

# Метод использования мелкоформатных векторных операций в оптимизирующем компиляторе

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

**Аннотация.** В последнее десятилетие в состав новых процессоров, как правило, входят мультимедийные расширения. В их основу положен принцип SIMD - single instruction - multiple data (одна инструкция - множественные данные), а соответствующие операции называются векторными. Статья посвящена методам автоматического выявления в программе векторизуемых вычислений и генерации соответствующих векторных операций для широкого класса векторизуемых циклов, включая циклы со сложным управлением, которые были разработаны и опробованы в рамках оптимизирующего компилятора для архитектуры Эльбрус.

## Введение

Впервые мультимедийное 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, SSE2 и SSE3 в семействе процессоров x86 (IE-32) компании Intel, 3DNow! AMD [2, 7, 8]. В передовом проекте Intel IA-64 [11] векторные операции поддерживались изначально, что свидетельствует о значении, придаваемом им создателями этой архитектуры.

В использовании аппаратных ресурсов, предоставляемых мультимедийными расширениями, все еще преобладает традиционный подход, когда соответствующие операции расширенной системы команд попадают в код, будучи вставлены в него 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.

## 1. Генерация векторных операций для произвольных выражений

### 1.1. Общий алгоритм

Алгоритм генерации векторных операций при векторизации цикла, в самом общем виде описанный в статье [1], содержит множество шагов. После обязательных оптимизаций (внутренней раскрутке цикла, unroll [4], и анализа операций обращения в память на предмет выравненности) ищутся пары, состоящие из операции чтения и операции записи, которые обращаются к соседним участкам памяти. Затем совершается проход по графу определений/использований [4], и операции доступа в память дополняются другими операциями, в результате чего получаются пары выражений, которые комбинируются с другими парами так, чтобы они образовали большие группы выражений. Таким образом, формируются группы *изоморфных* выражений (то есть выражений, в которых одноименные операции идут в одинаковом порядке). После этого генерируются векторные операции вместо соответствующих друг другу одноименных операций в группе.

Общий алгоритм универсален, но имеет определенные недостатки – он сложен и ориентирован на длинные векторы (в частности, этот алгоритм показывает лучшие результаты для векторов длиной 512 и 1024 бит, между тем как архитектура Эльбрус поддерживает только векторы длиной 64 бита). Есть еще одно ограничение - операции, преобразуемые в векторные, обязательно принадлежат одному и тому же линейному участку.

## 1.2 Базовый алгоритм

Далее представляется алгоритм, названный базовым алгоритмом векторизации, который является более простым в реализации и позволяет достичь высокой производительности для достаточно широкого круга мультимедийных приложений. Для базового алгоритма справедливы общие требования к оптимизируемым циклам, которые описаны в [2]. Его специфика состоит в том, что в результате оптимизации раскрутки итераций цикла (unroll) удается изначально добиться изоморфизма всех входящих в него выражений (нет необходимости осуществлять сложное построение изоморфных выражений, как в общем алгоритме) и, соответственно, исполнить одной векторной операцией множество операций с различных итераций цикла.

Рассмотрим цикл, в который входит множество арифметических выражений, включающих операции над элементами массивов, причем результаты этих выражений также записываются в элементы массивов. Предлагается простой подход к оптимизации этого цикла, основанный на генерации векторных операций. Сначала цикл трансформируется путем раскрутки итераций. Если  $F-1$  – число дополнительных итераций при раскрутке, то после этого каждое выражение и каждая входящая в него операция имеют  $F$  копий. В результате цикл готов к главному преобразованию - все  $F$  операций, соответствующие одной и той же операции исходного цикла, можно заменить одной векторной операцией. Ее операнды образуются последовательным объединением исходных операндов. Разумеется, для того, чтобы избежать избыточного применения преобразования unroll, нужно заранее узнать, возможно ли такое преобразование и будет ли оно выгодно.

Стоит упомянуть о самом существенном ограничении, накладываемом в этом случае на оптимизируемый код - цикл, подвергаемый векторизации, не должен быть раскручен изначально (программистом). Если это произошло, то следует выполнить его обратную скрутку, `reoll` [4], например, применив алгоритм поиска изоморфных выражений [1]. Если найдены изоморфные выражения, то обнаруживаются и различные итерации исходного (нераскрученного) цикла, после чего можно сделать скрутку. Так, в примере на Рис.1а алгоритм не сможет векторизовать цикл из-за конфликтов операций обращения в память. Но, найдя изоморфные выражения  $a[i] = b[i]$  и  $a[i+1] = b[i+1]$  и выполнив `reoll`, можно привести цикл к виду, в котором он легко векторизуется с помощью базового алгоритма (Рис 1б).

Предложенная схема в общем случае достаточно сложна, и в данной статье подробно не исследуется. Развитие возможностей векторизации в этом направлении является предметом наших дальнейших исследований.

## 1.3. Требования к упаковываемому выражению

Какие требования предъявляются к выражению, чтобы оно могло быть упаковано?

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

<pre>for ( i = 0; i &lt; N; i+=2) { a[i] = b[i];   a[i+1] = b[i+1] }</pre> <p>(а)</p>	<pre>for ( i = 0; i &lt; N; i++) { a[i] = b[i] }</pre> <p>(б)</p>
---	---

Рис. 1. Примеры циклов, подлежащих векторизации

- а) Цикл, который не векторизуется предлагаемым алгоритмом из-за конфликтов операций обращения в память  
 б) После нахождения изоморфных выражений и оптимизации `reoll` цикл сводится к виду, пригодному для векторизации

Во-вторых, на операцию записи в память (STORE) накладывается ограничение, связанное с особенностью архитектуры. Операция записи после применения преобразования 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: адреса, по которым происходят эти записи, должны быть соседними; адрес, по которому происходит первая запись последовательности, должен быть выравнен на количество байтов, соответствующее новой операции записи.

Последнее условие является ограничением реализации, хотя его можно ослабить, применив метод, аналогичный методу для невыравненных операций LOAD (см. ниже). Но при этом существенно возрастает количество вспомогательных операций, что снижает эффект от векторизации невыравненных записей. Использование аппаратной поддержки невыравненных записей также приводит к заметному снижению эффекта векторизации. Поэтому было выработано ограничение.

Аналогичными условиями можно ограничить применение операций чтения из памяти (LOAD), поскольку они также поддерживаются архитектурой при соблюдении выравненности. Однако эти условия легко обойти с помощью предлагаемой техники. Она состоит в том, что вместо одной невыравненной операции чтения формируются две операции выровненного чтения (Рис. 2а), которые считывают значения из соседних ячеек памяти. Затем, комбинируя нужные части результатов двух операций считывания, можно получить требуемый вектор (Рис 2 б). Отметим, что в теле цикла остается только одна операция чтения, так как для формирования вектора можно использовать младшие разряды результата чтения с предыдущей итерации цикла.

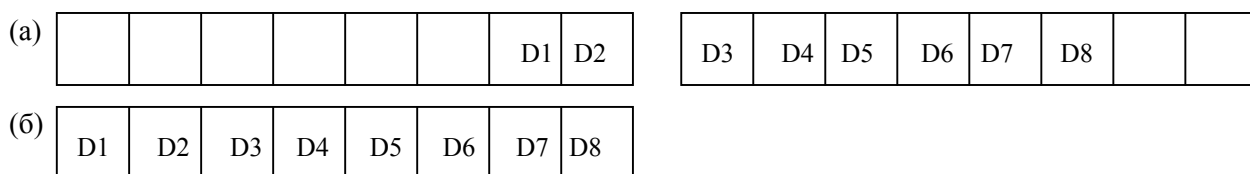


Рис. 2. Принцип выравнивания векторов для реализации операции записи

- а) Исходный вектор с размером элемента 1 байт не выравнен; нужный вектор получается, если извлечь D1 и D2 из результата первого чтения, и D3, D4, D4, D5, D6, D7 и D8 из результата второго
- б) Формируемый вектор

Такой метод позволяет избежать громоздкого малоэффективного решения, когда в цикле остаются все мелкоформатные операции чтения, а вектор формируется путем упаковки этих результатов. Тогда в теле цикла остается 2N-1 операций (Рис. 3б), в то время как при применении изложенной техники – только 3 операции (Рис. 3в).

В-третьих, преобразование должно быть эффективным, то есть в динамике количество исполненных инструкций должно быть меньше, чем без преобразования. Оценить эффективность такого преобразования сложно, поэтому описываемый алгоритм базируется на простой эвристике, учитывающей относительный вес количества упаковываемых операций в цикле. Если количество операций, которые могут быть векторизованы, составляют в цикле лишь небольшую часть, то преобразование, скорее всего, будет неоправданно, ибо тело цикла разрастется из-за оптимизации unroll, а оставшуюся (невекторизованную) часть цикла будет в дальнейшем сложно оптимизировать. Поэтому применяется следующая эвристика: если доля векторизуемых операций в исходном цикле составляет половину и более, то цикл будет векторизован.

Если доля векторизуемых операций невелика, то перед преобразованием unroll имеет смысл произвести другое преобразование, называемое loop distribution [12]. Цикл разделяется на векторную и не векторную части, а затем векторная векторизуется. Например, цикл на Рис 4(а) можно трансформировать в два цикла и после этого векторизовать первый из них (Рис. 4б). Разумеется, для корректности такого преобразования необходимо, чтобы между двумя разделяемыми частями цикла не было зависимостей.

<pre>f(char * a, char *b, char *c){ for ( i = 0; i &lt; M; i++) {a[i] = b[i+1] } }</pre> <p>(a)</p>	<pre>f(char * a, char *b, char *c) for ( i = 0; i &lt; 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; }</pre> <p>(б)</p>	<pre>f(char * a, char *b, char *c){ c1 = b[i...i+7]; for ( i = 0; i &lt; M; i+=8 ) {c2 = b[i+8...i+15];   a[i...i+7] = INS_FIELD c1[2..7] c2[0..1];   c1 = c2; } }</pre> <p>(в)</p>
---	---	---

Рис. 3. Пример работы с чтением по невыравненному адресу

- а) Исходный цикл, чтение из массива b не выравнено на 8 байт
- б) Цикл после применения преобразования unroll на 8 итераций и объединения выравненной 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)
- в) Цикл в результате применения изложенной техники. В операции INS\_FIELD вставляются младшие 6 байт c2 и два старших байта c1

В общем случае переход от скалярной к векторной форме каждого выражения связан с некоторыми расходами - упаковкой результата каждой не векторизуемой операции для дальнейшего использования в векторном выражении, а также - распаковкой аргументов для каждой не векторизуемой операции из векторного аргумента в скалярные. (Пример упаковки результатов операции чтения, не векторизуемой по причине невыравниваемости, приведен на Рис. 3б.).

<pre>for ( i = 0; i &lt; N; i++) { c[i] = a[i] + d;   b[i] = f( a[i] ) }</pre> <p>(a)</p>	<pre>for ( i = 0; i &lt; N; i++) { c[i] = a[i] + d; }  for ( i = 0; i &lt; N; i++) { b[i] = f( a[i] ) }</pre> <p>(б)</p>
---	--

Рис. 4. Пример преобразования loop distribution

- а) Цикл, в котором преобразование позволяет провести векторизацию
- б) Два цикла, полученные в результате преобразования. Теперь первый цикл можно векторизовать

Упаковка и распаковка могут сводиться к большому числу операций, так как количество дополнительных операций пропорционально F. В то же время в описываемом алгоритме число операций упаковки и распаковки сведено к минимуму – в нем не векторизуются только те выражения, в которых участвуют не векторизуемые операции. Исключением являются невыравненные операции чтения (об этом сказано выше), а также аргументы операций, являющиеся инвариантами цикла. Упаковка таких векторов – инвариантов проводится в предцикле<sup>1</sup> (в общем случае - в предциклах).

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

#### 1.4. Метод упаковки векторизуемых выражений

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

<sup>1</sup> Предцикл – узел управляющего графа. Управление от одного из предциклов всегда переходит непосредственно к самому циклу.

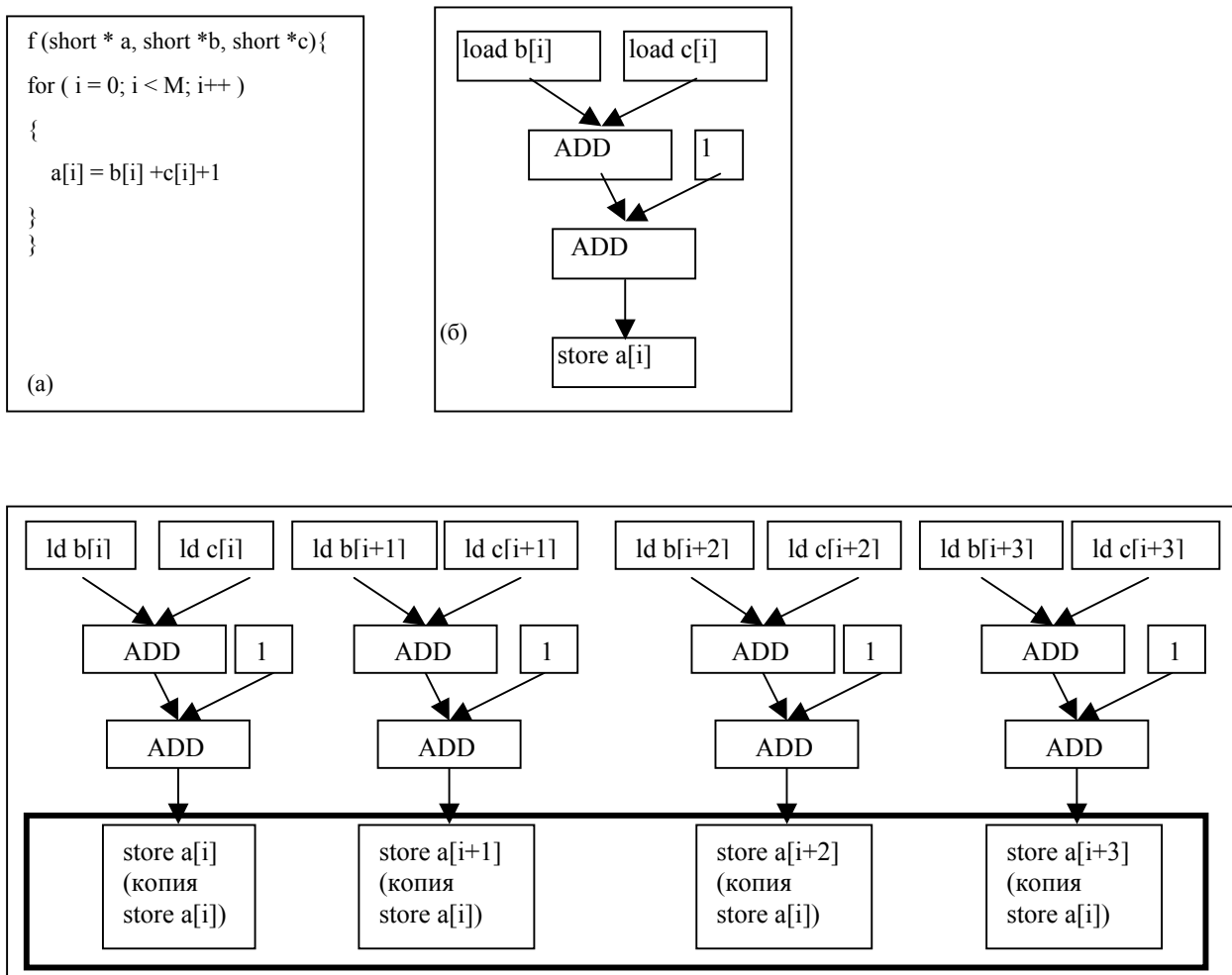


Рис. 5. Иллюстрация процесса оптимизации

- а) Исходный пример на языке Си
- б) Дерево выражения, составляющее тело цикла
- в) Тело цикла после преобразования unroll; операции записи заносятся в таблицу, при этом сохраняется информация о той операции записи, из которой произошла текущая операция записи

Как уже было упомянуто, после исполнения unroll каждая операция имеет Fкопий, а каждое выражение также имеет копии в виде F графов выражений с выделенной вершиной – мелкоформатной операцией STORE. Теперь нет необходимости ни находить все множества операций, которые преобразовываются в одну векторную, ни отдельно отмечать операции, которые должны быть упакованы - достаточно запомнить одну выделенную вершину графа каждого выражения, вследствие чего создается операция STORE общего размера. Далее, путем прохода вверх и вниз по исходному графу выражения, начиная от исходной операции STORE, можно просто генерировать векторную операцию, соответствующую исходной операции. Удалив все графы выражений, выделенными вершинами которых являются исходная операция STORE и ее копии, можно получить полностью преобразованный цикл.

Стоит отметить некоторые другие особенности алгоритма. Если аргумент операции - константа, то аргументом новой векторной операции будет вектор-константа, каждым элементом которого является константа-аргумент неупакованной операции. По тому же принципу выполняется работа с аргументами операции, являющимися инвариантом данного цикла. Аргумент векторной опера-

ции формируется в предцикле - в нем генерируется инициализирующий код, помещающий аргумент неупакованной операции в каждый элемент формируемого вектора.

На Рис. 5 и Рис. 6. приведен пример оптимизации простейшего цикла, выполняющего действия над 16-битными выражениями, в предположении, что компилятор в процессе трансляции может получить информацию о выравненности всех векторов на границу 64-разрядного формата. В результате векторизации выполнение такого цикла может быть ускорено почти в 4 раза.

Сам базовый алгоритм, включающий в себя принятие решения о векторизации, преобразование unroll в случае успешного решения и генерацию векторных выражений, приведен на Рис. 7.

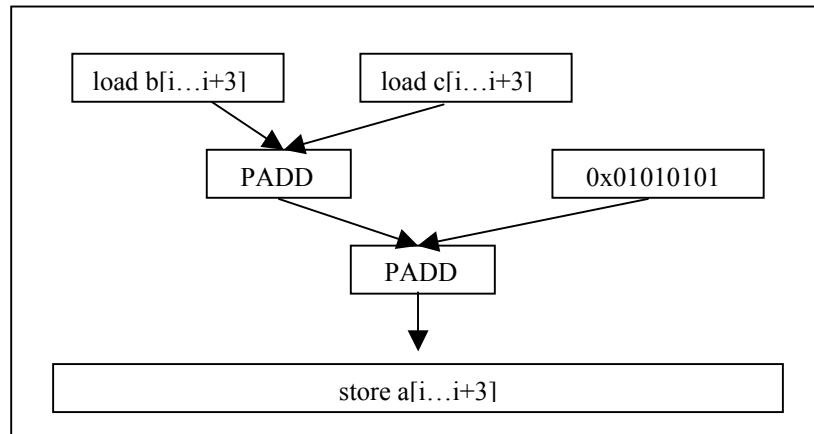


Рис. 6. Окончательный вид тела цикла после преобразования

<p><b>векторизация циклов</b></p> <pre> { for( всех самых вложенных циклов (loop) процедуры)   { принятие решения о векторизации цикла(loop);     if ( векторизовать цикл возможно)       { преобразование unroll(loop);         генерация векторных выражений(loop);       }   } } (a)         </pre>	<p><b>преобразование unroll( цикл)</b></p> <pre> {   копировать каждую операцию operation цикла F-1 раз;   if ( operation – операция записи &amp;&amp;       operation входит в таблицу       векторизуемых выражений )   {внести в таблицу копию операции    operation(копия operation-&gt;operation);} } (г)         </pre>
<p><b>принятие решения о векторизации цикла(loop)</b></p> <pre> { суммарное количество пакуемых операций в цикле loop = 0;   for( всех операций store цикла loop)     { if ( подсчет кол-ва пакуемых операций в выражении(       store)       == возможно упаковать)       { суммарное кол-во пакуемых операций в цикле loop +=         количество пакуемых операций в выражении; }       if ( 2* суммарное кол-во пакуемых операций в цикле loop ≥         общее количество операций в цикле)         { F = 64 / самый мелкий формат упак.-мой           операции записи в цикле;           внести все операции записи в таблицу           векторизуемых выражений;.           return (векторизовать цикл);         } else return (не векторизовывать цикл);       }     } } (б)         </pre>	<p><b>генерация векторных выражений(loop)</b></p> <pre> { for ( всех векторов операций записи )   { for ( всех членов вектора )     { if ( ( член вектора - операция записи ∈ таблице &amp;&amp;             является копией первого член вектора ) == ложь)       { вектор не подходит;         break;       }     }     if ( вектор подходит)       { генерировать упакованное выражение(операция         записи - первый член вектора);         for ( всех членов вектора )           стереть выражение, начиная от члена вектора;         }       }     } } (д)         </pre>
<p><b>подсчет кол-ва пакуемых операций в выражении( operation)</b></p> <pre> { количество пакуемых операций в выражении = 0;   if ( operation – операция чтения)     { if ( operation не конфликтует с др. операциями чте-       ния/записи)       количество пакуемых операций в выражении++;     }   }         </pre>	<p><b>генерировать упакованное выражение( operation)</b></p> <pre> {   if (operation – операция чтения)     создать операцию чтения общего размера;   else if (operation – операция записи)     создать операцию записи общего размера; }         </pre>

<pre> else return (невозможно); } else if (operation – операция записи) { if (operation выравнена &amp;&amp;     operation не конфликтует с др. операциями чтения/записи)     количество пакуемых операций в выражении ++;   else return (невозможно); } else if (операция operation пакуется)     количество пакуемых операций в выражении ++; else return (невозможно);  /* рекурсия */ for ( всех операций operation2 - определений и использований     операции operation (:operation2∈ циклу loop) {     подсчет количества пакуемых     операций в выражении( operation2);     if ( векторизация невозможна )         return (невозможно); }  return(возможно); } (в) </pre>	<pre> else     создать операцию - векторный     аналог операции operation;  /* рекурсия */ генерировать упакованное выражение( для всех операций - определений и использований operation); } (e) </pre>
--	---

Рис. 7. Базовый алгоритм векторизации циклов

- а) общий алгоритм
- б) алгоритм принятия решения о векторизации цикла - используется функция, приведенная на Рис. 7в
- в) функция, подсчитывающая количество упаковываемых операций в цикле
- г) алгоритм раскрутки циклов unroll
- д) алгоритм генерации векторных выражений, который использует рекурсивную процедуру, приведенную на Рис. 7е
- е) рекурсивная процедура поэлементной векторизации выражений

## 2. Использование специализированных векторных операций

В дополнение к алгоритму, описанному в предыдущей главе, предусмотрена возможность оптимизации некоторых сложных семантических конструкций оптимизируемой программы. Эту возможность предоставляют специальные операции, генерация которых позволяет уменьшить число операций в цикле, а порой – векторизовать циклы, которые без этого предварительного преобразования представляются не векторизуемыми.

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 и большинстве упомянутых во введении архитектур это 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<sup>2</sup>), SUBS - (вычитание двух векторных операндов с сатурацией). Назовем такие операции *специализированными*.

Рассматриваемая в этой главе оптимизация реализуется следующим образом. Перед тем как выполнить ее основную часть, в программе ищутся фрагменты, которые можно представить одной специализированной операцией. Таким образом, весь процесс состоит из двух этапов:

- 1) в промежуточном представлении программы распознается семантическая конструкция, соответствующая одной из специализированных операций, - она заменяется на эту операцию;
- 2) применяется метод, описанный в главе 1, - используется векторное свойство специализированной операции.

<sup>2</sup> Этот термин обозначает, что значение результата вышло за границы определенного диапазона, в таком случае результату присваивается значение границы этого диапазона.

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 оптимизации нужно, чтобы диапазон значений операндов был представлен количеством битов в элементе вектора, генерируемого операцией. Для установления этого факта используется информация, предоставляемая анализом диапазонов операндов [5, 10, 9].

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, то есть конструкции, подобной той, что приведена на рис. 8а, принималось во внимание, что все конструкции, семантически описывающие сведение *a* к диапазону  $[-N, N-1]$ , имеют похожую структуру. Все варианты сводятся к различному порядку операций, присваивающих границы диапазонов, и к различным комбинациям проверок на соответствие значениям границ, например,  $N \geq a$  или  $N+1 > a$ . Аналогично будет распознана знаковая сатурация в другой записи, приведенной на Рис. 8б.

Такая конструкция семантически полностью эквивалентна сатурации в записи на Рис. 8а, и этот факт легко устанавливается при анализе промежуточного представления программы, на котором мы работаем.

Заметим, что кроме традиционного способа записи беззнаковой сатурации, приведенного на Рис. 9а, есть и другие. Например, для сведения *a* к диапазону  $[0, 255]$ , можно произвести действия, приведенные на

Рис. 9б. Такая или подобная конструкция распознается специальным образом, при этом также рассматриваются различные границы и диапазоны.

После того, как распознана основная структура сатурации, не составляет труда узнать ее специфику (знаковая она или беззнаковая, к какому диапазону значений сводится результат), подобрать соответствующую векторную операцию и сгенерировать ее. Все эти действия делаются автоматически.

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

Описанное в этой главе преобразование фактически преобразует определенные фрагменты программы, содержащие условные разветвления, в линейные участки, после чего задача векторизации сводится к задаче, решенной в главе 1. Понятно, что такая оптимизация возможна только тогда, когда семантика фрагмента программы совпадает с семантикой какой-либо имеющейся в архитектурной платформе векторной операции. В тех же случаях, когда такой операции нет или преобразование невозможно по какой-то другой причине, при векторизации необходимо обработать операции сравнения отдельно. Именно этому посвящена следующая глава.

<pre>S = ( a &gt;= N-1 ) ? (N-1) : ((a &lt; N) ? (-N) : (a))</pre> <p>(а)</p>	<pre>if ( a &gt;= N-1 ) S= N-1; else if ( S &lt; -N)     S =-N; else S = a;</pre> <p>(б)</p>
---	--

Рис. 8. Семантически эквивалентные примеры распознавания знаковой сатурации

<pre>if ( a &gt;= 255 ) S= 255; else S = a;</pre> <p>(а)</p>	<pre>S = ( a &amp;&amp; 0xFF); if (S != a)     if ( a &lt; 0 ) {S = 0;}     else {S = 0xFF;}</pre> <p>(б)</p>
--	---

Рис. 9. Беззнаковая сатурация, записанная в традиционной (а) и альтернативной (б) форме

### 3. Использование операций векторного сравнения для циклов со сложным управлением

#### 3.1. Операция векторного сравнения

Помимо прочих векторных операций, архитектура Эльбрус предусматривает операции мелкоформатного сравнения. Аналогично арифметическим векторным операциям они имеют два аргумента-вектора. Сравнение выполняется между элементами этих векторов. В качестве результата получается вектор,  $i$ -й элемент которого состоит из всех единиц, если результат сравнения  $i$ -х элементов аргументов - "истина", либо состоит из всех нулей, если результат сравнения - "ложь".

Случаи, когда приходится использовать этот тип операций в цикле, можно разделить на два основных:

- сравнения, при любом исходе которых управление остается в цикле;
- сравнения, когда при одном из возможных исходов управление выходит из цикла.

Эти случаи обрабатываются в рамках описываемого метода по-разному. Рассмотрим их более подробно.

#### 3.2. Векторизация сравнения в случае, когда управление остается внутри цикла

В этом случае преобразование должно фактически свести условное разветвление в предикатную форму [14], то есть переместить вычисления, находящиеся на обеих ветвях условного разветвления, на один линейный участок, но так, чтобы реально вычислялись только выражения из ветви, которая должна была выполняться в неоптимизированном коде. Основная проблема здесь состоит в сведении потоков данных с разных веток управления. Известно, что сведение потоков можно выполнить с использованием операции *merge*. Например, на Рис. 10а представлено сведение потоков в обычном виде, а на Рис. 10б - с использованием операции *merge*. В зависимости от значения предиката *cond* операнд *a* принимает значение *b1* или *b2*. Поскольку это семантически эквивалентное преобразование исходной программы, его можно взять за основу аналогичного потокового преобразования над мелкоформатными значениями с использованием соответствующих операций сравнения (Рис.10в).

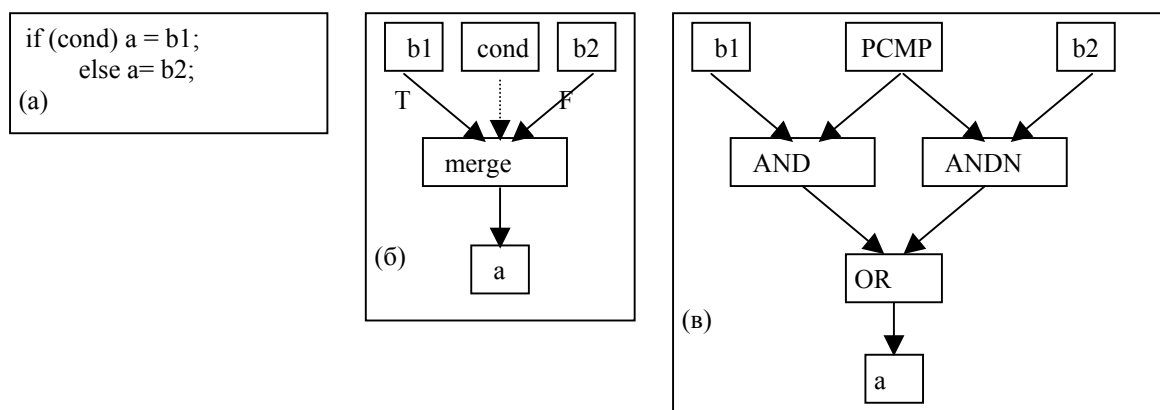


Рис. 10. Пример сведения потоков  $b1$  и  $b2$

- Формулировка условия
- Сведение с использованием операции *merge*
- Сведение с помощью операций *PCMP*, *AND* и *OR*

Векторная операция сравнения *PCMP* вырабатывает маску по результатам сравнения каждой пары элементов, взятых из векторов-аргументов. Операция *AND* не изменяет значение  $b1$  в случае, если условие выполняется (операция *PCMP* подала элемент, состоящий из единиц), и обнуляет  $b1$ , если условие не выполняется (операция *PCMP* подала элемент, состоящий из нулей). Операция

ANDN (AND с инвертированным первым аргументом), наоборот, пропускает значение b2 в случае невыполнения условия. Объединив результаты операций AND и ANDN с операцией OR, получаем нужный результат. Таким образом, эта схема моделирует действие операции merge за три операции.

Приведем следующий пример векторизации сравнения в теле цикла. Путь задан цикл, в котором в байтовый массив b[i] пересылается абсолютное значение элементов другого байтового массива a[i] (Рис. 11а). После преобразования unroll нужно, чтобы в каждый элемент b[i], в зависимости от знака a[i], записывалось либо значение a[i], либо значение -a[i]. Результатом векторной операции PCMPGT (a[i...i+7], 0) является вектор-маска, элемент которой k (k = i...i+7) состоит из единиц, если a[k] > 0, или состоит из нулей – в противном случае. Далее, эта маска накладывается на вектор a[i...i+7], а инвертированная маска – на minus\_a[i...i+7], то есть на результат векторного вычитания PSUB (0, a[i...i+7]). Выполнив побитовую операцию OR над результатами наложения масок, получаем искомый вектор, который записывается в b[i...i+7] (Рис. 11б).

Таким образом, для восьми итераций цикла потребовалось пять операций, или 0.625 операций на итерацию, вместо исходных трех операций на итерацию. Соответственно этой логике преобразуются и другие случаи вы-

```

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

for ( i = 0; i < N; i+=8)
{ mask = PCMPGT( a[i...i+7], 0);
  minus_a[i...i+7] = PSUB ( 0, a[i...i+7]);
  v1[i...i+7] = AND( mask, a[i...i+7]);
  v2[i...i+7] = ANDN( i_mask, minus_a[i...i+7]);
  b[i...i+7] = OR( v1[i...i+7], v2[i...i+7]); }
б)
    
```

Рис. 11. Пример векторизации сравнения в теле цикла  
 а) Исходный цикл – в байтовый массив b[i] пересылается абсолютное значение элементов другого байтового массива a[i]  
 б) Преобразованный цикл

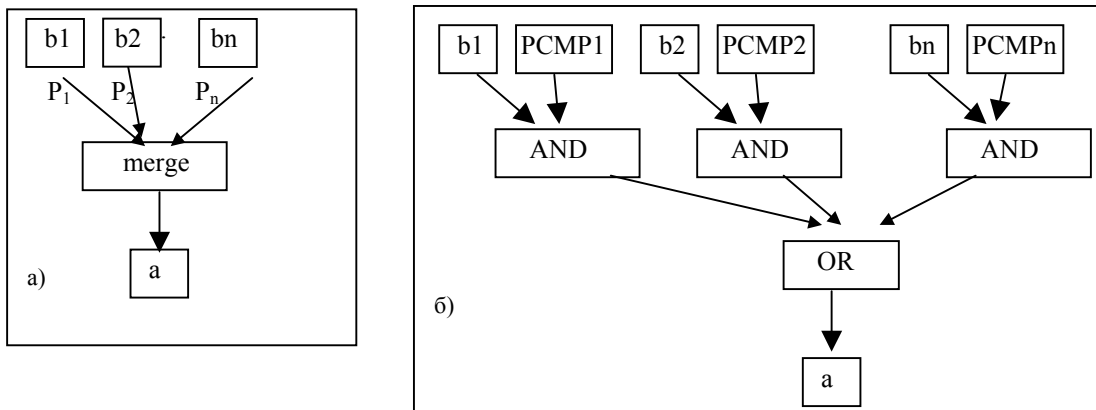


Рис. 12. Точная система предикатов  
 а) Предикатное представление  
 б) Представление с использованием векторных операций сравнения и логических операций

полнения сравнения внутри цикла. Вообще, случаи, когда можно применить сведение потоков с использованием операций AND и OR, можно обобщить следующим образом.

**Определение 1.** Пусть есть n сходящихся потоков данных, каждый из которых приходит под предикатом  $P_i$ ,  $0 < i < n+1$  (Рис. 12а). Тогда, если для всех предикатов выполняются два соотношения:

$$P_i \& P_j = 0 \quad (0 < i, j < n+1, i \neq j)$$

$P_1 | P_2 | \dots | P_n = 1$ , то условия, отвечающие данной системе предикатов, называются *полными условиями*, а сам набор предикатов называется *точной системой предикатов*.

Полные условия означают, что один и только один предикат может принимать значение "истина".

**Утверждение 1.** Для того чтобы отобразить сведение потока данных при помощи операций AND и OR, достаточно, чтобы система предикатов, под которой приходят данные, была точной.

Отсюда следует, что, убедившись в полноте условий, можно принять решение о векторизации цикла, поскольку все упомянутые выше свойства операций AND и OR сохраняются и в векторном случае (Рис. 12б). Значит, условное разветвление сводится к рамкам линейного участка, и после этого можно применить алгоритм, описанный в главе 1.

### 3.3. Векторизация сравнений в случае, когда управление выходит из цикла

В библиотеках и в прочих приложениях часто встречаются циклы с боковыми выходами<sup>3</sup>. В качестве примера можно привести операции со строками: вычисление длины строки (strlen)(Рис 13а) и сравнение двух строк (strcmp) (Рис. 13б).

<pre>for(i = 0; ; i++) {     if (!src[i]) break; } a)</pre>	<pre>for(i = 0; ; i++) {     if (src1[i] != src2[i]) break;     if (!src[i]) break; } б)</pre>
---	--

Рис 13. Примеры циклов с боковыми выходами

- а) Функция strlen
- б) Функция strcmp

Векторизация таких циклов, использующая особенности операций векторного сравнения, может дать хороший рост производительности. Ее можно провести следующим образом.

После того, как выполнена раскрутка цикла на F итераций (Рис.14а), у него образуется F боковых выходов (Рис.14б). Для того чтобы векторизовать такой цикл, все боковые выходы нужно свести к одному; также следует свести к одному все постциклы<sup>4</sup>, которые соответствуют исходным

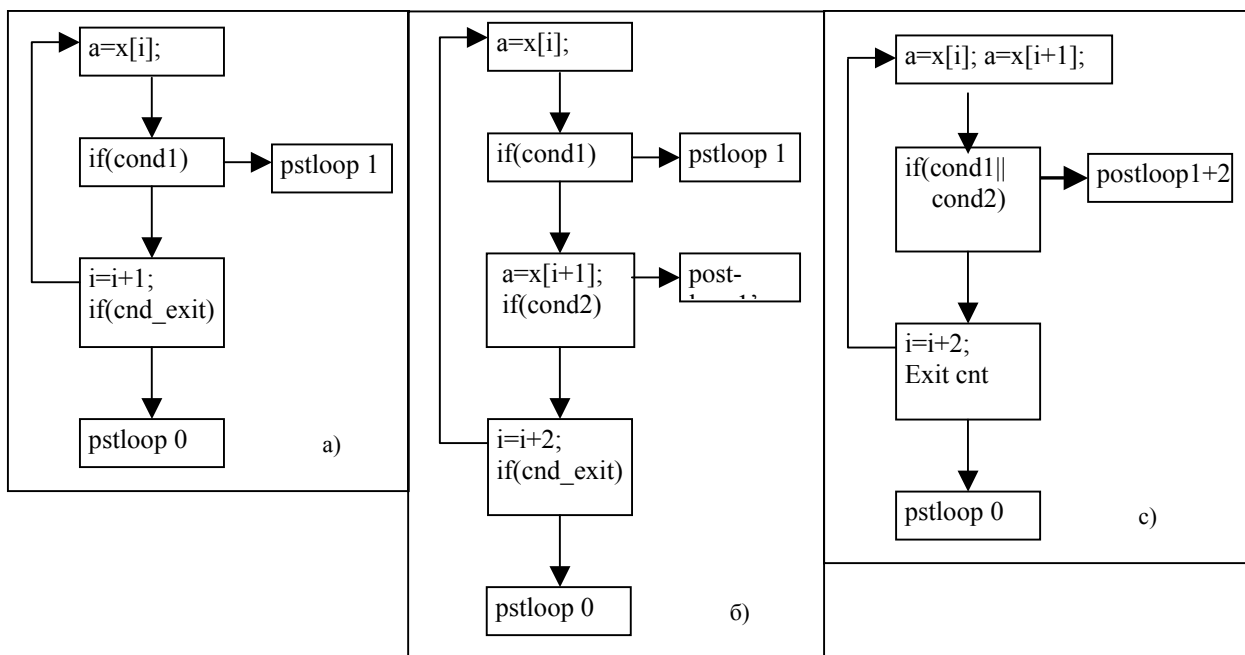


Рис. 14. Векторизация цикла с боковым выходом

- а) Исходный цикл
- б) Цикл после преобразованная unroll
- в) Цикл после объединения выходов

<sup>3</sup> Под боковым выходом из цикла подразумевается выход, осуществляющийся не по счетчику цикла

боковым выходам (Рис.14в). 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 в ситуации, когда условие сравнения не выполняется), и выйти из цикла.

Поскольку в преобразованном цикле управление может выйти из цикла по боковому выходу один раз на  $F$  итераций (где  $F$ -число объединяемых выходов из цикла), то реальное число исполненных итераций исходного цикла кратно  $F$ . Зная значения предикатов, по которым происходили боковые выходы в исходном цикле, можно вычислить систему точных предикатов, которая позволит восстановить вне цикла значения переменных, вычислявшихся в цикле, включая управляющую переменную цикла.

**Утверждение 2.** При объединении условий выхода значение каждой используемой за выходом переменной может быть получено путем объединения ее значений в теле цикла под точными предикатами, удовлетворяющими правилу взаимоисключающих условий.

Так, в примере на Рис.14в, в боковом постцикле преобразованного цикла нужно сформировать систему точных предикатов, основываясь на значениях `cond1` и `cond2` (Рис.15).

Главное ограничение на описанное в данной главе преобразование – отсутствие пробочных эффектов в цикле. Так, например, если в цикле с боковым выходом есть операция записи в элемент массива, то боковые выходы нельзя объединить в один, ибо в этом случае могут выполняться не предусмотренные исходной семантикой цикла записи, и будут происходить изменения контекста, которые не должны были случиться в исходном цикле.

Данное ограничение можно устранить, добавив для каждого записываемого элемента массива его предварительное считывание и запись в результирующий вектор под полным условием выхода из цикла. Но этот дополнительный код снижает эффективность от векторизации, поэтому он не рассматривается в описываемом методе.

### 3.4. Пример векторизации цикла

В мультимедийных приложениях циклы с боковым выходом встречаются очень редко, поэтому для демонстрации рассмотренного метода была выбрана задача из пакета SPECint92 [13]. Прекрасным примером векторизации цикла служит оптимизация основного цикла из задачи 023.eqntott, в котором пришлось векторизовать как сведение потоков данных, описанное в разделе 3.2, так и боковой выход из цикла. Этот цикл выполняется много раз, так что на него приходится более половины динамически исполняемых инструкций при запуске на исполнение задачи eqntott.

Состояние исходного цикла после ряда преобразований приведено на Рис.16а. При этом форматы массивов `a[i]` и `b[i]` занимают 16 битов, что и будет использовано при векторизации цикла. Функции этого фрагмента кода сводятся к трем операциям. Первая исполняет сравнение переменной `aa` с числом 2, вторая присваивает этой переменной значение 0, если заданное условие выполнилось, третья выполняет выход из цикла, если значения переменных `aa` и `bb` не равны.

Векторизация данного цикла выполняется следующим образом.

- Раскручивание тела цикла на 4 итерации (преобразование unroll).
- Считывание значений `aa` и `bb` – 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. Благодаря этому, 4 операции LOAD двух байт заменяются одной операцией LOAD восьми байт.

```

if (cond1)
{ a=x[i];}
if ( !cond &&
    cond2)
{i=i+1}

```

Рис. 15. Восстановление значений переменной цикла  $i$  и скалярной переменной  $a$ , изменяющейся в цикле, в постцикле `postloop 1+2`

<sup>4</sup> *Постцикл* – узел управляющего графа. Управление от цикла всегда переходит непосредственно к одному из постциклов данного цикла.

<pre> for ( i = 0; i &lt; ninputs; i++ ) { aa = a[i]; bb = b[i]; if (aa == 2) aa = 0; if (aa != bb) { goto exit2; } } ... exit2 if (aa &lt; bb) { return (-1);} else { return (1);} (a)                 </pre>	<pre> for ( i = 0; (i &lt; ninputs); i+=4 ) { aa[0...3] = a[0]-&gt;ptand[i...i+3]; bb[0...3] = bb_[i...i+3]; cc = PCMPEQ(aa[0...3], 0x2000200020002); aa[0...3] &amp;= ~cc ; if (aa[0...3] != bb[0...3] ) { goto exit2; } } ... exit2: if (aa[0] != bb[0]){aa= aa[0];bb=b[0];} else if (aa[1] != bb[1]){aa= aa[1];bb=b[1];} else if (aa[2] != bb[2]){aa= aa[2];bb=b[2];} else {aa= aa[3];bb=b[3];} if (aa &lt; bb) { return (-1);} else { return (1);} (b)                 </pre>
--	---

Рис. 16. Пример оптимизации, выполненной для задачи 023.eqntott пакета SPECint92

- а) Исходный цикл
- б) Полностью преобразованный цикл

- Сравнение aa с двойкой и присваивание aa значения 0 в случае выполнения сравнения - частный случай сведения потоков данных, описанное в разделе 3.2. Генерируемые в общем случае операции приведены на Рис. 10. К изложенному можно добавить элементарное потоковое преобразование: так как в качестве b1 в данном случае выступает 0, результат AND всегда 0, а значит, и результат OR всегда совпадает с результатом операции ANDN. Следовательно, остается только пара векторных операций. Первая – векторное сравнение вектора из четырех членов массива, каждый элемент вектора сравнивается с двойкой. В шестнадцатеричном виде такой второй аргумент выглядит как 0x2000200020002 (Рис. 17а). 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. 18).

- Сравнение aa и bb и выход в случае неравенства преобразовывается в простое сравнение векторов из четырех aa и bb (такое упрощение можно применить всегда, когда выход из цикла происходит по условиям "равно" или "не равно"). Единственный побочный эффект такого преоб-

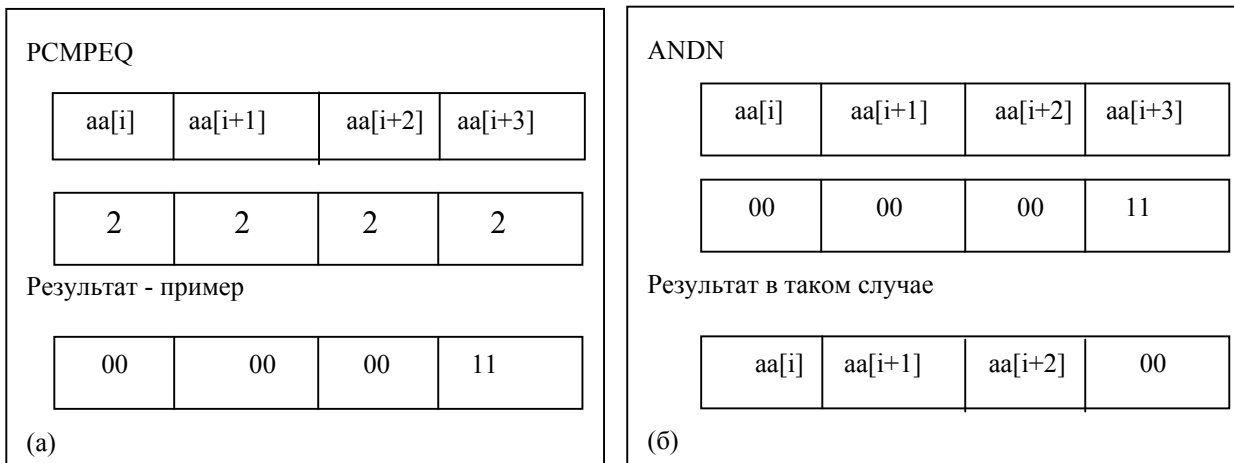


Рис. 17. а) Пример работы векторной операции PCMPEQ (под 0 в «результате» обозначен байт, заполненный битами-нулями, а под 1 – байт, заполненный битами-единицами)  
 б) Пример работы операции ANDN

разования – то, что в боковом постцикле не известен вектор масок от сравнений  $aa[i]$   $bb[i]$  и в постцикле добавляется сравнение элементов этих векторов для формирования системы точных предикатов, с помощью которых восстанавливается значение переменных  $aa$  и  $bb$  за циклом.

Таким образом, после векторизации (Рис. 16б) в теле цикла остается 5 операций на 4 итерации исходного цикла, или 1.25 операций на одну итерацию.

## 4. Экспериментальные результаты

### 4.1. Уменьшение количества исполняемых инструкций

Эффективность любой оптимизации определяется тем, насколько уменьшилось количество инструкций, исполняемых в динамике в результате ее применения. Здесь будут приведены результаты исполнения группы тестов, каждый из которых моделирует поведение какой-либо реальной 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 насыщением). В Табл.1 приведены коэффициенты уменьшения количества исполняемых инструкций в результате генерации векторных операций. Цифры приведены для различных групп тестов, в которых обрабатывались элементы вектора-аргумента размером 8, 16 и 32 бита.

Табл. 1. Коэффициент уменьшения количества 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	-

### 4.2. Количество применений оптимизации

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

В Табл. 2 показаны случаи, в которых при оптимизирующей компиляции пакетов SPECint95 и SPECint92 применялись векторизация цикла и замена сложной семантической конструкции на специализированную операцию.

Табл. 2. Показатели применения векторизации цикла и замены сложной семантической конструкции на специализированную операцию

Тест из пакета SPECint95	Количество применений
099.go	96
124.mksim	32
126.gcc	2
129.compress	14
132.jpeg	4
134.perl	3

Тест из пакета SPECint92	Количество применений
023.eqntott	2
026.compress	2
072.sc	8
085.gcc	12

## Заключение

В статье описан метод оптимизации циклов с использованием векторных операций. Он базируется на алгоритме, позволяющем векторизовать произвольные выражения в цикле. Суть алгоритма состоит в запоминании выделенной вершины графа каждого из выражений при раскрутке цикла, а затем при проходе по единственной копии исходного выражения - генерации векторных операций, соответствующих каждой из обычных операций. В рамках общего алгоритма разработаны методы распознавания сложных семантических конструкций (таких, как нахождение среднего арифметического, максимума, минимума, знаковой и беззнаковой сатурации), описаны способы их автоматической оптимизации с использованием векторных операций, базирующиеся на выделении основных семантических компонентов каждой из конструкций. Другой целью метода является преобразование, основанное на свойствах операций векторного сравнения, позволяющее 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 – коэффициент уменьшения количества динамически исполняемых инструкций для различных групп мультимедийных приложений, а также количество срабатываний описанного оптимизирующего преобразования при оптимизации задач из пакетов тестов SPECint95 и SPECint95.

В дальнейшем предполагается развитие описанного метода по следующим направлениям:

- полная реализация общего алгоритма векторизации циклов с выходами (снятие имеющихся в данный момент ограничений);
- векторизация циклов, раскрученных вручную программистом, использующая оптимизацию `roll`;
- расширение алгоритма за счет средств упаковки и распаковки аргументов и результатов любой операции, которая не может быть векторизована;
- расширение алгоритма на упакованные операции плавающей арифметики.

## Литература

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. - pp. 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, pp. 1-7.
7. A. Pelag, U. Weiser. MMX Technology Extension to Intel Architecture. // IEEE Micro, 16(4), pp 42-50, Aug 1996.
8. R. Lee. Subword Parallelism with MAX-2. // IEEE Micro, 16(4), pp. 51-59, Aug 1996.
9. M. Stephenson, J.Babb, S.Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. // In Proceeding of SIGLAN '00 Conference on 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.
11. W. Triebel. Itanium Architecture for Software Developers. Intel Press, 2000.
12. D.F. Bacon, S.L. Graham, O.J. Sharp. Compiler Transformations for High-Performance Computing. // ACM Computing Surveys, Vol. 26, No. 4, Dec 1994, pp. 361-368.
13. <http://www.spec.org>
14. Волконский В.Ю., Окунев С.К. Предикатное представление как основа оптимизации программы для архитектур с явно выраженной параллельностью // Информационные технологии, №4. Апрель, 2003.

**Ровинский Евгений Владимирович.** Родился в 1978 году. Окончил Московский физико-технический институт в 2001 году. Автор 4 научных работ. Область научных интересов - теория оптимизирующей компиляции. Аспирант Института микропроцессорных вычислительных систем, научный сотрудник ЗАО "МЦСТ".