Развитие метода автоматической генерации мелкоформатных векторных операций

Волконский В.Ю., Дроздов А.Ю., Ровинский Е.В.,

ИМВС РАН, г. Москва

1. Введение.

Последнее десятилетие в состав новых процессоров обязательно входят мультимедийные расширения. В их основу положен принцип SIMD - single instruction - multiple data (одна инструкция - множественные данные). Впервые такое расширение, названное MAX-1, представила компания Hewlett Packard, а после этого появились VIS фирмы SUN, VMX/AntiVec, представленное IBM/Motorola, MMX и его расширения SSE и SSE2 компании Intel, 3DNow! AMD. [2, 7, 8].

В использовании аппаратных ресурсов, предоставляемых мультимедийными расширениями, все еще преобладает традиционный подход, когда соответствующие операции расширенной системы команд попадают в код, будучи вставлены в код специально написанными ассемблерными вставками. Другой способ получения высокопроизводительного кода - автоматическое выявление в программе высокого уровня векторизуемых вычислений и генерация соответствующих мультимедийных операций с помощью компилятора.

В статье рассматриваются новые подходы к генерации операций SIMD, которые мы будем называть *векторными* операциями. Основная идея работы векторных операций заключается в следующем. Пусть даны операнды размера n_p бит, состоящие из вектора значений, размер каждого элемента - n_b бит. (Число n_p мы назовем *общим* размером операнда, а n_b – *базовым* размером элемента вектора операнда). Векторная операция выполняет действие над всем операндом, при этом над каждым из базовых операндов выполняется какое-то особенное действие. В общем случае это действие не одно и то же, однако, большую часть операций мультимедийного расширения составляют операции, совершающие одно и тоже действие для всех базовых операндов данного операнда.

В качестве примера рассмотрим работу векторной операции PADDB системы команд (*СК*) Эльбрус 3М [3],[6]. У этой операции n_b=8, а n_p=64. Эта операция имеет два аргумента, каждый из которых рассматривается как вектор из восьми восьмибитных значений. В качестве результата выдается один вектор из восьми восьмибитных значений, каждый член которого есть сумма двух соответствующих членов векторов – аргументов операции PADDB.

Понятно, что такая операция может позволить ускорить вычисление сложения восьмибитных элементов до 8 раз, но сгенерировать ее именно в тех случаях и таким образом, чтобы это привело к увеличению производительности, является сложной задачей. Изучению этой проблемы, а также предложение оптимальных вариантов ее решения и посвящена вторая глава данной статьи. В ней обсуждается новый метод обнаружения векторизуемых фрагментов кода, описывается быстрый алгоритм автоматической генерации векторных операций для произвольных арифметических выражений

В третьей главе описывается распознавание сложных семантических конструкций, которым соответствуют отдельные операции мультимедийного расширения, и генерация этих операций в рамках разработанного метода. Четвертая глава посвящена общим принципам и некоторым частным случаям применения операций векторного сравнения, в пятой описываются предварительные результаты, полученные в результате реализации описанных в статье алгоритмов и методов.

2. Генерация векторных операций для произвольных выражений

2.1 Общий алгоритм.

Алгоритм генерации векторных операций в самом общем виде описан в статье [1]. Этот алгоритм содержит множество очень сложных шагов. После необходимых и в любом другом алгоритме оптимизации unroll [4] и анализа выравненности операций обращения в память - ищутся пары операций чтения (или записи), работающие с соседними участками памяти. В паре каждая операция занимает либо *левое*, либо *правое* положение. Затем совершается проход по графу определений/использований [4], и операции доступа в память дополняются другими операциями, в результате чего авторы получают пары выражений (каждое из которых также является левым или правым в зависимости от положения операции чтения/записи, от которой начат обход). Эти пары выражений нужно скомбинировать с другими парами так, чтобы они образовали большие группы выражений. Таким образом формируются группы *изоморфных* выражений (т.е. выражений, в которых одноименные операции идут в одинаковом порядке). После этого генерируются векторные операции вместо соответствующих друг другу одноименных операций в группе.

Общий алгоритм, несмотря на свою универсальность, имеет некоторые недостатки – он сложен, ориентирован на длинные векторы, и не вполне оправдан для коротких векторов (описываемый алгоритм применяется для векторов длиной до 1024 бит, между тем как архитектура Эльбрус 3М поддерживает только векторы длиной 64 бита). Есть еще оно ограничение - операции, преобразуемые в мультимедийные, обязательно принадлежат одному и тому же линейному участку.

2.2. Упрощенный алгоритм.

Мы представляем более простой алгоритм, который не имеет большей части указанных недостатков общего алгоритма. Он прост в реализации и позволяет достичь высокой производительности для достаточно широкого круга реальных мультимедийных приложений.

В статье [2] описаны общие требования к коду и ограничения на циклы, которые можно подвергнуть векторизации. В этом разделе мы обсудим требования к свойствам цикла с точки зрения нашего алгоритма, а также сам алгоритм, позволяющий генерировать векторные операции в большинстве случаев, когда применение векторных операций вообще оправданно.

Главная идея упрощенного алгоритма заключается в том, что после применения оптимизации раскрутки итераций (unroll) есть возможность сделать так, чтобы в каждой векторной команде исполнялось множество команд с разных итераций цикла. (Конечно, это возможно при условии отсутствия зависимости между упаковываемыми выражениями с различных итераций цикла). Таким образом, делая unroll цикла, мы изначально добиваемся изоморфизма

всех выражений цикла, и нет необходимости осуществлять очень сложный поиск изоморфных выражений.

Стоит сразу упомянуть о самом существенном ограничении предложенного нами алгоритма, накладываемом на оптимизируемый код.

Цикл, который мы пытаемся векторизировать, не должен быть подвергнут раскрутке изначально (т.е. программистом). В этом случае для того, чтобы применить наш алгоритм, нужно сделать reroll цикла [4], т.е. его обратную скрутку. А для этого имеет смысл применить алгоритм из статьи [1] – обнаружив изоморфные выражения, мы таким образом обнаруживаем различные итерации исходного (нераскрученного) цикла, после чего можно сделать скрутку. Так, наш алгоритм не сможет векторизовать из-за конфликтов операций обращения в память следующий цикл:

```
for ( i = 0; i < N; i+=2)
{ a[i] = b[i];
  a[i+1] = b[i+1]}
```

Но, найдя изоморфные выражения `a[i] = b[i]` и `a[i+1] = b[i+1]` и сделав reroll:

```
for ( i = 0; i < N; i++)
{ a[i] = b[i]}
```

мы приводим цикл к виду, в котором он легко векторизуется.

Предложенная схема в общем случае достаточно сложна, и в данной статье подробно не исследуется. Однако, развитие возможности векторизации в этом направлении - объект наших дальнейших исследований.

Рассмотрим цикл, в который входит множество арифметических выражений. Предлагается простой подход к оптимизации этого цикла, основанный на генерации операций мультимедийного расширения. Сначала цикл трансформируется путем раскрутки итераций (unroll), после чего каждое выражение имеет F копий, где F – число итераций unroll'а, а каждая операция выражения – F копий, т.е. операций того же наименования. После этого цикл готов к главному преобразованию - все F операций, соответствующие одной и той же операции исходного цикла, можно заменить одной векторной операцией. Операнды этих операций образуются, соответственно, последовательным объединением исходных операндов.

Разумеется, для того, чтобы избежать избыточного применения преобразования unroll, нужно заранее узнать, во-первых, возможно ли такое преобразование, а во-вторых, будет ли от этого из этого преобразования извлечена выгода.

Рассмотрим выражение

$R = x_1 \otimes x_2 \otimes x_3 ... \otimes x_N$

Какие требования мы предъявляем выражению, для того, чтобы установить, что оно может быть упаковано?

Во-первых, каждая операция, участвующая в этом выражении, должна иметь операцию - векторный аналог. Например, операция сложения (ADD) в архитектуре Эльбрус 3М имеет мелкоформатный аналог - упакованное сложение (PADD).

Во-вторых, на операцию записи в память (STORE), накладывается ограничение, связанное с особенностью архитектуры. Операция записи после применения преобразования раскрутки цикла (unroll) превращается в последовательность операций записи. Для получения выгоды от применяемой оптимизации достаточно объединить эту последовательность в одну операцию записи большего формата. В свою очередь, для такого объединения необходимо выполнение двух условий. Во-первых, адреса, по которым происходят эти записи, должны быть соседними. Во-вторых, адрес, по которому происходит первая запись последовательности, должен быть выровнен на количество байт, соответствующее новой операции записи. Аналогичные требования можно применить к операциям чтения из памяти (LOAD), поскольку к ним архитектура также предъявляет требование выравненности. Однако эти требования легко обойти с помощью предлагаемой техники. Ее суть заключается в том, что вместо одной невыровненной операции чтения формируется две операции выровненного чтения (рис. 1), которые считывают значения из соседних ячеек памяти. Затем, скомбинировав нужные части результатов двух операций считывания, получается требуемый вектор (рис 2).

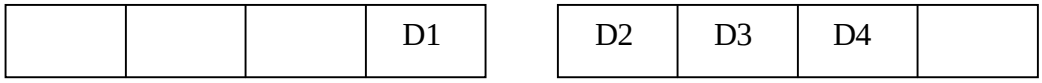


Рис. 1. Исходный вектор с базовым размером 2 байта не выровнен. Нужный вектор получается, если извлечь D1 из результата первого чтения, и D2 D3 D4 из результата второго.

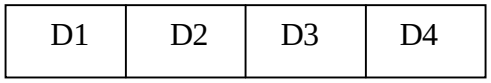


Рис. 2. Формируемый вектор

Отметим, что в теле цикла остается только одна операция чтения, т.к. для формирования вектора можно использовать младшие разряды результата чтения с предыдущей итерации цикла.

Такая техника работы с невыровненными операциями чтения позволяет избежать громоздкого малоэффективного решения, когда в цикле остаются все мелкоформатные операции чтения, а вектор формируется при помощи упаковки этих результатов, т.е. в теле цикла остается 2N-1 операций (рис. 3.2)), а при применении этой техники - 3 операции (рис. 3.3).

```
f (char * a, char *b, char *c){
for ( i = 0; i < M; i++ )
{a[i] = b[i+1] }  }
```

(1)

```
f (char * a, char *b, char *c)
for ( i = 0; i < M; i+=8 )
{ p1 = INS_FIELD b[i] b[i+1];
  p2 = INS_FIELD b[i+2] b[i+3];
  p3 = INS_FIELD b[i+4] b[i+5];
  p4 = INS_FIELD b[i+6] b[i+7];

  r1 = INS_FIELD p1 p2;
  r2 = INS_FIELD p3 p4;

  a[i…i+7] = INS_FIELD r1 r2;
}
```

(2)

```
f (char * a, char *b, char *c){
c1 = b[i…i+7];
for ( i = 0; i < M; i+=8 )
{ c2 = b[i+8…i+15];
  a[i…i+7] = INS_FIELD c1[1..7] c2[0];
  c1 = c2;
}  }
```

(3)

Рис. 3. Пример работы с чтением по невыровненному адресу.
(1) Исходный цикл, чтение из массива b не выровнено на 8 байт.
(2) После применения преобразования unroll на N=8 итераций и объединения выровненной записи в массив a.(INS_FIELD – операция вставки битового поля. Здесь при определении p1…p4 применяется вставка двух байтовых полей в одно двухбайтовое, при определении r1 и r2 – вставка двух двухбайтовых полей в четырехбайтовое, а при определении a[i…i+7] – вставка двух четырехбайтовых полей в восьмибайтовое).
(3) Цикл после применения изложенной техники работы с невыровненным чтением. В операции INS_FIELD вставляются младшие 7 байт c2 и один старший байт c1.

В-третьих, преобразование должно быть эффективным, т.е. при его применении динамическое количество исполненных инструкций должно быть меньше, чем без преобразования. Оценка эффективности такого преобразования является очень сложной задачей – мы ограничились простой эвристикой, учитывающей относительный вес количества упаковываемых операций в цикле. Если количество операций, которые могут быть векторизованы, составляют в цикле лишь небольшую часть, скорее всего преобразование будет неоправданно, т.к. тело цикла разрастется из-за оптимизации unroll, а ставшуюся (невекторизованную) часть цикла будет в дальнейшем сложно оптимизировать. Поэтому мы применяем следующую эвристику: если доля векторизуемых операций в исходном цикле – половина и более, значит, цикл будет векторизован.

Если доля векторизуемых операций невелика, перед преобразованием unroll имеет смысл произвести другое преобразование, называемое loop distribution. Цикл разделяется на векторную и невектоную часть, а затем векторная векторизуется. Например, цикл

```
for ( i = 0; i < N; i++)
{ c[i] = a[i] + d;
b[i] = f( a[i]) }
```

можно преобразовать в два цикла

```
for ( i = 0; i < N; i++)
{ c[i] = a[i] + d; }

for ( i = 0; i < N; i++)
{b[i] = f( a[i]) }
```

и после этого векторизовать первый цикл. Разумеется, для корректности такого преобразования необходимо, чтобы между двумя разделяемыми частями цикла не было зависимостей.

В общем случае переход от скалярной к векторной форме каждого выражения связан с некоторыми «накладными расходами» - необходимостью упаковки/распаковки каждой операции, которая не может быть векторизована. В нашем алгоритме число операций упаковки/распаковки сведено к минимуму – мы не векторизуем выражения, в которых участвуют невекторизуемые выражения. Единственным исключением являются аргументы

операций, являющиеся инвариантами цикла. Упаковка таких векторов – инвариантов проводится в предцикле[1].

В-четвертых, векторизованные операции обращения в память не должны быть зависимыми – иначе векторизация выражения может привести к конфликту.

Допустим, принято решение о том, что к данному циклу будет применена упаковка. Нужно узнать, сколько итераций unroll нужно применить. Предпочтительно исходить из тех соображений, что операции самого мелкого формата, встречающегося в данном цикле, должны быть упакованы в векторную операцию размера 64 бита, т.е. unroll должен иметь множитель F=64/size, где size - этот формат.

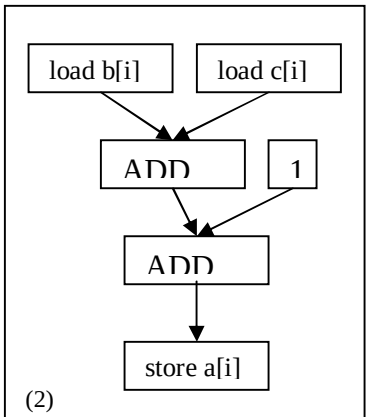
Как мы уже упоминали, после того, как сделан unroll, каждая операция имеет F-1 копий, а каждое выражение также имеет копии в виде F-1 деревьев выражений с вершиной – мелкоформатным STORE'ом. Теперь нет необходимости ни находить все множества операций, которые преобразовываются в одну векторную, ни как-то отдельно отмечать операции, которые должны быть упакованы - достаточно запомнить вершину дерева каждого выражения. Вместо каждой вершины дерева - мелкоформатного STORE'a – и всех его копий создается операция STORE общего размера. Далее, идя вверх по исходному дереву выражения, начиная от исходного STORE'a, можно просто генерировать векторную операцию, соответствующую исходной операции. Удалив все деревья выражений, вершинами которых являются исходный STORE и его копии, мы получаем полностью преобразованный цикл.

Отметим некоторые другие особенности алгоритма. Если аргумент операции - константа, то аргументом новой векторной операции будет вектор - константа, каждым элементом которого является константа - аргумент неупакованной операции. Руководствуясь тем же принципом, мы работаем с аргументами операции, являющимися инвариантом данного цикла. Аргумент векторной операции формируется в предцикле - в нем генерируется инициализирующий код, помещающий в каждый элемент формируемого вектора аргумент неупакованной операции.

На рис. 4 и 5. приведен пример оптимизации простейшего цикла в предположении, что компилятор в процессе трансляции может определить информацию о выравненности всех векторов на границу 64-разрядного формата. В результате векторизации выполнение такого цикла может быть ускорено почти в 4 раза.

```
f (short * a, short *b, short *c){
for ( i = 0; i < M; i++ )
{
   a[i] = b[i] +c[i]+1
}
}
```

(1)



(2)

[1] *Предцикл* – узел управляющего графа. Управление от предцикла всегда переходит непосредственно к самому циклу.

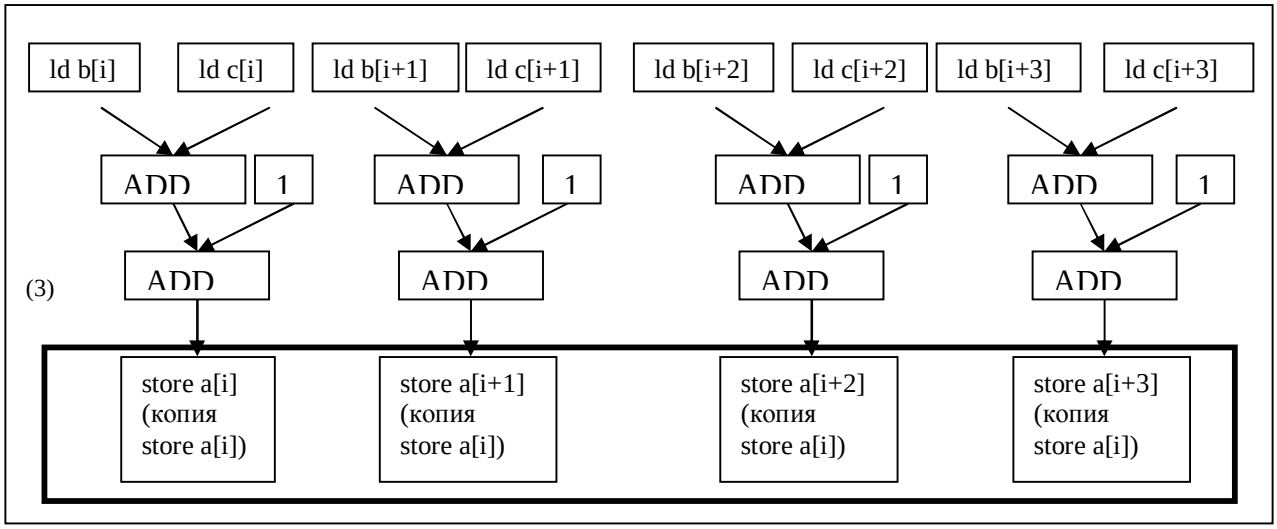


Рис. 4.Иллюстрирация работы оптимизации**.**
(1) Исходный пример на языке Си. (2) Дерево выражения, составляющее тело цикла. (3) Тело цикла после преобразования unroll. Операции записи заносятся в таблицу, при этом сохраняется информация об операции, Ё.

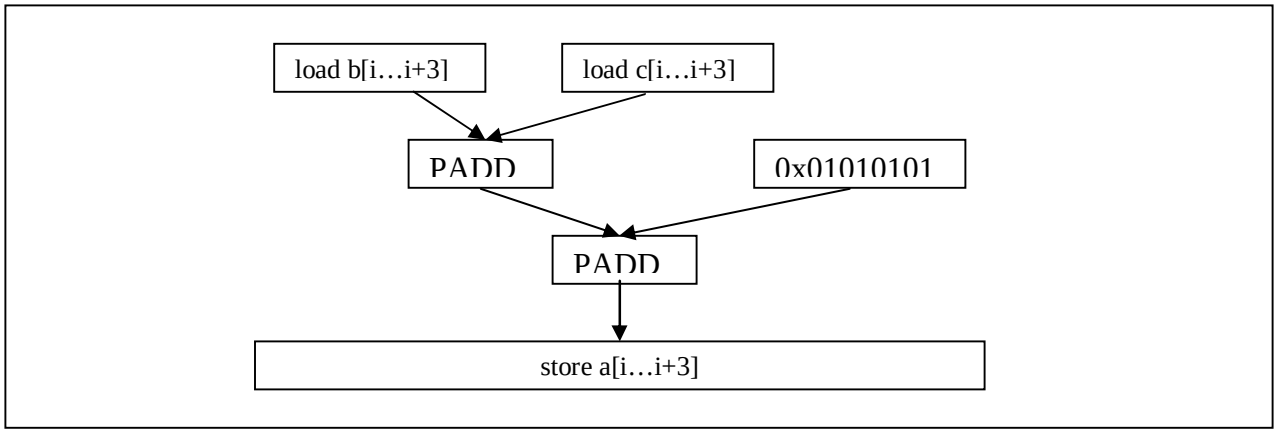


Рис. 5. Окончательный вид тела цикла после преобразования.

Описание алгоритма.

1. **Принятие решения о векторизации цикла**(loop)
2. для всех операций store цикла loop
3. {
4. *количество пакуемых операций в выражении* = 0;
5. если (**подсчет количества пакуемых операций в выражении**(store) == возможно упаковать)
6. {
7. *количество пакуемых операций в цикле* += *количество пакуемых операций в выражении*:
8. }
9. если (2**количество пакуемых операций в цикле* ≥ *общее количество операций в цикле)*
10. {вернуть (*векторизовать цикл*)}иначе{ вернуть (*не векторизовывать цикл*) }
11.
12. }

```
13. подсчет количества пакуемых операций в выражении( operation)
14. {
15.   если (operation – операция чтения)
16.   {
17.     если (operation выровнена &&
18.           operation не конфликтует с другими операциями чтения/записи)
19.     {
20.       количество пакуемых операций в выражении++;
21.       вернуть (невозможно)
22.     }
23.
24.   } иначе если (операция operation пакуется)
25.     для всех операций - operation2 аргументов operation (:operation2∈ циклу)
26.     {
27.       количество пакуемых операций в выражении++;
28.       подсчет количества пакуемых операций в выражении( operation2)
29.     }
30.   иначе
31.   {
32.     вернуть (невозможно)
33.   }
34. внести все операции записи в таблицу векторизуемых выражений.
35. вернуть (возможно)
36. }
37.
38. При преобразовании unroll.( цикл)
39. {
40.    копировать каждую операцию operation цикла F-1 раз.
41.    если(operation – операция записи &&
42.          operation входит в таблицу векторизуемых выражений )
43.    {
44.       внести в таблицу копию операции operation(копия operation->operation);
45.    }
46. }
47.
48. для всех векторов операций записи
49. {
50.    для всех членов вектора()
51.    {
52.       если (член вектора - операция записи входит в таблицу &&
53.             является коией первого член вектора ) == ложь
54.       {
55.         вектор не подходит;
56.         выход из цикла;
57.       }
58.    }
59.
60.    если( вектор подходит)
61.    {
62.       генерировать упакованное выражение(чтение - первый член вектора);
63.       для всех членов вектора()
64.       {
65.         стереть выражение, начиная от члена вектора;
66.       }
67.    }
68. }
69.
70. генерировать упакованное выражение( операция)
71. {
72.    если (operation – операция чтения)
73.    {
74.       создать операцию чтения общего размера
75.    } иначе если (operation – операция записи)
76.    {
77.       создать операцию записи общего размера
```

```
78.    } иначе
79.    {
80.     создать операцию - веторный аналог операции operation
81.    }
82.
83.  генерировать упакованное выражение( для всех операций - аргументов operation- операций);
84.  }
```

3. Распознавание сложных семантических конструкций, которые поддаются оптимизации с использованием векторных операций.

В рамках метода автоматической генерации векторных операций, в дополнение к алгоритму, описанному в предыдущей главе, мы предусмотрели возможность оптимизации отдельных случаев сложных семантических конструкций оптимизируемой программы. Эту возможность предоставляет наличие специальных операций, генерация которых позволяет уменьшить число операций в цикле, а порой – векторизовать циклы, которые без этого предварительного преобразования кажутся невекторизуемыми.

Дело в том, что обычно в состав системы команд мультимедийного расширения процессора входит набор специальных векторных операций, работа которых семантически эквивалентна работе нескольких обычных операций. В архитектуре Эльбрус 3М такими операциями являются группы операций AVG (векторное среднее арифметическое двух векторных операндов), MIN и MAX (большее и меньшее из двух векторных операндов), ADDS (сложение двух векторных операндов с сатурацией результата[1]), SUBS - (вычитание двух векторных операндов с сатурацией). Назовем такие операции *универсальными*.

В рамках разработанной нами схемы описываемая в данной главе подоптимизация имеет следующее место. Мы ищем в программе фрагменты, которые можно представить одной универсальной операцией, перед тем, как осуществить основную часть оптимизации. Таким образом, вся оптимизация состоит из двух этапов.

1) На первом этапе в промежуточном представлении программы распознается семантическая конструкция, соответствующая одной из универсальных операций. Эта конструкция меняется на соответствующую ей универсальную операцию.

2) На втором этапе применяются метод, описанный в **главе 2** - используется свойство векторности этих операций.

Вообще говоря, первый пункт является самостоятельной оптимизацией - если по какой-то причине второй пункт (т.е. векторизация цикла) не будет выполнен, сама замена нескольких операций на одну уже достаточно эффективна. В таком случае задействуется только самый младший элемент векторного операнда.

Понятно, что для корректности проведения подоптимизации нужно, чтобы диапазон значений операндов был представим количеством бит базового размера операнда генерируемой операции. Для установления этого факта используется информация, предоставляемая анализом диапазонов операндов [5, 10, 9].

---

[1] т.е. если значение результата вышло за границы определенного диапазона, результату присваивается значение границы этого диапазона.

Мы используем достаточно простую схему автоматического распознавания подходящих конструкций, которая, тем не менее, позволяет различить семантическую суть конструкции вне зависимости от способа ее записи. Например, при распознавании знаковой сатурации, т.е. конструкции, подобной следующей:

```
S = ( a > N ) ? (N) : ((a<= -N) ? (-N) : (a)))
```

мы принимали во внимание тот факт, что все конструкции, семантически описывающие сведение *a* к диапазону [-N, N-1], имеют похожую структуру. Все варианты сводятся к различному порядку операций присвоения границ диапазонов и к различным комбинациям проверок на строгое/нестрогое равенство значениям границ - например, N≥a или N+1>a. Аналогично будет распознана знаковая сатурация в другой записи:

```
if ( a > N )  S= N;
  else if ( S<= -N)
              S =-N;
        else S = a;
```

Такая конструкция семантически полностью эквивалентна сатурации в предыдущей записи, и этот факт легко устанавливается при анализе промежуточного представления программы, на котором мы работаем.

(Есть еще один способ записать беззнаковую сатурацию - например, для сведения *a* к диапазону [0, 255], можно произвести следующие действия:

```
S = ( a && 0xFF );
if (S != a)
  if ( a < 0 )   {S = 0;}
           else {S = 0xFF;}
```

Мы распознаем такую (или подобную) конструкцию специальном образом, при этом также рассматривая различные границы и диапазоны.)

После того, как мы распознали основную структуру сатурации, не составляет труда узнать, какая именно это сатурация (знаковая она или беззнаковая, к какому диапазону значений она сводит результат), подобрать соответствующую векторную операцию и сгенерировать ее. Все эти действия делаются автоматически в рамках нашего метода.

Аналогично распознаются и генерируются конструкции, соответствующие другим группам универсальных операций.

Описанное в этой главе преобразование, фактически, сводит определенные фрагменты программы, содержащие условные разветвления, к линейным участкам. Понятно, что эта оптимизация возможна лишь в отдельных случаях, когда семантика этого фрагмента программы совпадает с семантикой какой-либо имеющейся в СК операции. В тех же случаях, когда такой операции нет, или это невозможно, по какой-то другой причине, при векторизации необходимо обработать операции сравнения отдельно. Именно этому посвящена следующая глава.

4. Преобразование, основанное на свойствах операций векторного сравнения.

4.1. Операция векторного сравнения.

Существенная часть выражений не "умещается" в рамках линейного участка. Прежде всего, затруднение вызывает случай, когда значение какой-либо векторизуемой операции потребляется операцией сравнения, после чего происходит условный переход, зависящий от результата этого сравнения. Векторизовать циклы, в состав которых входят в такие выражения, очень сложно. В данной главе мы рассказываем об общих принципах векторизации таких циклов, а также приводим примеры преобразования таких циклов в векторную форму. Общий алгоритм векторизации, использующий векторные сравнения, в данный момент разрабатывается нами, и будет опубликован в наших дальнейших работах. Некоторые же частные случаи этого алгоритма описываются в данной главе.

Итак, в числе прочих векторных операций, в СК Эльбрус 3М, есть операции работы с мелкоформатными сравнениями. Их работа аналогична работе арифметических векторных операций: они обрабатывают те же два аргумента - вектора, и пара соответствующих элементов сравниваются. В качестве результата получается вектор, i–й элемент которого заполнен битами – единицами, если результат i-го сравнения элементов - "истина", либо битами-нулями, если соответствующие сравнение имеет результатом "ложь".

Случаи, когда приходится использовать операции мелкоформатного сравнения в цикле, можно разделить на два основных.

1) Сравнения, когда при одном из исходов управление выходит из цикла.

2) Сравнения, когда при любом исходе управление остается в цикле.

Эти случаи мы обрабатываем в рамках нашего метода по-разному. Остановимся на каждом из них более подробно.

4.2. Обработка сравнения, при одном из исходов которого управление выходит из цикла.

В библиотеках, и в прочих приложениях, работающих, например, со строками, часто встречаются циклы с боковыми выходами (под *боковым* выходом из цикла мы подразумеваем выход, осуществляющийся не по счетчику цикла). В качестве примера таких циклах можно привести такие. Вычисление длины строки (strlen).

```
for(i = 0; ; i++)
{
  if (!src[i]) break;
}
```

Другой пример – сравнение двух строк (strcmp).

```
for(i = 0; ; i++)
{
  if (src1[i] != src2[i]) break;
  if (!src[i]) break;
}
```

Векторизация таких циклов, использующая особенности операций векторного сравнения, может дать хороший прирост производительности. Каким образом можно произвести подобное преобразование?

После того, как сделана раскрутка цикла на F итераций, у цикла образуется F боковых выходов. Для того, чтобы векторизовать такой цикл, все боковые выходы нужно свести к одному, а все постциклы, которые соответствуют исходным боковым выходам - также к одному постиклу. При этом все скалярные операции сравнения, по которым происходит боковой выход, превращаются в одну векторную, а результат этого векторного сравнения - в вектор, в котором нужно обнаружить элемент из нулей (если выход происходит при неисполнении условия сравнения элементов), и выйти из цикла.

Поскольку по боковому выходу в преобразованном цикле управление может из цикла раз в F итераций (где - F-число объединяемых выходов из цикла), то реальное число исполненных итераций исходного цикла кратно F. Поэтому после выхода их цикла может понадобиться восстановить точное значение цикловых переменных (если их значение потребляется после цикла), которое было бы при выходе из не преобразованного цикла. В общем случае в новом боковом постцикле необходимо создать "досчетный" цикл, в котором бы корректировалось значение нужных в дальнейшем цикловых переменных, соответствующих реальному числу итераций исходного цикла. В этом "досчетном" цикле считается значение цикловых переменных для исходных итераций внутри диапазона [N*F…(N+1)*F) (где N - число итераций преобразованного цикла), в зависимости от того, на какой исходной итерации внутри новой итерации произошел выход, т.е. от положения нулей в векторе, по которому управление вышло из цикла.

Главное ограничение на описываемое в данном разделе преобразование - отсутствие пробочных эффектов в цикле. Так, например, если в цикле с боковым выходом есть операция записи, то боковые выходы нельзя объединить в один, т.к. в этом случае могут выполнятся не предусмотренные исходной семантика цикла записи, и будут происходить изменения контекста, которые не должны били случиться в исходном цикле.

Нами были реализованы отдельные частные случаи алгоритма векторизации операции сравнения для бокового выхода из цикла. Прекрасным примером такого преобразования служит проведенная нами оптимизация основного цикла из задачи 023.eqntott пакета SPECint92. Этот цикл исполняется много раз, так что на него приходится более половины динамически исполняемых инструкций при запуске на исполнение задачи eqntott.

Исходный цикл выглядит следующим образом (рис. 6).

```
for ( i = 0; i < ninputs; i++ )
{
  aa = a[0]->ptand[i];
  bb = b[0]->ptand[i];
  if (aa == 2) aa = 0;
  if (bb == 2) bb = 0;
  if (aa != bb)
  {
    if (aa < bb) { return (-1);}
              else { return (1);}
  }
}
```

Рис. 6. Исходный цикл.

К этому циклу применяется ряд оптимизаций, в т.ч. “вынесение векторного инварианта” – оптимизация, которая выносит инвариантную часть вложенного цикла перед охватывающим. Возможность применения оптимизации основана на результате межпроцедурного анализа. Перед охватывающим циклом здесь возникает инициализирующий цикл (рис. 7).

```
for ( i = 0; i < M; i++ )
{
   if (b[0]->ptand[i]  == 2 )
   {
      bb_[i] = 0;
   } else
   {
      bb_1[i] = b[0]->ptand[i];
   };
}
```

Рис. 7. Инициализирующий цикл.

Рассматриваемый цикл преобразуется в следующий (рис. 8).

```
for ( i = 0; i < ninputs; i++ )
{
  aa = a[0]->ptand[i];
  bb = bb_[i];
  if (aa == 2) aa = 0;
  if (aa != bb)
  {
    goto exit2;
  }
}
…
exit2
   if (aa < bb) { return (-1);}
            else { return (1);}
```

Рис. 8. Преобразованный цикл.

Семантика этого фрагмента кода сводится к трем операциям. Первая – сравнение aa с двойкой, вторая – присвоение aa значения 0, если сравнение выполнилось, и третья – выход из цикла, если значения aa и bb не равны. Кроме того, для постцикла[2] необходимо сохранить сами значения aa и bb.

Идея преобразования заключается в том, чтобы векторизовать каждую из этих операций. Это делается следующим образом.

- Раскручивание тела цикла на 4 итерации (преобразование unroll).

- Считывание значений aa и bb – это две операции LOAD, которые читают из памяти 2 байта. Чтение происходит из массивов ptand[i] и bb_[i], элементы которых, естественно, располагаются в памяти подряд. Пользуясь этим, 4 операции LOAD двух байт меняется на одну операцию LOAD восьми байт.

- Сравнение aa с двойкой и присваивание aa нулю в случае выполнения сравнения заменяется парой векторных операций. Первая – векторное сравнение вектора из четырех членов массива, каждый элемент вектора сравнивается с двойкой. В

[2] *Постцикл* - – узел управляющего графа. Управление от цикла всегда переходит непосредственно к постциклу.

шестнадцатеричном виде такой второй аргумент выглядит как 0x2000200020002. (рис. 9)

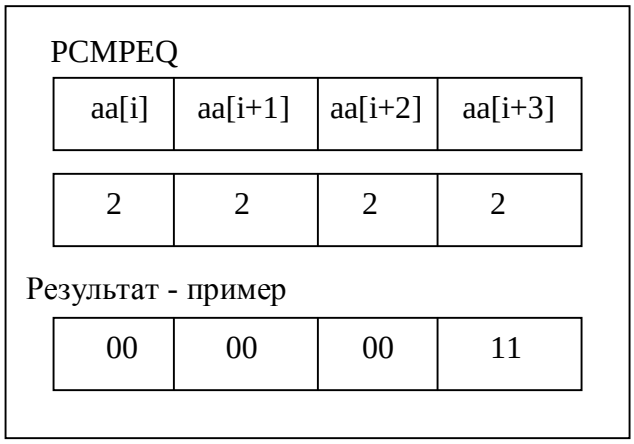


Рис. 9. Векторная опенрация PCMPEQ и пример ее работы (под 0 в «результате» обозначен байт, заполненный битами-нулями, а под 1 – байт, заполненный битами-нулями).

Вторая операция – ANDN того же вектора aa с результатом PCMPEQ.

Рис. 10. Пример работы операции ANDN

Таким образом, результатом работы этих двух векторных операций будет вектор, каждый член которого – или aa[i], или в нужном случае 0, что соответствует семантике исходного цикла (рис. 10).

- Сравнение aa и bb и выход в случае неравенства преобразовывается в простое сравнение векторов из четырех aa и bb. Единственный побочный эффект такого преобразования – то, что неизвестно, какие именно из четырех aa и bb не совпали. Для этого в постцикле добавляется разбор этих векторов с целью найти первые несовпадающие элементы, а затем сравнить их, как это делается в исходном постцикле. Здесь можно применить оптимизацию, основанную на том, что в векторе все элементы совпадают вплоть до нужных, которые затем сравниваются. Значит, поменяв элементы векторов таким образом, что в начале вектора идут более ранние,

т.е. совпадающие элементы, а значит, сравнив эти векторы как целые числа, получим тот же ответ, что и от сравнения первых несовпадающих элементов.

Таким образом, тело цикла можно представить в следующем виде (рис. 11).

```
for ( i = 0; (i < ninputs/4); i++ )
{
  aa[0…3] = a[0]->ptand[i…i+3];
  bb[0…3] = bb_[i…i+3];
  cc = CMPEQ(aa[0…3], 0x2000200020002);
  aa[0…3] &=  ~cc ;
  if (aa[0…3] != bb[0…3] )
  {
    goto exit2;
  }
}
…
exit2:
aa[0…3] = PSHUFH (aa[0…3]);
bb[0…3] = PSHUFH (bb[0…3]);
if (aa[0…3]  < bb[0…3]) { return (-1);}
                        else { return (1);}
```

Рис. 11. Полностью преобразованный цикл. PSHUFH - операция, меняющая порядок элементов вектора..

Таким образом, в теле цикла остается 5 операций на 4 итерации исходного цикла, или 1.25 операций на одну итерацию.

Приведенный пример является частным случаем преобразования цикла с боковым выходом. В будущем будет разработан общий алгоритм оптимизации таких циклов.

4.3. Обработка сравнения, при любом исходе которого управление остается в цикле

В этом случае преобразование должно фактически свести условное разветвление в предикатную форму, т.е. переместить вычисления, находящиеся на обеих ветках условного разветвления, на один линейный участок, но так, чтобы реально вычислялись только выражения из ветки, которая должна была выполниться до преобразования. Суть такого преобразования в векторном случае – наложить маску, получаемую от операции векторного сравнения на результаты вычислений, производимых в обеих ветках разветвления.

Поясним эту мысль простым примером. Путь есть цикл, в котором в байтовый массив b[i] пересылается абсолютное значение элементов байтового массива a[i]:

```
for ( i = 0; i < N; i++)
{ if (a[i] > 0) b[i] = a[i];
          else b[i] = -a[i];}
```

После преобразования unroll нужно, чтобы в каждый элемент b[i] записывался или a[i] или -a[i] в зависимости от его знака. Результат векторной операции PCMGT a[i…i+7], 0 – вектор-маска, элемент которой единицы, если a[i] > 0, или нули – в противном случае. В соответствие с логикой нашего метода, эту маску мы накладываем на вектор a[i…i+7], а инвертированную маску – на -a[i…i+7] (т.е. результат векторного вычитания PSUB 0,

a[i…i+7]). Сделав побитовое «или» результатам наложенных масок, мы получаем искомый вектор, который можно записать в b[i…i+7]:

```
for ( i = 0; i < N; i+=8)
{
  mask = PCMPGT a[i…i+7], 0
  i_mask = ~mask;
  minus_a[i…i+7] = 0, a[i…i+7];
  v1[i…i+7] = a[i…i+7] & mask;
  v2[i…i+7] = minus_a[i…i+7] & i_mask;
  b[i…i+7] = v1[i…i+7] | v2[i…i+7];
}
```

Таким образом, на 8 итераций цикла мы имеем 6 операций, или 0.75 операций на итерацию, вместо исходных 3 операций на итерацию (CMP и две операции MOV).

Руководствуясь подобной логикой, преобразуются и другие случаи «внутрициклового» сравнения. Здесь мы рассмотрели общую логику такого преобразования и отдельный частный случай, а в дальнейшем будет разработан и опубликован полный алгоритм векторизации циклов со сравнением.

5. Результаты.

В этой главе мы опишем экспериментальные результаты.

5.1. Уменьшение количества динамически исполняемых инструкций.

Основной показатель эффективности любой оптимизации - это то, на сколько уменьшилось количество динамически исполняемых инструкций в результате ее применения. Здесь мы приводим результаты на примере группы тестов, каждый из которых моделирует поведение какой-либо реальный мультимедийной функции или небольшого приложения. (Получены результаты для следующих групп тестов: «copy&set» – копирование и инициализация массивов, «arith&logic» вычисление арифметических и логических выражений, «shift» - выражения со сдвигом, «min/max» - запись в массив минимума/максимума различных величин, «saturation» вычисление суммы и разности величин с насыщение). В таблице 1 приведены коэффициенты уменьшения количество динамически исполняемых инструкций в результате генерации векторных операций. Цифры приведены для различных групп тестов для базового размера аргумента 8, 16 и 32 бита.

Таблица 1. Коэффициент уменьшения количества динамически исполняемых инструкций в результате генерации векторных операций для различных групп тестов.

Имя теста	8 бит	16 бит	32 бита
copy&set	15.79	7.94	3.98
arith&logic	7.92	3.98	1.99
shift	3.98	3.96	1.98

min/max	6.39	3.19	-
saturation	4.09	2.08	-

5.2. Количество применений оптимизации.

Косвенный показатель эффективности оптимизации - это количество случаев, в которых она применяется. Хотя такая статистика непосредственно не обнаруживает вклад данной оптимизации в уменьшение количества динамически исполняемых инструкций (что является главным критерием для любой оптимизации), существует прямая зависимость между количеством случаев применения оптимизации и ее эффективностью, особенно если хорошо настроены эвристики применения оптимизации.

Далее приведена таблица количества раз, которое применилась оптимизация векторизации цикла и замены сложной семантической конструкции на универсальную операцию на примере пакета SpecINT95 (таблица 2).

Таблица 2. Количество раз, которое применилась оптимизация векторизации цикла и замены сложной семантической конструкции на универсальную операцию примере пакета SpecINT95

Тест	Количество применений
099.go	96
124.mksim	32
126.gcc	2
129.compress	14
132.ijpeg	4
134.perl	3

6. Заключение.

В статье описано развитие метода оптимизации циклов путем генерации векторных операций. В рамках данного метода, разработан алгоритм, позволяющий векторизовать произвольные выражения в цикле, суть которого заключается в запоминании вершины дерева каждого из выражений при раскрутке цикла, а затем, при проходе по единственной копии исходного выражения - генерация векторных операций, соответствующих каждой из обычных операций. В качестве подоптимизации общего алгоритма, разработаны методы распознавание сложных семантических конструкций, описаны способы их автоматической оптимизации с использованием векторных операций, базирующиеся на выделении основных семантических компонент каждой из конструкций. Другой целью описываемого метода является преобразование, основанное на свойствах операций векторного сравнения, позволяющее очень эффективно ускорять определенные группы циклов. В статье описаны общие принципы оптимизации циклов, в состав которых входит векторизуемая операция сравнения, и подробно разобраны примеры таких преобразований. Приведены результаты.

8. Ссылки.

1. S. Larsen, S.Amarasinghe. Exploiting Superword Parallelism with Multimedia Instruction Sets. // PLDI 2000, Vancouver, British Columbia, Canada.
2. G. Ren, P. Wu, D.Padua. A Preliminary Study of Multimedia Applications for Multimedia Extensions. // 16th Workshop on Languages and Compilers for Parallel Computing, 2003.
3. Babayan B. A. E2k Technology and Implementation. // Proceedings of the Euro-Par 2000 - Parallel Processing: 6th International. - V. 1900/2000. - January, 2000. - P. 18-21.
4. Muchnik S.S. Advanced Compiler Design Implementation. // 17.4. Loop Unrolling, pp 559-562; Chapter 8. Data Flow Analysis; 20.4.2 Loop Transformations, pp 689- 69. // Morgan Kaufmann Publishers, 1997.
5. W. Blume, R. Eigenmann. Symbolic Range Propagation. // Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1994.
6. K. Dieffendorf. The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee // Microprocessor Report, V. 13, No 2. February 15, 1999. P. 1-7.
7. A. Pelag, U. Weiser. MMX Technology Extension to Intel Architecture. // IEEE Micro, 16(4):42-50, Aug 1996.
8. R. Lee. Subword Parallelism with MAX-2. // IEEE Micro, 16(4):51-59, Aug 1996.
9. M. Stephenson, J.Babb, S.Amarasinghe. Bitwidth Analysis with Application to Silicon Complilation. // In Proceeding of SIGLAN ’00 Conferenceon Programming Language Design and Implementation, Vancouver, BC, June 2000.
10. W. Blume, R. Eigenmann. Demand-Driven, Symbolic Range Propagation. // Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1995.