

Оптимизации, применяемые к графу потока управления программы для повышения эффективности векторизации плоских циклов

Б.М. Шабанов¹, А.А. Рыбаков², А.Д. Чопорняк³

¹МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН, Москва, Россия, shabanov@jssc.ru;

²МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН, Москва, Россия, rybakov@jssc.ru, +7 903 138-88-77;

³МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН, Москва, Россия, adc@jssc.ru

Аннотация. Повышение эффективности суперкомпьютерных расчетов является актуальной задачей ввиду постоянного усложнения моделей, увеличения размеров расчетных сеток и количества обрабатываемых данных. Для удовлетворения актуальных потребностей в вычислительных ресурсах для решения практических и научных задач требуется проведение оптимизации высокопроизводительных вычислений на всех уровнях: распределение вычислений между узлами суперкомпьютерного кластера, оптимизация работы многопоточных алгоритмов вычислений для систем с общей памятью, оптимизация расчетов внутри одного потока вычислений. Векторизация программного кода является низкоуровневой оптимизацией, позволяющей в несколько раз ускорить выполнение горячих участков путем объединения нескольких экземпляров фрагмента программы для одновременного выполнения на одном процессорном ядре. В данной статье описан подход к векторизации программного контекста специального вида, называемого «плоский цикл». При этом особое внимание уделяется оптимизациям графа потока управления с известным профилем исполнения для уменьшения издержек, связанных с наличием обильного управления внутри таких циклов.

Ключевые слова: векторизация, AVX-512, плоский цикл, граф потока управления программы, профиль исполнения программы, слияние линейных участков, локализация маловероятных ветвей исполнения

1. Введение

Векторизация вычислений является одной из наиболее важных низкоуровневых оптимизаций программного кода, позволяющей кратно повысить производительность суперкомпьютерных приложений. С помощью векторизации выполняется объединение однотипных операций со скалярными operandами в векторные инструкции, позволяющие одновременно выполнять данные операции над всеми элементами operandов-векторов. В настоящее время векторные инструкции присутствуют практически во всех современных архитектурах микропроцессоров: SVE (Scalable Vector Extension) в архитектуре ARM [1], инструкции AltiVec и VMX (Vector & Media Extension) в архитектуре POWER [2], расширения MMX (Multimedia Extension), SSE (Streaming SIMD Extension) и AVX (Advanced Vector Extension) в микропроцессорах Intel [3], упакованные инструкции в архитектуре «Эльбрус» [4]. Однако наиболее продвинутым набором инструкций для реализации векторных вычислений является расширение AVX-512 в микропроцессорах Intel, состоящее из векторных

операций, которые работают с 512-битными регистрами zmm (расширения 256-битных ymm регистров из набора инструкций AVX). Данный набор инструкций имеет ряд уникальных особенностей, позволяющих создавать высокоэффективный параллельный код [5]. Наиболее важной особенностью данных инструкций, отличающей их от векторных инструкций в других архитектурах, является возможность выборочного применения операции к элементам векторов. Такое выборочное применение реализуется с помощью использования специальных масочных регистров, которые могут выступать в качестве operandов у большинства инструкций из набора AVX-512. Всего существует 8 масочных регистров (k0-k7). При осуществлении поэлементной векторной операции значение бита в используемом масочном регистре определяет, требуется ли применить операцию к соответствующим элементам векторных operandов (если значение бита маски равно 1, то операция выполняется).

Набор инструкций AVX-512 впервые появился в системе команд микропроцессоров Intel Xeon Phi KNL (Knights Landing) [6]. До этого векторные инструкции присутствовали также в

сопроцессорах Intel Xeon Phi KNC (Knights Corner), однако они не были x86-совместимыми в отличие от KNL. После этого расширение AVX-512 твердо заняло место в наборах инструкций линейки микропроцессоров Intel Xeon (Skylake, Cascade Lake и далее), по мере развития наполняясь новыми инструкциями. В настоящее время набор инструкций состоит из нескольких наборов, реализованных в разных семействах микропроцессоров. Вот только некоторые из них: AVX-512 Foundation – базовые операции для работы с 32-битными и 64-битными данными, включая поэлементные операции над векторами, операции с масками, слияние векторов по маске, сравнение векторов, операции перестановок, конвертации и другие; AVX-512 Conflict Detection – набор операций, предназначенный для разрешения конфликтов при векторизации циклов с помощью динамической проверки диапазонов адресов на пересечение; AVX-512 Exponential and Reciprocal – набор инструкций для поэлементного вычисления с повышенной точностью функций 2^x , x^{-1} , $x^{-0.5}$; AVX-512 Prefetch – инструкции предварительной подкачки данных для операций gather/scatter в микропроцессоре Intel Xeon Phi KNL; AVX-512 Vector Length – расширение многих инструкций на элементы данных размера 128 и 256 бит; AVX-512 Doubleword and Quadword – дополнительные инструкции для работы с элементами данных размера 32 и 64 бита; AVX-512 Byte and Word – расширение набора инструкций для работы с данными размера 8 и 16 бит; AVX-512 Vector Neural Network Instructions – дополнительные инструкции для оптимизации алгоритмов машинного обучения.

Условно по схеме работы множество инструкций AVX-512 можно разделить на несколько групп. В качестве первой группы рассмотрим операции, получающие на вход один операнд – zmm регистр и вырабатывающие один результат – также zmm регистр. При выполнении таких векторных операций к каждому элементу входного вектора применяется определенная функция, а после ее вычисления результат записывается в выходной регистр (если соответствующий бит маски выставлен в единицу). Примерами таких операций являются получение абсолютного значения, извлечение корня, операции сдвигов на константу и другие. Другой группой операций являются операции, получающие на вход два векторных операнда. Отличием от первой группы является только то, что операции, применяемые к элементам векторов, принимают два аргумента вместо одного: операции сложения, вычитания, умножения, деления и другие. В качестве следующей группы можно выделить

операции поэлементной конвертации векторов из одного типа данных в другой (множества инструкций cvt и pack). Важной группой операций являются векторные операции поэлементного вычисления значений вида $\pm a \cdot b \pm c$. Также отдельной группой операций являются операции перестановки элементов векторов; они не выполняют никаких арифметических действий, а меняют порядок расположения элементов внутри векторов (к таким операциям относятся разнообразные инструкции unpk, shuf, align, blend, perm). Среди других специальных операций можно отметить операции по множественному доступу в память одновременно к элементам данным, расположенным не последовательно, а по произвольным адресам, операции пересылки данных с дублированием, операции предварительной подкачки данных в кэш и многие другие.

Использованию специфических свойств векторных операций в мире уделяется достаточно много внимания. Можно отметить работы, направленные на повышение эффективности векторизации различных высокопроизводительных приложений: ускорение программного кода LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [7], использование масочных операций для векторизации циклов с нерегулярным числом итераций на примере построения множества Мандельброта [8], векторизация упрощенного римановского решателя для численного решения уравнений мелкой воды [9,10], создание векторизованных версий сортировки массивов чисел [11], векторизация операций над разреженными матрицами [12,13], оптимизация других прикладных пакетов высокопроизводительных вычислений [14,15].

2. Плоский цикл как удобный для векторизации контекст

Векторизация программного контекста не может быть применена автоматически к коду произвольного вида. В данном разделе определим подходящих для векторизации программный контекст специального вида – плоский цикл – и опишем его свойства.

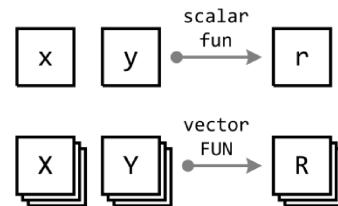


Рис. 1. Объединение нескольких экземпляров вызова функции fun для формирования векторной версии функции FUN.

На рис. 1 сверху представлена схема функции `fun`, которая получает на вход два аргумента `x` и `y` и на основании них формирует результат `r`. Будем считать, что данная функция является чистой, то есть результат ее выполнения зависит только от значений `x` и `y` (например, отсутствуют побочные эффекты через глобальную память или операции ввода-вывода). Идеология чистых вычислений берет свое начало из парадигмы функционального программирования [16], использование такого подхода открывает возможности для оптимизации программного кода, в частности для компиляторов. Если рассмотреть вместо одного вызова функции `fun` несколько вызовов (`w` штук) с разными наборами входных параметров, то их можно трактовать как вызов некоторой функции `FUN`, входными параметрами которой являются векторы `X` и `Y` длины `w`, а выходным значением является вектор `R` также длины `w`. В данном случае можно говорить, что функция `FUN` представляет собой реализацию векторизованной функции `fun` при ширине векторизации `w` (рис. 1 снизу). В этом случае семантику функции `FUN` можно записать в виде

```
for (int i = 0; i < w; i++)
{
    r[i] = fun(x[i], y[i])
}
```

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

Плоские циклы, обладающие описанными свойствами, представляют собой удобный контекст для векторизации, и в большинстве случаев они могут быть векторизованы с помощью векторных инструкций AVX-512 с помощью перевода тела цикла в предикатное представление и замены скалярных инструкций векторными аналогами, реализованными с помощью функций-интринсиков [17,18].

Многие практические вычислительные задачи состоят из выполнения однотипных вычислений, применяемых к разным наборам данных, которые можно сгруппировать, трансформировав в плоский цикл, как это продемонстрировано на рис. 1 на примере функции `fun`.

Сложность векторизации тела полученного плоского цикла зависит от особенностей исходной функции `fun`. Чем сложнее управление внутри тела векторизуемого плоского цикла, тем больше трудностей может возникнуть в процессе выполнения векторизации. Для оценки сложности структуры векторизуемого тела цикла требуется построить граф потока управления данного цикла.

3. Представление структуры тела плоского цикла в виде графа потока управления

Граф потока управления (control flow graph, CFG) является одним из видов промежуточного представления программы, которые используются в частности для выполнения оптимизаций исполняемого кода [19]. Узлами данного графа являются линейные участки, состоящие из последовательностей инструкций, а ребрами – передача управления между этими линейными участками [20]. Граф потока управления является логической структурой, отражающей параллелизм программы на уровне линейных участков. Он активно используется компилятором для применения различных глобальных оптимизаций [21].

Кроме самой структуры графа потока управления для оптимизации программного кода важна статистическая информация об исполнении программы: количество выполнений различных линейных участков и данные о частоте переходов между ними. Такая статистическая информация называется профилем исполнения, и для корректного проведения оптимизаций требуется правильным образом собирать и корректировать данный профиль [22]. Для векторизации программного кода профиль исполнения программы приобретает особенную важность, так как на эффективность векторизации сильно влияет даже структура векторных масок, которая сильно разнится при сравнении результатов, полученных при генерации случайных входных данных, и при использовании реальных данных расчетов [23].

Мы в качестве профиля исполнения приложения будем использовать данные только счетчики узлов и ребер, а также вероятности ребер. Обозначим некоторый узел CFG через `v`. Пусть в

него входят несколько ребер $E_i(v)$, а также выходят несколько ребер $E_o(v)$ (рис. 2). Счетчиком произвольного ребра e будем называть количество переходов по этому ребру в процессе выполнения программы (значение счетчика ребра будем обозначать $n(e)$). Счетчиком узла будем называть суммарное количество счетчиков всех входных ребер либо всех выходных ребер (для внутренних узлов CFG эти значения совпадают).

$$n(v) = \sum_{e \in E_i(v)} n(e) = \sum_{e \in E_o(v)} n(e)$$

Вероятностью ребра будем называть отношение счетчика данного ребра к счетчику узла, из которого это ребро выходит.

$$p(e \in E_o(v)) = n(e)/n(v)$$

Построенный по телу плоского цикла граф потока управления с собранным профилем исполнения будем использовать для принятия решения о выборе методов векторизации программного контекста.

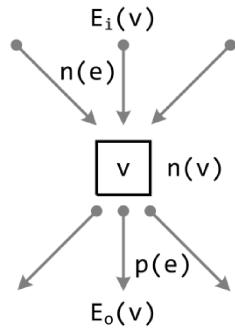


Рис. 2. Определение счетчиков узла и ребра, а также вероятности ребра.

Тривиальной разновидностью CFG является граф, состоящий из единственного линейного участка. Это означает, что в теле рассматриваемого плоского цикла отсутствуют операции передачи управления. Векторизация для таких видов цикла не представляет никаких проблем, она выполняется любым оптимизирующими компилятором (в частности gcc [24] или icc [25]) автоматически.

Если же CFG является сложным графом, то принятие решения о векторизации не всегда может быть принято компилятором. В некоторых случаях приходится прибегать к дополнительным глобальным оптимизациям, связанным с модификацией CFG. Проблема заключается в том, что переходы между линейными участками выполняются по предикатам, которые строятся исходя из условий между элементами векторов, то есть данные условия принципиально являются скалярными. При векторизации же вычислений мы сталкиваемся с ситуациями, что при

выполнении векторных операций для одних элементов вектора условия переход должен быть выполнен, а для других не должен, что одновременно невозможно. На помощь в данном случае и приходят оптимизации графа потока управления.

4. Оптимизации графа потока управления для обеспечения векторизации плоского цикла

Наиболее простой и прямолинейной оптимизацией CFG для обеспечения векторизации является слияние параллельных ветвей исполнения. Рассмотрим слияние двух альтернатив одного условия, записанного в виде

```

if (cnd)
{
    block1
}
else
{
    block2
}
  
```

В данном случае при выполнении условия cnd будет выполнен участок кода $block1$, в противном случае будет выполнен участок кода $block2$. Пусть у нас собран профиль исполнения программы, и известны вероятности передачи управления на участки $block1$ и $block2$ (они равны p_1 и $p_2 = 1 - p_1$ соответственно). Также пусть известны времена выполнения t_1 и t_2 данных линейных участков в некоторых единицах измерения (например, в количестве инструкций). В этом случае общее время работы плоского цикла с приведенным телом равно $T_w = (p_1 t_1 + p_2 t_2)w$. Для возможности проведения векторизации тело цикла необходимо переписать в предикатной форме [26], в которой операции перехода заменены на операции под предикатами, имеющие векторные аналоги в наборе инструкций AVX-512. Условно тело цикла может быть переписано в следующем виде:

```

block1 ? CND
block2 ? ~CND
  
```

При успешной векторизации данного фрагмента время работы векторизованного плоского цикла составит $T_w = t_1 + t_2$. При этом скалярные операции заменены на векторные аналоги, однако оба линейных участка теперь должны выполняться под предикатами, а значит они будут занимать расчетное время. Ожидаемое ускорение от применения векторизации в данном случае равно

$$S = \frac{T_1}{T_w} = \frac{(p_1 t_1 + p_2 t_2)w}{t_1 + t_2}$$

Заметим, что данная оценка верна в предположении, что сами линейные участки *block1* и *block2* не содержат операций передачи управления и векторизуются.

В общем случае, можно привести оценку эффективности векторизации методом слияния всех ветвей исполнения. Пусть тело плоского цикла состоит из *n* линейных участков одной длины, и все они равновероятны, то есть выполняются с вероятностью $p = n^{-1}$. Тогда теоретическое ускорение от применения векторизации составит

$$S = \frac{\left(\sum_{i=1}^n p_i t_i \right) w}{\sum_{i=1}^n t_i} = \frac{w}{n}$$

Видно, что с ростом количества линейных участков даже теоретическое ускорение от векторизации падает, на практике эффективность падает гораздо сильнее, и уже при наличии 3-4 параллельных ветвей исполнения применение векторизации оказывается невыгодным.

Немного сгладить негативный эффект от безусловного слияния всех ветвей исполнения помогает проверка маски перед выполнением векторизованных линейных участков с достаточно низкой вероятностью.

Представим в графическом виде наше рассматриваемое простое условие (рис. 3).

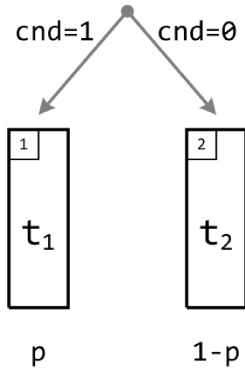


Рис. 3. Графическое представление невекторизованного условия.

В данном графе потока управления у приведенного условия есть всего две альтернативы (в зависимости от значения условия *cnd*). Однако при выполнении векторизация маска предикатов *CND* состоит из *w* элементов, каждый из которых может принимать значения 1 или 0. Причем если все элементы вектора *CND* равны 1, то выполняться должны только инструкции из линей-

ного участка *block1* (рис. 4 слева); если все элементы вектора *CND* равны 0, то выполняться должны только инструкции из линейного участка *block2* (рис. 4 справа); если же вектор *CND* содержит различные значения (и 1, и 0), то необходимо выполнять инструкции из обоих линейных участков под нужными предикатами (рис. 4 по центру).

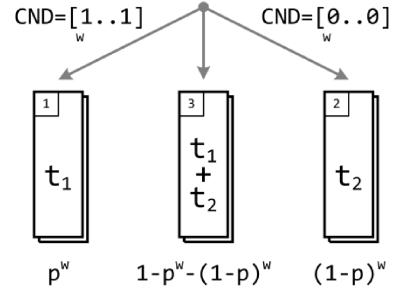


Рис. 4. Графическое представление альтернатив простого условия при выполнении векторизации.

Поэтому при выполнении слияния линейных участков векторизованные операции из соответствующего линейного участка можно выполнять при том условии, что маска предикатов этого линейного участка ненулевая.

```

if (CND != 0x0)
{
    block1 ? CND
}
if ((~CND) != 0x0)
{
    block2 ? ~CND
}
    
```

На первый взгляд может показаться, что такие проверки масок на пустоту не способны принести значительную пользу при векторизации, так как теоретические вероятности данных событий (p^w , $(1-p)^w$) – величины маленькие по сравнению со значениями p , $1-p$. Однако, как показывает практика, в реальных расчетных приложениях, относящихся к задачам компьютерного моделирования физических процессов, при объединении нескольких экземпляров функций в плоские циклы значения $a[i]$ всех элементов некоторого вектора оказываются крайне близкими между собой. Поэтому часто оказывается, что и все значения предикатных векторов также близки между собой. А так как элементами предикатных векторов являются только два значения (1 или 0), то часто оказывается, что предикатный вектор это либо пустая, либо полная маска, поэтому простые проверки маски на пустоту способны существенно повысить производительность векторизуемого кода. Данный эффект подробно описан в [23]. Стоит правда заме-

тить, что данный прием работает только с расчетными кодами, касающимися компьютерного моделирования физических процессов. Для программ с дискретным контекстом (таких, как сортировка массивов чисел или задачи обработки дискретных структур данных) этот метод не приносит ничего кроме накладных расходов на дополнительные операции сравнения [27].

В рассмотренных выше случаях слияния ветвей исполнения под разными предикатами всегда предполагалось, что программный код всех линейных участков векторизуется. Однако, возможны ситуации, когда в теле плоского цикла су-

ществуют пути исполнения с экстремально низкой вероятностью, векторизация которых либо невозможна, либо не представляет никакого практического смысла (требует слишком много затрат). Однако плоские циклы с такими редкими путями исполнения также могут быть векторизованы. Одним из способов векторизации может выступать локализация ветви с низкой вероятностью. Данный метод описан в [28] на примере практической задачи. Мы же опишем этот метод в общем случае с помощью подхода, который в некоторых источниках применительно к своим предметным областям встречается под названием *blackhole* (черная дыра) [29].

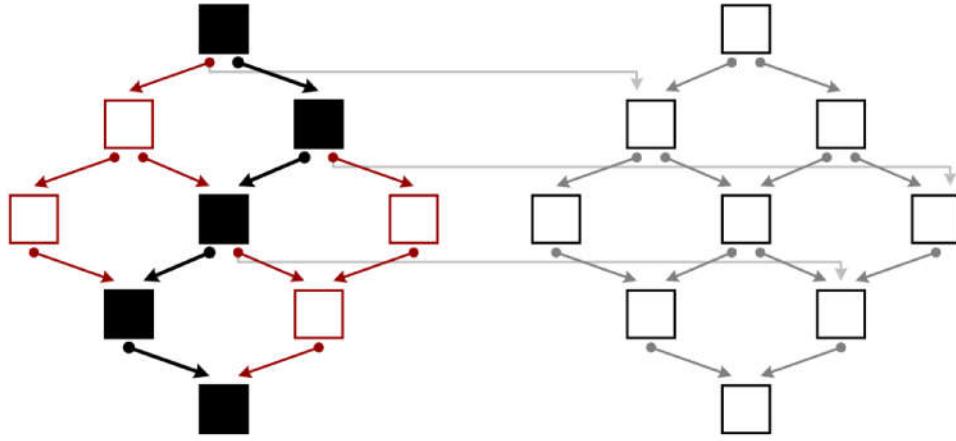


Рис. 5. Иллюстрация схемы работы оптимизации *blackhole*.

Пусть дам некоторый CFG тела плоского цикла (рис. 5 слева), в котором присутствует явно выделенный путь исполнения (на рисунке выделен черным цветом), вероятность прохождения которого близка к единице. Другие крайне маловероятные линейные участки (выделенные на рисунке красным цветом) представляют собой программный контекст, который практически никогда не исполняется, и векторизацию которого проводить нецелесообразно, либо невозможно. Оптимизация *blackhole* заключается в создании точной копии CFG, на которую перенаправляются все маловероятные переходы из основного тела. После выполнения перенаправления маловероятных переходов все ребра и линейные участки, отмеченные на рис. 5 красным цветом, могут быть удалены. Таким образом, после выполнения оптимизации в качестве объекта векторизации остается ограниченный и пригодный к векторизации программный контекст. В случае же если один из маловероятных переходов все же осуществляется, то выполнится переход на точную копию изначального CFG, который хоть и не оптимально, но корректно отработает данную редкую ситуацию. Данная оптимизация получила свое название *blackhole* потому что после выполнения редкого перехода на копию CFG

программа не может вернуться обратно в векторизованную версию, в этом случае данный экземпляр плоского цикла закончит свое исполнение в черной дыре. Данная оптимизация может применяться в различных вариациях. Например, можно создавать копию не целого CFG, а только конкретных маловероятных регионов; в этом случае мы будем иметь оптимизацию локализации редких путей исполнения. Наоборот, более консервативным вариантом оптимизации является перенаправление маловероятных переходов сразу на голову невекторизованной копии тела цикла, что соответствует просто проведению повторного расчета при возникновении исключительной ситуации.

В данном разделе был рассмотрен набор оптимизаций, которые могут быть применены к CFG тела плоского цикла для обеспечения возможности применения к нему векторизации. Они различаются по условиям применения и характеристикам профиля исполнения.

При отсутствии профиля исполнения программы единственным вариантом применения векторизации остается безусловное слияние ветвей исполнения. Если профиль исполнения доступен, то выбор того или иного метода вектори-

зации зависит от значений вероятностей линейных участков рассматриваемого CFG. При умеренно низких вероятностях исполнения линейных участков и относительной доступности векторизации их кода целесообразно ограничиваться добавлением проверок на пустоту масок этих линейных участков. Если же в коде присутствуют линейные участки с крайне низкой вероятностью исполнения, которые к тому же трудно поддаются векторизации, то такие участки стоит подвергать локализации, выносить из циклов. В случае преобладания в цикле доминантного хорошо векторизуемого пути исполнения, вероятность которого близка к единице (в условиях наличия в теле цикла обильных участков кода с экстремально низкой вероятностью), применение оптимизации blackhole представляется наилучшим выбором.

5. Заключение

Проведение исследований в области разработки новых методов векторизации программного кода имеет важное значение для повышения эффективности выполнения суперкомпьютерных расчетов. В современных суперкомпьютерных приложениях часто встречаются фрагменты кода с сильно разветвленным графом потока исполнения, что препятствует векторизации с помощью автоматических средств. Особенности набора векторных инструкций AVX-512 позволяют выполнять векторизацию практически произвольного программного контекста. Использование подхода по описанию вычислений в виде плоских циклов позволяет еще больше повысить шансы на успешную вектори-

зацию. Существует набор методов по векторизации программного кода с разветвленным управлением, основанных на использовании статистики исполнения отдельных линейных участков программы. В зависимости от статистики для эффективной векторизации кода могут быть выбраны различные методы с разной степенью агрессивности выполнения оптимизации. Сбор и анализ профиля исполнения программы крайне важен для принятия решения о выборе стратегии векторизации. Разработка и анализ различных подходов и методов векторизации сложного программного контекста были произведены коллективом авторов данной статьи на основе опыта проведения векторизации программного кода реальных высокопроизводительных приложений: была выполнена полная векторизация кода точного римановского решателя, что позволило ускорить его в 7 раз по сравнению со скалярной версией [30]; были разработаны векторизованные версии функций для работы с геометрическими примитивами [31] в рамках реализации метода погруженных границ численного решения задач газовой динамики [32]; были разработаны векторизованные версии функций для работы с матрицами специального вида [33] при реализации метода RANS/ILES расчета турбулентных течений [34]; были опробованы методы векторизации гнезд циклов на примере сортировки массивов чисел [35], была векторизована реализация определения конфликтов со спутными следами летательных аппаратов [28].

Работа выполнена при поддержке гранта РФФИ № 20-07-00594. При проведении исследований были использованы суперкомпьютеры МСЦ РАН: МВС-10П, МВС-10П ОП [36].

Optimizations Applied to the Control Flow Graph of a Program to Improve the Efficiency of Flat Loops Vectorization

B.M. Shabanov, A.A. Rybakov, A.D. Choporniyak

Abstract. Improving the efficiency of supercomputer calculations is an urgent problem due to the permanent complication of models, an increase in the size of computational meshes and the amount of data being processed. To meet the current needs for computing resources for solving practical and scientific problems, it is necessary to optimize high-performance computing at all levels: the distribution of computations between the nodes of the supercomputer cluster, optimization of the operation of multithreaded computational algorithms for systems with shared memory, optimization of computations within one computation thread. Vectorization of program code is a low-level optimization that allows several times to speed up the execution of hot spots by combining several instances of a program fragment for simultaneous execution on one processor core. This article describes an approach to vectorizing a program context of a special kind called a flat loop. At the same time, special attention is paid to optimizing the control flow graph with a known execution profile to reduce the costs associated with the presence of abundant control within such loops.

Keywords: vectorization, AVX-512, flat loop, program control flow graph, program execution profile, linear sections merging, unlikely execution branches localization

Литература

1. N. Stephens, S. Biles, M. Boettcher et al. The ARM scalable vector extension. *IEEE Micro*, 2017, 37 (2), p. 26-39.
2. M. Gschwind. Workload acceleration with the IBM POWER vector-scalar architecture. *IBM Journal of Research and Development*, vol. 60 (2016), No. 2/3, paper 14, p. 1-18.
3. Intel 64 and IA-32 architectures software developer's manual. Combine volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Intel Corporation, April 2021.
4. В.Ю. Волконский, А.В. Брегер, А.В. Грабежной и др. Методы распараллеливания программ в оптимизирующем компиляторе для ВК семейства Эльбрус. Современные информационные технологии и ИТ-образование, 2011, № 7, с. 46-59.
5. В.М. Шабанов, А.А. Рыбаков, С.С. Шумилин. Vectorization of high-performance scientific calculations using AVX-512 instruction set. *Lobachevskii Journal of Mathematics*, vol. 40 (2019), № 5, p. 580-598.
6. J. Jeffers, J. Reinders, A. Sodani. Intel Xeon Phi processor high performance programming, Knights Landing edition. Morgan Kaufmann Publishers, 2016.
7. W. McDoniel, M. Höhnerbach, R. Canales et al. LAMMPS' PPPM long-range solver for the second generation Xeon Phi. J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, vol. 10266 (2017), p. 61-78.
8. O. Krzikalla, F. Wende, M. Höhnerbach. Dynamic SIMD vector lane scheduling. M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, vol. 9945 (2016), p. 354-365.
9. M. Bader, A. Breuer, W. Holtz, S. Rettenberger. Vectotization of an augmented Riemann solver for the shallow water equations. Proceedings of the 2014 International Conference on High Performance Computing and Simulation, HPCS 2014, p. 193-201.
10. 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, p. 415-422.
11. B. Bramas. A novel hybrid quicksort algorithm vectorized using AVX-512 on Intel Skylake. *International Journal of Advanced Computer Science and Applications*, vol. 8 (2017), № 10, p. 337-344.
12. E. Coronado-Barrientos, M. Antonioletti, A.G. Loureiro. A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA. *Advances in Engineering Software*, vol. 156 (2021).
13. B. Bramas, P. Kus. Computing the sparse matrix vector product using block-based kernels without zero padding on processors with AVX-512 instructions. arXiv:1801.01134v2, 2018.
14. I. Kulikov, I. Chernykh, A. Tutukov. Hydrogen-helium chemical and nuclear galaxy collision: Hydrodynamic simulations on AVX-512 supercomputers. *Journal of Computational and Applied Mathematics*, 2021, 391 (1).
15. I.M. Kulikov, I.G. Chernykh, A.V. Tutukov. A new parallel Intel Xeon Phi hydrodynamics code for massively parallel supercomputers. *Lobachevskii Journal of Mathematics*, vol. 39 (2018), No. 9, p. 1207-1216.
16. J. Armstrong. Programming Erlang. Software for a concurrent world. The Pragmatic Programmers, 2013, 520 p.
17. G.I. Savin, B.M. Shabanov, A.A. Rybakov, S.S. Shumilin. Vectorization of flat loops of arbitrary structure using instructions AVX-512. *Lobachevskii Journal of Mathematics*, vol. 41 (2020), No. 12, p. 2566-2574.
18. Intel Intrinsics Guide. <https://software.intel.com/sites/landingpage/IntrinsicsGuide> (дата обращения 01.06.2021)
19. S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, 1997.
20. А.А. Рыбаков. Алгоритм создания случайных графов потока управления для анализа глобальных оптимизаций в компиляторе. Parallel and Distributed Computing Systems PDSC 2013, Collection of scientific papers, p. 269-275.
21. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ulman. Compilers: principles, techniques, and tools. Prentice Hall, 2nd ed, 2006.
22. О.А. Четверина. Методы коррекции профильной информации в процессе компиляции. Труды ИСП РАН, том 27 (2015), вып. 6, с. 49-63.
23. А.А. Рыбаков, А.Д. Чопорняк. Повышение производительности векторного кода с помощью

- мониторинга плотности масок в векторных инструкциях. Труды НИИСИ РАН, т. 10 (2020), № 4, с. 40-47.
24. Using the GNU Compiler Collection. For GCC version 7.4.0. Richard M. Stallman and the GCC Developer Community.
25. Intel C++ Compiler Classic Developer Guide and Reference. <https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html> (дата обращения 01.06.2021)
26. 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.
27. 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.
28. А.А. Рыбаков. Оптимизация задачи об определении конфликтов с опасными зонами движения летательных аппаратов для выполнения на Intel Xeon Phi. Программные продукты и системы, т. 30 (2017), № 3, с. 524-528.
29. B. Ilbeyi. Co-optimizing hardware design and meta-tracing just-in-time compilation. A dissertation for the degree of Doctor of Philosophy, Cornell University, 2019.
30. A.A. Rybakov, S.S. Shumilin. Vectorization of the Riemann solver using the AVX-512 instruction set. Program Systems: Theory and Applications, vol. 10 (2019), № 3 (42), p. 41-58.
31. А.А. Рыбаков. Векторизация нахождения пересечения объемной и поверхностной сеток для микропроцессоров с поддержкой AVX-512. Труды НИИСИ РАН, т. 9 (2019), № 5, с. 5-14.
32. Y.-H. Tseng, J.H. Ferziger. A ghost-cell immersed boundary method for flow in complex geometry. Journal of Computational Physics, v. 192 (2003), p. 593-623.
33. Л.А. Бендерский, А.А. Рыбаков, С.С. Шумилин. Векторизация перемножения матриц специального вида с использованием инструкций AVX-512. Современные информационные технологии и ИТ-образование, т. 14 (2018), № 3, с. 594-602.
34. L.A. Benderskii, D.A. Lyubimov, A.O. Chestnykh, B.M. Shabanov. The use of the RANS/ILES method to study the influence of coflow wind on the flow in a hot, nonisobaric, supersonic airdrome jet during its interaction with the jet blast deflector. High temperature, vol. 56 (2018), No. 2, p. 247-254.
35. А.А. Рыбаков, С.С. Шумилин. Исследование эффективности векторизации гнезд циклов с нерегулярным числом итераций. Программные системы: Теория и алгоритмы, т. 10 (2019), № 4 (43), с. 77-96.
36. JSCC RAS Supercomputing resources. <http://www.jsc.ru/supercomputing-resources/> (дата обращения 01.06.2021)