Автоматическая векторизация циклов со сложным управлением

Ермолицкий А.В., аспирант ИНЭУМ
ermolitsky@nm.ru
Шлыков С.Л., нач. отд. МЦСТ
shlykov@mcst.ru

Введение

Широкое распространение мультимедийных приложений привело к тому, что сейчас практически во всех микропроцессорах общего назначения используются наборы коротких векторных инструкций. Как показывает практика, это позволяет увеличить производительность процессора при выполнении задач со значительной степенью параллелизма на уровне данных.

К настоящему моменту для оптимизирующих компиляторов разработан ряд методов автоматической генерации векторных инструкций [1, 2, 3, 4]. Они позволяют векторизовать код без разветвлений управления либо с некоторыми шаблонными ветвлениями. Тем не менее, в ряде практически важных приложений встречаются циклы со сложным управлением, к которым эти методы неприменимы. Это приводит к необходимости ручной векторизации кода для достижения максимальной производительности, что увеличивает время разработки приложений и негативно влияет на их переносимость. В работе рассмотрен разработанный авторами метод генерации векторных инструкций, позволяющий векторизовать циклы с произвольными разветвлениями управления.

1 Базовый метод векторизации

Рассмотрим базовый метод векторизации, позволяющий векторизовать циклы без разветвлений управления и являющийся основой для описываемых далее методов. Для этого введем следующие определения. *Выражением* будем называть слабо связную компоненту графа определений-использований за исключением инструкций, вычисляющих адреса инструкций обращения к памяти. Два графа назовем *изоморфными*, если существует взаимно однозначное соответствие между их вершинами и ребрами, сохраняющее смежность и инцидентность. Соответствующие инструкции изоморфных графов определений-использований будем называть *изоморфными*.

Основная идея базового метода состоит в раскрутке цикла (дублировании тела, Loop Unroll [5]) для создания копий исходного скалярного выражения и дальнейшей замене векторными инструкциями групп изоморфных скалярных инструкций. При этом

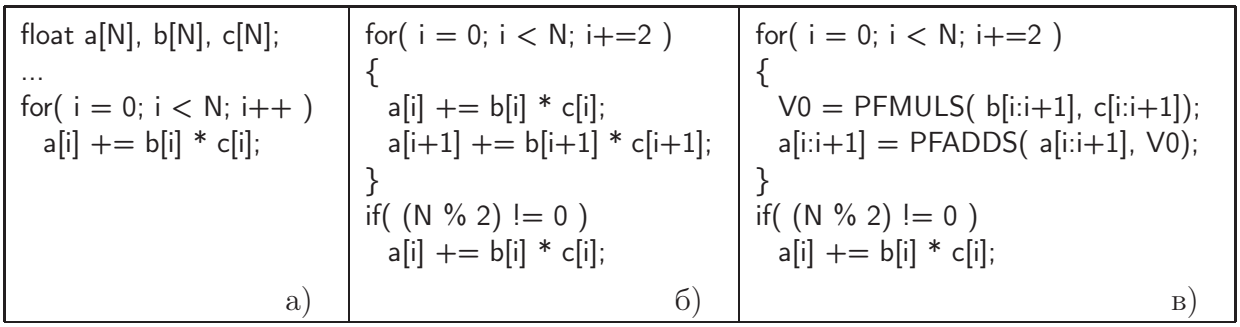


Рис. 1: Пример векторизации цикла без разветвлений управления

фактор раскрутки цикла выбирается таким образом, чтобы число изоморфных скалярных инструкций было кратным количеству элементов векторных инструкций. Более подробно базовый метод описан в [6].

В качестве примера рассмотрим цикл на рисунке 1а, содержащий одно скалярное выражение. В результате раскрутки в цикле появляется изоморфная копия исходного выражения, а после цикла - досчетная итерация, выполняемая в случае, когда исходный цикл имеет нечетное количество итераций (рис. 1б). (В дальнейшем для простоты досчетную итерацию будем опускать.) Далее, две 32-разрядные скалярные инструкции умножения заменяются 64-разрядной векторной инструкцией умножения PFMULS, скалярные инструкции сложения - векторной PFADDS, а мелкоформатные инструкции чтения/записи - соответствующими векторными обращениями к памяти (рис. 1в). (Здесь и далее используются векторные инструкции из системы команд процессора «Эльбрус».) В результате векторизации цикл значительно ускоряется за счет параллельного исполнения итераций исходного цикла.

Базовый метод векторизации неприменим к циклам с разветвлениями управления за исключением случая, когда некоторые семантические конструкции могут быть заменены специализированными векторными инструкциями (такими как вычисление максимума) [7]. Следующие разделы посвящены снятию данного ограничения.

2 Векторизация циклов с разветвлениями управления

Многие современные мультимедийные расширения содержат инструкции векторного сравнения, вырабатывающие битовые маски в качестве результата. Они принимают два аргумента и сравнивают их соответствующие элементы. Если результат сравнения i-ой пары элементов есть *истина*, то i-ый элемент результата заполняется единичными битами, если *ложь*, - то нулевыми.

Инструкции векторного сравнения позволяют векторизовать условный код при помощи метода *битового маскирования* (bit masking). Основная идея предлагаемого ме-

```
char a[N], b[N], c[N]; int x;
...
for( i = 0; i < N; i++ ) {
  if( a[i] > b[i] )
    x = a[i];
  else
    x = b[i];
  c[i] = x;
}
```
а)

```
for( i = 0; i < N; i+=8 ) {
  V0 = PCMPGTB( a[i:i+7], b[i:i+7]);
  V1 = PAND( a[i:i+7], V0);
  V2 = PANDN( b[i:i+7], V0);
  c[i:i+7] = POR( V1, V2);
}
```
б)

Рис. 2: Пример векторизации цикла с разветвлениями управления

тода заключается в преобразовании множества узлов управляющего графа выражения в линейный участок, так что в получившемся выражении на каждой итерации цикла безусловно исполняются векторные образы всех скалярных инструкций исходного выражения. Затем при помощи битовых масок, вырабатываемых инструкциями векторного сравнения, и логических инструкций PAND, PANDN, POR (И, И-НЕ, ИЛИ соответственно) выбираются нужные элементы результатов векторных инструкций. Битовую маску при этом можно рассматривать как *векторный предикат* (ВП), содержащий векторное условие передачи управления в некоторый узел (по некоторой дуге) управляющего графа исходного выражения - элемент ВП заполняется единичными битами, если в этот узел (по этой дуге) передается управление на соответствующей итерации исходного цикла, или нулевыми битами в противном случае. Операции PAND, PANDN и POR таким образом используют ВП для *сведения потоков данных* с разных веток управления. Данный метод позволяет векторизовать выражения с произвольными разветвлениями управления.

В качестве простого примера рассмотрим цикл на рисунке 2а, содержащий выражение с разветвлениями управления. В результате векторизации в цикле остается только один линейный участок, в котором векторные инструкции выполняются безусловно (рис. 2б); здесь V0,V1,V2 - некоторые регистры, PCMPGTB - инструкция векторного сравнения «больше». Регистр V0 содержит ВП - векторное условие передачи управления в узел с операцией x = a[i]; если на некоторой итерации исходного цикла управление попадает в этот узел, то соответствующий элемент V0 будет заполнен единичными битами и в массив c[] попадет значение, считанное из a[], в противном случае элемент V0 заполняется нулевыми битами и в c[] записывается значение из b[].

В общем случае сведение потоков данных производится следующим образом. Пусть в рассматриваемый узел цикла N сходится управление из узлов N_1, N_2, ..., N_k цикла по дугам E_1, E_2, ..., E_k соответственно (рис. 3). Пусть по этим дугам передаются вектора значений V_1, V_2, ..., V_k, а ВП этих дуг равны C_1, C_2, ..., C_k соответственно. Тогда векторное значение на входе в узел N вычисляется по формуле $V = (V_1 \& C_1)|(V_2 \& C_2)|...|(V_k \& C_k)$, где

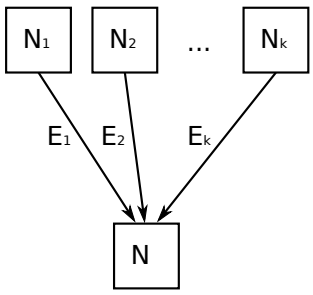


Рис. 3: Схождение потоков данных в исходном цикле

через & и | обозначены векторные инструкции PAND и POR соответственно.

ВП узлов и дуг, необходимые для сведения потоков данных, вычисляются путем распространения ВП по управляющему графу цикла вниз, начиная от самого верхнего узла выражения. Для этого узлы выражения обходятся согласно RPO-нумерации, т.е. каждый узел рассматривается только после обхода всех его предшественников [8]. ВП узлов, постдоминирующих голову цикла всегда истинны, т.е. все их элементы заполнены единичными битами. ВП остальных узлов вычисляется как логическая сумма ВП всех дуг, входящих в этот узел: $P_N = C_1|C_2|...|C_k$, где P_N - ВП узла N (рис. 3). Из узла может выходить одна или две управляющих дуги. Если из узла выходит две дуги, то ВП дуг вычисляется как логическое умножение (PAND для *true* дуги и PANDN для *false* дуги) ВП узла и результата инструкции векторного сравнения, являющейся векторным образом инструкции сравнения исходного узла. Если из узла выходит только одна дуга, то ее ВП совпадает с ВП узла.

Полученный в результате векторизации условного выражения код зачастую содержит множество избыточностей, для устранения которых необходимо выполнить ряд локальных потоковых оптимизаций. Наряду с простыми преобразованиями, такими как замена $x\&0$ на 0 и $x|0$ на x, локальные оптимизации включают в себя множество более сложных преобразований, таких как:

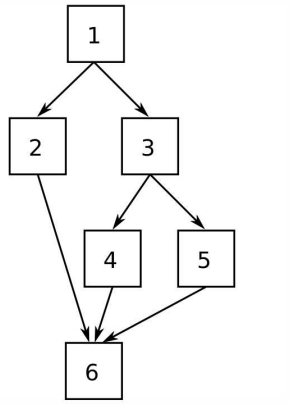
$$X + (C\&\vec{1}) \rightarrow X - C$$
$$X - ((X\&C)|(Y\&!C)) \rightarrow (X - Y)\&!C$$
$$((X + Y)\&C)|(X\&!C) \rightarrow X + (Y\&C)$$

где под &, |, + и − обозначены соответствующие векторные инструкции, X и Y - произвольные вектора, C - произвольный ВП, а $\vec{1}$ - единичный вектор, все элементы которого равны 1.

Рассмотрим пример векторизации цикла с вложенными разветвлениями управления (рис. 4а). Пронумеруем узлы управляющего графа цикла согласно RPO-нумерации (рис. 4б). Эта нумерация задает порядок обхода узлов для вычисления их ВП и генерации векторных инструкций по скалярным. В результате векторизации цикл содержит

```
short *src, *dst; int x;
...
for( i = 0; i < N; i++ ) {
1:    if( src[i] > 1 )
2:        x = 0x7FFF;
3:    else if( src[i] <= 0 )
4:        x = 0x8000;
      else
5:        x = 0;
6:    dst[i] = x;
}
```

а)



б)

```
C0 = 0x0;
C1 = 0x1000100010001;
C2 = 0xFFFFFFFFFFFFFFFF;
C3 = 0x7FFF7FFF7FFF7FFF;
C4 = 0x8000800080008000;
...
for( i = 0; i < N; i+=4 ) {
  V0 = PCMPGTH( src[i:i+3], C1); // ВП узла 2
  V1 = PCMPGTH( src[i:i+3], C0);
  V2 = PXOR( V0, C2);            // ВП узла 3
  V3 = PANDN( V2, V1);           // ВП узла 4
  V4 = PAND( V2, V1);            // ВП узла 5
  V5 = PAND( C0, V4);
  V6 = PAND( C4, V3);
  V7 = PAND( C3, V0);
  V8 = POR( V5, V6);
  dst[i:i+3] = POR( V7, V8);
}
```

в)

```
C0 = 0x0;
C1 = 0x1000100010001;
C3 = 0x7FFF7FFF7FFF7FFF;
C4 = 0x8000800080008000;
...
for( i = 0; i < N; i+=4 ) {
  V0 = PCMPGTH( src[i:i+3], C1);
  V1 = PCMPGTH( src[i:i+3], C0);
  V2 = PANDN( C4, V1);
  V3 = PAND( C3, V0);
  dst[i:i+3] = POR( V2, V3);
}
```

г)

Рис. 4: Пример векторизации цикла с разветвлениями управления

множество инструкций сведения потоков данных, комментарием помечены вычисляющие ВП инструкции (рис. 4в). Это выражении далее упрощается с помощью локальных оптимизаций, позволяющих значительно повысить эффективность векторного кода (рис. 4г).

3 Векторизация циклов с боковыми выходами

Одним из условий применимости базового алгоритма векторизации является *исчислимость цикла*, т.е. возможность вычисления количества итераций до его исполнения. В данном разделе описывается разработанный авторами метод генерации компенсирующего кода, позволяющий снять данное ограничение.

Для дальнейшего изложения необходимо дать формальное определение *бокового выхода* из цикла. Будем называть *счетчиком цикла* целочисленную переменную, контро-

лирующую повторы выполнения цикла и изменяющуюся каждую итерацию на инвариантную величину. Выход из цикла, осуществляемый по достижению счетчиком некоторого инвариантного значения, будем называть *выходом по счетчику*. Выход по счетчику, инцидентный узлу цикла, из которого выходит также обратная дуга, будем называть *базовым выходом*. Выход из цикла, не являющийся базовым, будем называть *боковым выходом*. Таким образом, базовый алгоритм векторизации работает только с циклами, содержащими единственный выход, являющийся базовым.

Метод векторизации цикла с боковыми выходами заключается в следующем. Как и в случае цикла с разветвлениями управления, все внутрицикловые разветвления удаляются и строятся инструкции сведения потоков данных. Все боковые выходы из цикла сохраняются, однако условия передачи управления по этим выходам меняются - боковой выход исполняется, если хотя бы один элемент ВП соответствующей управляющей дуги не равен нулю. На каждую дугу бокового выхода вставляется компенсирующий код, являющийся копией исходного скалярного цикла. Таким образом, в случае выхода из векторизованного цикла по боковому выходу все данные, вычисленные на последней итерации, утрачиваются и пересчитываются в компенсирующем коде.

В теле векторизованного цикла все инструкции записи должны находиться ниже бокового выхода, иначе в векторизованном цикле может быть стерто содержимое ячеек памяти, к которым не было обращений в исходном цикле. Аналогичное требование касается инструкций, результат которых используется за пределами цикла. Для снятия данного ограничения перед векторизацией выполняется реорганизация тела цикла, которая заключается в переносе таких инструкций в нижнюю часть цикла.

В качестве примера рассмотрим векторизацию цикла с боковым выходом. Исходный цикл (рис. 5а) содержит выход по счетчику цикла i и боковой выход по условию (src[i] == 0). Генерация компенсирующего кода заключается в создании копии исходного цикла, управление на которую передается в случае исполнения бокового выхода (рис. 5б). В исходном цикле присутствует инструкция записи в массив dst, стоящая выше бокового выхода, поэтому для векторизации необходимо опустить ее ниже бокового выхода (рис. 5в). В результате раскрутки и дальнейшей векторизации основного цикла создается скалярная инструкция сравнения ВП дуги бокового выхода из цикла, вырабатывающая условие передачи управления по боковому выходу векторизованного цикла (рис. 5г).

4 Результаты

Предложенные методы были реализованы в составе оптимизирующего компилятора для архитектуры «Эльбрус». Их эффективность была проверена в ходе экспериментальных исследований, проведенных на вычислительном комплексе «Эльбрус 3М1».

а)

```
char *dst, *src;
...
i = 0;
do {
    dst[i] = src[i];
    if( src[i] == 0 )
        break;
    i++;
} while( i < N );
```

б)

```
do {
    dst[i] = src[i];
    if( src[i] == 0 )
        goto comp_code;
    i++;
} while( i < N );
goto end;
comp_code:
do {
    dst[i] = src[i];
    if( src[i] == 0 )
        break;
    i++;
} while( i < N );
end:
```

в)

```
do {
    if( src[i] == 0 )
        goto comp_code;
    dst[i] = src[i];
    i++;
} while( i < N );
goto end;
comp_code:
do {
    dst[i] = src[i];
    if( src[i] == 0 )
        break;
    i++;
} while( i < N );
end:
```

г)

```
do {
    V0 = PCMPEQB( src[i:i+7], 0);
    if( V0 != 0 )
        goto comp_code;
    dst[i:i+7] = src[i:i+7];
    i+=8;
} while( i < N );
goto end;
comp_code:
do {
    dst[i] = src[i];
    if( src[i] == 0 )
        break;
    i++;
} while( i < N );
end:
```

Рис. 5: Пример векторизации цикла с боковым выходом

Автоматическая векторизация позволила увеличить производительность на ряде задач из пакетов тестов SPEC CINT и SPEC CFP [9]. Величина прироста производительности составила 83% на задаче 023.eqntott, а также 29%, 61%, 117%, 11% на задачах 052.alvinn, 102.swim, 124.m88ksim, 256.bzip2 соответственно. Задачи 023.eqntott и 124.m88ksim были ускорены за счёт векторизации горячих циклов с боковым выходом.

Кроме того, эффективность предложенных алгоритмов была исследована на 373 функциях, входящих в состав высокопроизводительной библиотеки векторных вычислений EML (Elbrus Math Library [10]). Данные функции реализуют наиболее распространенные операции над векторами и матрицами. Средний прирост производительности функций EML за счет векторизации составил 52%.

Список литературы

[1] Larsen, S., Amarasinghe, S. Exploiting superword level parallelism with multimedia instruction sets. SIGPLAN Not., volume 35, № 5:pp 145–156. — 2000

[2] Cheong, G., Lam, M. An optimizer for multimedia instruction sets. Proceedings of the Second SUIF Compiler Workshop, http://www-suif.stanford.edu/suifconf/suifconf2. 1997

[3] Sreraman, N., Govindarajan, R. A vectorizing compiler for multimedia extensions. International Journal of Parallel Programming, volume 28:pp 363–400. — 2000

[4] Bik, A.J.C., Girkar, M., Grey, P.M., Tian, X. Automatic intra-register vectorization for the intel architecture. Int. J. Parallel Program., volume 30, № 2:pp 65–98. — 2002

[5] Kennedy, K., Allen, J.R. Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001

[6] Волконский, В., Дроздов, А., Ровинский, Е. Метод использования мелкоформатных векторных операций в оптимизирующем компиляторе. Информационные технологии и вычислительные системы, , № 3:pp 63–77. — 2004

[7] Bik, A.J.C., Girkar, M., Grey, P.M., Tian, X. Automatic detection of saturation and clipping idioms. LCPC, pp 61–74. 2002

[8] Muchnick, S.S. Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco, CA, 1997

[9] Standard Performance Evaluation Corporation. The SPEC Benchmark Suites. CPU-intensive benchmark suite. [Electronic resource]. — 1995-2000. http://www.spec.org/cpu

[10] MCST. Elbrus Math Library. [Electronic resource]. — 2007. http://mossigplan.acm.org/EML_introduction_engl.pdf