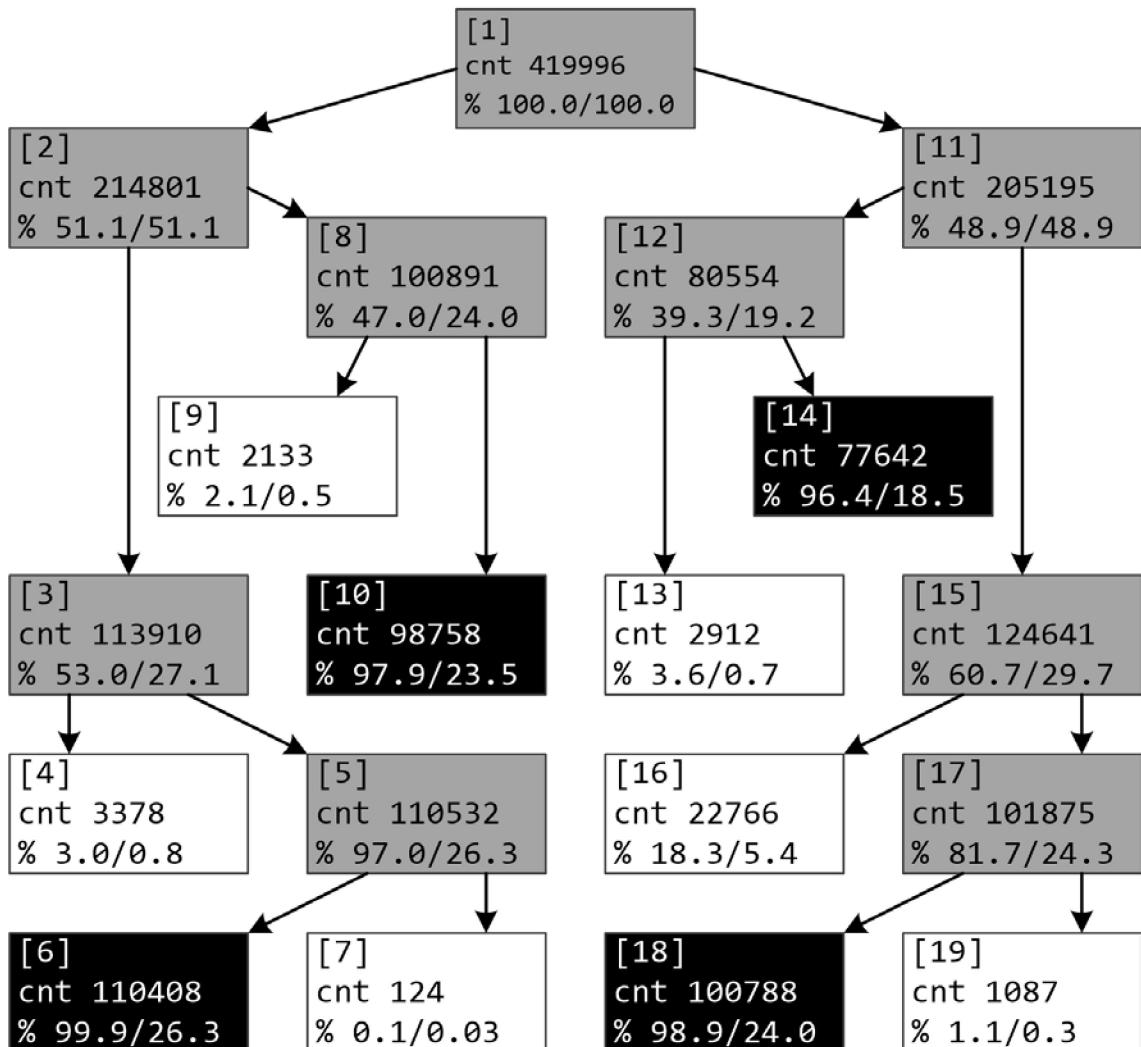


Рис. 4. Граф потока управления тела векторизуемого цикла со сложным управлением.

Для каждого узла CFG с рис. 4 приведены следующие его характеристики: номер узла (в квадратных скобках), счетчик или количество исполнений линейного участка (после слова cnt), локальная вероятность исполнения (отношение счетчика линейного участка к счетчику его непосредственного доминатора, умноженное на 100%), глобальная вероятность исполнения (отношение счетчика линейного участка к счетчику головы цикла, умноженное на 100%). Все узлы CFG окрашены в три цвета. Белым цветом

закрашены маловероятные участки кода, черным – наиболее вероятные листовые линейные участки, серым – все остальные линейные участки.

Код данного цикла был подготовлен для векторизации – была декларирована независимость всех массивов, участвующих в вычислениях (все массивы были переданы в функцию с модификатором `_restrict_`), было декларировано выравнивание расположения массивов в памяти на 64 байта. После подготовки программного кода

к векторизации от компилятора icc был получен отказ по эффективности.

Таким образом, компилятор смог построить векторизованный код, однако его теоретическая эффективность оказалась слишком низкой ввиду слишком большого количества ветвей исполнения.

Для поддержки ручной векторизации были предприняты следующие действия.

Прежде всего было замечено, что 4 линейных участка содержали определение газодинамических параметров d , u и p , которое можно было изменить на инициализацию с помощью выноса операций присваивания вверх по коду цикла. Таким образом, линейные участки с номерами 4, 9, 13, 16 были удалены, что привело к ускорению кода примерно на 10% (см. рис. 5).

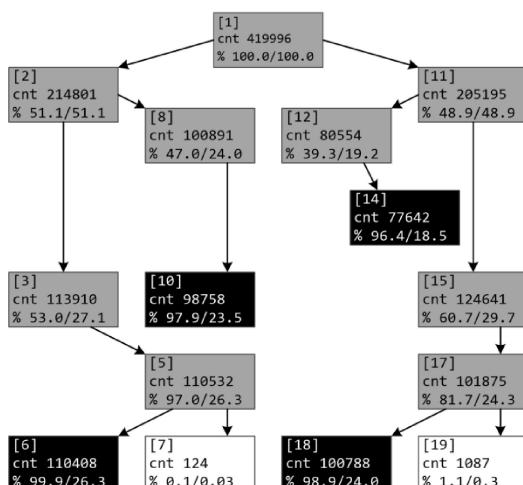


Рис. 5. Удаление линейных участков из тела цикла путем выноса операций присваивания вверх.

Далее было отмечено, что функция sample обрабатывает одинаковым образом правый и левый профиль распада разрыва с незначительными изменениями.

С помощью простой замены переменных удалось выполнить слияние кода для двух поддеревьев узла с номером 1 (поддеревья, опирающиеся на узлы с номерами 2 и 11).

Данное преобразование вдвое уменьшило объем расчетного кода и ожидали сократило время исполнения еще на 45% (см. рис. 6).

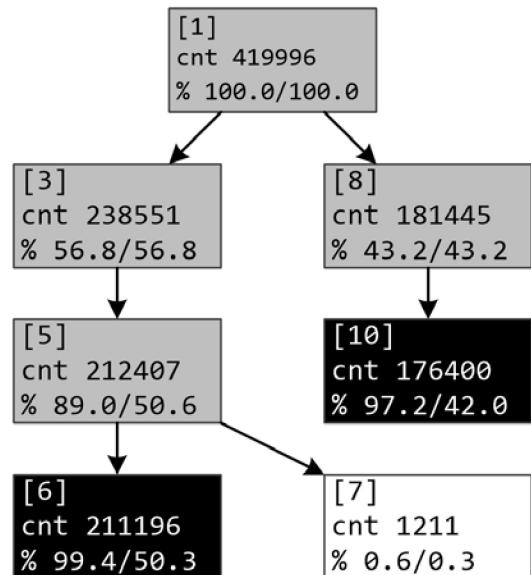


Рис. 6. Выполнение слияния симметричных линейных участков в теле цикла.

Однако даже после сокращения тела цикла до 7 линейных участков, связанных 4 условиями переходов автоматическая векторизация оказалась неэффективной, и компилятор выдал отказ в проведении данной оптимизации.

Для достижения положительного эффекта от применения векторизации следует воспользоваться описанными в предыдущем разделе подходами. Во-первых, обратим внимание на линейный участок с номером 7. Это линейный участок с крайне низким счетчиком, однако обладающий большой длиной, так как он содержит тяжелые операции (вещественное деление и возведение в степень). К данному линейному участку целесообразно применение выноса маловероятной ветки из цикла.

Так как тело цикла само по себе достаточно объемное, то мы будем применять не вынос маловероятной ветки, а локализацию этой ветки. Локализация маловероятной ветки выполняется следующим образом. Если в оригинальном варианте выполнение маловероятной ветки обслуживалось предикатом p , то в векторизованном варианте соответствующий векторный предикат в большинстве случаев нулевой, а значит перед выполнением векторизованного кода сначала следует проверить векторный предикат на наличие хотя бы одного ненулевого элемента. Такой подход выгоден только для маловероятных

веток, так как одной проверкой отсекается сразу 16 итераций тяжелого кода, который и не должен быть выполнен. К тому же локализация маловероятной ветки избавляет от необходимости содержания дополнительного временного массива булевых переменных.

После локализации маловероятной ветки остаются две равнозначные ветки с практической одинаковой вероятностью, к

```

1  for (int j = 0; j < TESTS_COUNT; j += 16)
2  {
3      // Load.
4      v_pm = LD(&pm[j]);
5      v_um = LD(&um[j]);
6      // d/u/p/c/ums
7      cond_um = _mm512_cmp_ps_mask(v_um, z, _MM_CMPINT_LT);
8      d = _mm512_mask_blend_ps(cond_um, LD(&d1[j]), LD(&dr[j]));
9      u = _mm512_mask_blend_ps(cond_um, LD(&ul[j]), LD(&ur[j]));
10     p = _mm512_mask_blend_ps(cond_um, LD(&pl[j]), LD(&pr[j]));
11     c = _mm512_mask_blend_ps(cond_um, LD(&cl[j]), LD(&cr[j]));
12     ums = LD(&um[j]);
13     u = _mm512_mask_sub_ps(u, cond_um, z, u);
14     ums = _mm512_mask_sub_ps(ums, cond_um, z, ums);
15     // Calculate main values.
16     pms = DIV(v_pm, p);
17     sh = SUB(u, c);
18     st = _mm512_fnmadd_ps(POW(pms, g1), c, ums);
19     s = _mm512_fnmadd_ps(c, SQRT(_mm512_fnmadd_ps(g2, pms, g1)), u);
20     // Conditions.
21     cond_pm = _mm512_cmp_ps_mask(v_pm, p, _MM_CMPINT_LE);
22     cond_sh = _mm512_mask_cmp_ps_mask(cond_pm, sh, z, _MM_CMPINT_LT);
23     cond_st = _mm512_mask_cmp_ps_mask(cond_sh, st, z, _MM_CMPINT_LT);
24     cond_s = _mm512_mask_cmp_ps_mask(~cond_pm, s, z, _MM_CMPINT_LT);
25     // Store.
26     d = _mm512_mask_mov_ps(d, cond_st, MUL(d, POW(pms, igama)));
27     d = _mm512_mask_mov_ps(d, cond_s, MUL(d, DIV(ADD(pms, g6), _mm512_fnmadd_ps(pms, g6, v1))));
28     u = _mm512_mask_mov_ps(u, cond_st | cond_s, ums);
29     p = _mm512_mask_mov_ps(p, cond_st | cond_s, v_pm);
30     // Low prob - ignore it.
31     cond_sh_st = cond_sh & ~cond_st;
32     if (cond_sh_st)
33     {
34         u = _mm512_mask_mov_ps(u, cond_sh_st, MUL(g5, _mm512_fnmadd_ps(g7, u, c)));
35         uc = DIV(u, c);
36         d = _mm512_mask_mov_ps(d, cond_sh_st, MUL(d, POW(uc, g4)));
37         p = _mm512_mask_mov_ps(p, cond_sh_st, MUL(p, POW(uc, g3)));
38     }
39     // Final store.
40     u = _mm512_mask_sub_ps(u, cond_um, z, u);
41     ST(&od[j], d);
42     ST(&ou[j], u);
43     ST(&op[j], p);
44 }
```

Рис. 7. Итоговый вариант векторизованного тела цикла для функции sample.

В строках 8-13 выполняется изначальная инициализация результатов с помощью операций blend. Далее в строках 16-19 вычисляются значения, вынесенные вверх из вероятных листовых линейных участков. Строки 21-24 нужны для вычисления условий, по которым

которым следует применить слияние в предикатный код, который поддается векторизации, что и представлено на рис. 7.

На рис. 7 представлен результирующий код для рассматриваемого векторизуемого цикла. Дадим некоторые краткие пояснения. Сначала в строках 4-5, 8-11 выполняется загрузка требуемых векторов с помощью LD (_mm512_load_ps).

определяется листовой линейный участок, который должен исполниться. В строках 26-29 выполняется условная пересылка результатов функции для разных веток исполнения. Строки 32-38 содержат локализацию маловероятной ветки

исполнения. Заключительные строки 41-44 записывают результат в память.

Отметим отдельно строки 13, 14 и 40, обеспечивающие замену переменных, которая позволила получить 45% ускорение.

Заключение

В статье рассмотрены вопросы векторизации программного кода, содержащего сложное управление. Описаны подходы, основанные на слиянии равновероятных ветвей исполнения, а также на выносе из цикла и локализации маловероятной ветви исполнения. Рассмотрено применение описанных методов к части программного кода, входящего в состав решения задачи о распаде произвольного разрыва. Финальный векторизованный код, представленный на рис. 7 показал сокращение времени исполнения на 91% (более, чем в 10 раз) по

сравнению с оригинальной версией при исполнении на микропроцессоре Intel Xeon Phi KNL. Если исключить из данного значения ускорение, полученное вследствие удаления линейных участков и замены переменных, то полученное ускорение, прямо связанное с применением векторизации равняется около 82% (более, чем в 5 раз).

Работа выполнена в МСЦ РАН в рамках государственного задания по теме «Разработка архитектур, системных решений и методов для создания вычислительных комплексов и распределенных сред мультипетафлопсного диапазона производительности, в том числе нетрадиционных архитектур микропроцессоров». Для расчетов при проведении исследований использовался суперкомпьютер МВС-10П, находящийся в МСЦ РАН.

Vectorization of highly branched control using AVX-512 instructions

A.A. Rybakov, S.S. Shumilin

Abstract: Vectorization of computations is one of the most important optimizations, with the help of which a multiple acceleration of the calculation codes can be achieved. With the development of modern microprocessors, the vector length in vector operations is constantly increasing. In modern families of Intel x86 microprocessors (Xeon Phi KNL and Xeon Skylake) the vector length already reaches 512 bits. However, often the program code is written in such a way that the automatic application of vectorization is impossible. The reasons for not using vectorization may be the presence of dependencies between operations in the code, function calls or a non-constant number of iterations of loops in vectorized sockets. The current problem of vectorization of loops whose bodies contain complex control is discussed in the article. Two approaches to the application of vectorization for loops with a condition are proposed and the application of these approaches to a practical task containing highly branched control is discussed.

Keywords: optimization, loop vectorization, predicated execution, AVX-512, intrinsic functions, highly branched control.

Литература

1. А.А. Рыбаков, П.Н. Телегин, Б.М. Шабанов. Проблемы векторизации гнезд циклов с использованием инструкций AVX-512. Электронный научный журнал: Программные продукты, системы и алгоритмы. 2018. № 3. С. 1-11.
2. А.А. Рыбаков. 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.
3. S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann. 1997.
4. R. Allen, K. Kennedy. Optimizing compilers for modern architectures. Morgan Kaufmann. 2001.
5. Intel 64 and IA-32 architectures software developer's manual. Combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Intel Corporation. 2017.
6. J. Jeffers, J. Reinders, A. Sodani. Intel Xeon Phi processor high performance programming. Knights Landing edition. Morgan Kaufmann. 2016.
7. Intel C++ compiler 16.0 user and reference guide. Intel Corporation. 2015.
8. 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.
9. A. Sloss, D. Symes, C. Wright. ARM System developer's guide. Designing and optimizing system software. Morgan Kaufmann. 2004.
10. А.К. Ким, В.И. Перекатов, С.Г. Ермаков. Микропроцессоры и вычислительные комплексы семейства «Эльбрус». Питер. 2013.
11. А.А. Рыбаков, М.В. Маслов. Быстрый региональный компилятор системы двоичной трансляции для архитектуры «Эльбрус». В Международная научно-практическая конференция «Современные информационные технологии и ИТ-образование», сборник избранных трудов, под редакцией проф. В.А. Сухомлина. Москва. 2010. С. 436-443.
12. А.Г. Куликовский, Н.В. Погорелов, А.Ю. Семенов. Математические вопросы численного решения гиперболических систем уравнений. М.: ФИЗМАТЛИТ. 2001.
13. E.F. Toro. Riemann solvers and numerical methods for fluid dynamics. A practical introduction. 2nd edition. Springer. 1999.
14. <https://github.com/dasikasunder/NUMERICA>. Дата обращения 14.08.2018.
15. 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. 2015. Том 15. № 4. С. 731-740.
16. S. Brill. Verification of the wavelet adaptive multiresolution representation method to a prescribed error threshold. Department of Aerospace and Mechanical Engineering University of Notre Dame, AME 48491 Undergraduate research, June 4, 2014.