МЕТОДЫ ПОВЫШЕНИЯ ЭФФЕКТИВНОСТИ АВТОМАТИЧЕСКОЙ ВЕКТОРИЗАЦИИ ВЫЧИСЛЕНИЙ

А.В. Ермолицкий, М.И. Нейман-заде
Институт Электронных Управляющих Машин
Россия, 119334, Москва, ул. Вавилова, 24
E-mail: ermolitsky@nm.ru, muradnz@mcst.ru

Ключевые слова: векторизация, оптимизирующий компилятор, векторные инструкции.
Рассматриваются методы повышения эффективности автоматической векторизации. Предложен ряд преобразований, позволяющих увеличить количество выровненных инструкций обращения к памяти и динамически определить эффективность векторизации. Разработанные методы реализованы в составе оптимизирующего компилятора. Их эффективность проверена в ходе экспериментальных исследований.

INCREASING AUTOMATIC VECTORIZATION EFFICIENCY / A.V. Ermolitsky, M.I. Neiman-zade (Institute of Electronic Controlling Machines, Vavilov str., 24, Moscow, 119334, Russia).
Key words: vectorization, optimizing compiler, vector instructions.
Methods for increasing automatic vectorization efficiency are considered. New transformations for increasing number of aligned memory references and dynamic vectorization efficiency checking are proposed. Developed methods are implemented in optimizing compiler. Experimental validation of their effectiveness is presented.

1 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 различных оптимизирующих компиляторов разработан ряд методов автоматической генерации векторных инструкций [1, 2, 3, 4, 5]. Тем не менее, в некоторых случаях они оказываются недостаточно эффективными, и автоматически сгенерированный код уступает написанному вручную. В данной работе рассмотрено несколько разработанных авторами преобразований, позволяющих повысить производительность векторного кода.

2 Схема векторизации

Рассмотрим схему векторизации цикла на основе раскрутки (Loop Unroll [6]), впервые предложенную в работе [2] и затем усиленную в [4, 7]. В рамках этой схемы векторизация цикла состоит из следующих фаз, выполняемых для всех наиболее вложенных циклов процедуры:

- Вспомогательные преобразования
- Анализ цикла
- Раскрутка цикла
- Генерация векторных инструкций

Данная работа посвящена вспомогательным преобразованиям, выполняемым перед векторизацией и позволяющим повысить ее эффективность. Эти преобразования будут подробно рассмотрены в следующих разделах. Рассмотрим вкратце остальные фазы векторизации. Для этого введем следующие определения.

Определение 1 *Выражением* будем называть слабо связную компоненту графа определений-использований цикла за исключением инструкций, вычисляющих адреса обращений к памяти.

Определение 2 Два графа назовем *изоморфными*, если существует взаимно однозначное соответствие между их вершинами и ребрами, сохраняющее смежность и инцидентность. Соответствующие инструкций изоморфных графов определений-использований будем называть *изоморфными*.

На *фазе анализа цикла* выполняется статический анализ возможности и целесообразности векторизации. Вначале производится поиск содержащихся в цикле выражений. Для этого выполняется обход графа определений-использований, начиная от инструкций обращения к памяти. Далее, для каждого выражения выполняется анализ зависимостей, диапазонов значений и другие анализы, позволяющие определить семантическую корректность векторизации. Если векторизация выражения была признана возможной, выполняется эвристический анализ, позволяющий оценить эффективность векторизации. Если и этот анализ дал положительный результат, то выражение помечается как пригодное для векторизации (векторизуемое). Далее векторизация производится только для циклов, содержащих векторизуемые выражения.

На *фазе раскрутки цикла* выполняется дублирование тела цикла для создания изоморфных копий исходных скалярных выражений. Фактор раскрутки F_{unroll} (количество копий тела цикла) определяется форматом обращений к памяти:

$$F_{unroll} = kL_v/S_{mem}$$

где L_v - размер векторных регистров (в байтах), S_{mem} - минимальный формат инструкций обращения к памяти в цикле, k - произвольное целое число больше 0. В дальнейшем для простоты будем считать, что $L_v = 8$, а $k = 1$.

float a[N], b[N], c[N];
...
for(i = 0; i < N; i++)
  a[i] += b[i] * c[i];
 	
а)
	for(i = 0; i < N; i+=2)
	{
	  a[i] += b[i] * c[i];
	  a[i+1] += b[i+1] * c[i+1];
	}
	if((N % 2) != 0)
	  a[i] += b[i] * c[i];
	
	б)
		for(i = 0; i < N; i+=2)
		{
		  V0 = PFMULS(b[i:i+1], c[i:i+1]);
		  a[i:i+1] = PFADDS(a[i:i+1], V0);
		}
		if((N % 2) != 0)
		  a[i] += b[i] * c[i];
		
		в)

Рис. 1: Пример векторизации цикла

На *фазе генерации векторных инструкций* выполняется замена групп изоморфных скалярных инструкций, созданных в результате раскрутки, соответствующими векторными инструкциями.

В качестве примера рассмотрим цикл на рис. 1а, содержащий одно скалярное выражение. В результате раскрутки ($F_{unroll} = 2$) 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, считая что количество итераций цикла кратно фактору раскрутки. Далее, две 32-разрядные скалярные инструкции умножения заменяются 64-разрядной векторной инструкцией умножения PFMULS, скалярные инструкции сложения - векторной PFADDS, а 4-байтовые инструкции чтения/записи - соответствующими 8-байтовыми векторными обращениями к памяти (рис. 1в). В результате векторизации цикл значительно ускоряется за счет параллельного исполнения итераций исходного цикла.

3 Выравнивающие преобразования

3.1 Векторизация невыровненных обращений к памяти

Введем следующее определение

Определение 3 Инструкция обращения к памяти является *выровненной*, если адрес области памяти, к которой производится обращение, является кратным формату этой инструкции.

В результате векторизации формат обращений к памяти увеличивается, из-за чего они могут стать невыровненными. В зависимости от архитектуры, обращение в память по невыровненному адресу приводит либо к возникновению исключительной ситуации, либо к значительному увеличению времени исполнения инструкции. В приведенном выше примере (рис. 1) неявно предполагалось, что все массивы выровнены на 8 байт, так что как в исходном, так и в векторизованном цикле все обращения к памяти являются выровненными. Если инструкция оказывается невыровненной на требуемый для векторизации размер, или статический анализ не может определить ее выровненность, оптимизирующий компилятор должен использовать специальную технику векторизации таких инструкций

[8, 9], что влечет за собой снижение эффективности генерируемого кода. Ситуация, когда невозможно статически определить выровненность обращения к памяти, очень часто встречается в различных приложениях.

а)	б)

`void memcpy( char* dst, char* src, int n) {`
`  for( i = 0; i < n; i++ )`
`    dst[i] = src[i];`
`}`
	`ma = src & 0x7;`
	`asrc = src & 0xFFFFFFF8;`
	`for( i = 0; i < n; i+=8 ) {`
	`  V1[0:7] = asrc[i:i+7];`
	`  V2[0:7] = asrc[i+8:i+8+7];`
	`  V3[0:7] = INSF( V1[0:7], V2[0:7], ma);`
	`  dst[i] = V3[0];`
	`  dst[i+1] = V3[1];`
	`  ...`
	`  dst[i+7] = V3[7];`
	`}`

Рис. 2: Пример векторизации невыровненных инструкций

В качестве примера рассмотрим простейшую реализацию функции копирования байтового массива (рис. 2а). Будем считать, что значения указателей **src** и **dst** неизвестны, так что статический анализ не может определить выровненность обращений к памяти. При векторизации чтения по указателю **src** строятся две 8-байтовые инструкции чтения по выровненному адресу (**asrc**), покрывающие читаемую область памяти, и их результаты объединяются с помощью операции INSF (рис. 2б). Здесь V1, V2 и V3 - некоторые векторные регистры, а операция INSF в зависимости от архитектуры может быть реализована различными инструкциями. При этом в векторизованном цикле остались скалярные инструкции записи в память. Также в цикле неявно присутствуют операции выделения элементов векторного регистра V3. Таким образом, векторизованный цикл содержит 19 операций, в то время как в случае выровненных обращений к памяти потребовалось бы только 2 операции (чтение и запись в память).

Для решения данной проблемы существуют два основных вспомогательных преобразования: открутка итераций (Peeling) и динамическая проверка выровненности (Align Versioning). Они называются также *выравнивающими преобразованиями*.

3.2 Открутка итераций цикла

Открутка итераций цикла для выравнивания инструкций обращения к памяти была впервые предложена в работе [3]. Данное преобразование заключается в вынесении нескольких первых итераций цикла в отдельный цикл или скалярный код перед исходным циклом. Количество вынесенных итераций подбирается такое, чтобы на первой итерации цикла адрес выравниваемой инструкции был выровнен на L_v байт. Это позволяет выровнять один указатель в цикле.

Рассмотрим применение открутки итераций на примере цикла, содержащего два указателя с неизвестным выравниванием (рис. 3а). Выравнивание записи является более приоритетным, поскольку векторизация невыровненной инструкции записи в память требует больше ресурсов по сравнению с инструкцией чтения. В результате открутки **np** первых итераций (верхний цикл на рис. 3б) указатель **px** становится выровненным на

<pre>char *px, *py;
...
/* px и px не выровнены */
for(i = 0; i < N; i++)
    px[i] = py[i];</pre> а)
	<pre>np = (8 - (px & 0x7)) & 0x7;
	if(np > N)
	    np = N;
	for(i = 0; i < np; i++)
	    px[i] = py[i];
	/* px выровнен, py - нет */
	for(; i < N; i++)
	    px[i] = py[i];</pre> б)

Рис. 3: Пример выравнивания инструкции при помощи открутки итераций цикла

8 байт в нижнем цикле. Выровненность **py** остается при этом неизвестной и далее может быть проверена динамически при помощи метода, рассматриваемого в следующем разделе.

3.3 Динамическая проверка выровненности

В случае, когда необходимая для эффективной векторизации информация о выровненности инструкций цикла не может быть получена статически, можно получить ее динамически - путем построения проверок выровненности и создания версий цикла. Впервые данный метод был предложен в работе [10], а затем усилен в [2, 3].

<pre>char a[N], *p;
...
/* p не выровнен */
for(i = 0; i < N; i++)
    p[i] = a[i];</pre> а)
	<pre>if((p & 0x7) == 0) {
	    /* p выровнен */
	    for(i = 0; i < N; i++)
	        p[i] = a[i];
	} else {
	    /* p не выровнен */
	    for(i = 0; i < N; i++)
	        p[i] = a[i];
	}</pre> б)

Рис. 4: Пример применения динамической проверки выровненности

В качестве примера рассмотрим цикл на рис. 4а. Будем считать, что статический анализ не может определить выровненность указателя **p**. Динамическая проверка позволяет проверить его выровненность, так что в верхнем цикле все инструкции обращения к памяти оказываются выровненными (рис. 4б). При этом оба цикла могут быть векторизованы, однако верхний цикл векторизуется более оптимально. Таким образом, если при исполнении программы указатель **p** окажется выровненным, то управление будет передано на более эффективный верхний цикл, в противном случае будет исполнен менее эффективный нижний цикл.

В общем случае, если цикл содержит $N > 1$ указателей, выровненность которых не может быть определена статически, возникает проблема: как выбрать способ проверки выровненности этих указателей. Например, компилятор может создать две копии цикла

и N проверок выровненности всех указателей, так что в одной копии цикла все указатели выровнены, а в другой все указатели считаются невыровненными (хотя некоторые из них могут оказаться выровненными). Если выровненность всех указателей независима, то можно создать 2^N копий цикла и 2^N-1 проверок выровненности, так что в каждой копии цикла будет точно известна выровненность всех указателей. Последний метод приводит к очень быстрому росту размера кода, из-за чего на практике он может быть использован только для небольших значений N.

В некоторых случаях различные указатели в цикле могут обладать одинаковой выровненностью, что позволяет уменьшить количество динамических проверок. Данный вопрос подробно рассматривается в следующем разделе.

Стоит отметить, что рассмотренный метод не позволяет гарантированно выровнять инструкции обращения в память, он позволяет лишь проверить их выровненность. Это означает, что данный метод не повышает эффективность векторизации в случае, когда все указатели в цикле являются невыровненными.

3.4 Алгоритм выравнивания обращений к памяти

Описанные выше методы выравнивания обращений к памяти работают на уровне отдельных указателей. Из-за этого они оказываются недостаточно эффективными в случае наличия в цикле множества взаимосвязанных обращений к памяти. Авторами данной работы предложен алгоритм работы выравнивающих преобразований, устраняющий данный недостаток. Кроме того, в рамках этого алгоритма предложен эффективный способ минимизации количества динамических проверок выровненности за счет объединения в группы указателей с одинаковой выровненностью.

Формально задача выравнивания инструкций в цикле может быть сформулирована следующим образом. Имеется цикл, содержащий N инструкций обращения к памяти, часть из которых (возможно, все) могут быть невыровненными. Каждая инструкция имеет некоторый вес w_i. Требуется выполнить выравнивающие преобразования, минимизирующие сумму весов невыровненных инструкций:

$$\sum_{i=1}^{N} x_i w_i \to min$$

где x_i равен 0, если инструкция выровнена, и 1 в противном случае. При этом разумно наложить ограничение на количество создаваемых копий цикла и динамических проверок выровненности. Конкретные значения w_i зависят от типа инструкции и для каждой архитектуры подбираются эмпирически. Введем следующее

Определение 4 *Кортежем* будем называть множество инструкций чтения/записи одинакового формата, обращающихся к смежным ячейкам памяти.

Предлагаемый алгоритм выравнивания инструкций цикла состоит из следующих фаз:

- поиск кортежей инструкций обращения к памяти
- объединение кортежей с одинаковой выровненностью в *группы*

- ранжирование групп инструкций и выполнение выравнивающих преобразований

Рассмотрим фазы алгоритма более подробно. На *фазе поиска кортежей* выполняется анализ адресов инструкций обращения к памяти и объединение смежных инструкций в кортежи. Инструкции, не имеющие смежных инструкций, заносятся в *вырожденные* кортежи, содержащие только одну инструкцию. Результатом работы фазы является список кортежей инструкций чтения/записи. В дальнейшем будем считать, что инструкции кортежа упорядочены по адресу в порядке возрастания, так что первая инструкция имеет наименьший адрес.

Кортеж является наименьшей выравниваемой сущностью в цикле. Выравнивающие преобразования применяются к первой инструкции кортежа, при этом выровненность остальных его инструкций вычисляется по смещению относительно первой инструкции - это возможно, поскольку смещение всегда константное. В этом заключается основное отличие предлагаемого алгоритма от описанных в литературе, в которых производится выравнивание отдельных указателей, т.е. по сути выравнивается произвольная инструкция кортежа и при этом не предлагается какого-либо способа для определения выровненности остальных его инструкций.

Отметим, что не все кортежи могут быть выровнены. Выравниваемость зависит от шага S адреса инструкций кортежа за итерацию исходного цикла. Кортеж может быть выровнен только когда S является константой и выполняется условие:

$$SF_{unroll} \equiv 0 \bmod L_v \tag{1}$$

где L_v - длина векторных регистров в байтах, а F_{unroll} - фактор раскрутки цикла (напомним, что раскрутка цикла производится после выравнивающих преобразований). Невыполнение условия (1) приведет к тому, что уже на второй итерации векторизованного цикла инструкции кортежа станут невыровненными.

На следующей *фазе формирования групп* происходит объединение кортежей с одинаковой выровненностью в *группы*. Два кортежа имеют одинаковую величину выровненности, если разность адресов первых их инструкций кратна L_v (напомним, что инструкции кортежей упорядочены по возрастанию адресов). В данной работе предлагается эффективный метод определения эквивалентности выровненности двух кортежей, основанный на представлении адресов инструкций в виде *ps-форм* [11]. Будем называть *ps-формой* полином вида:

$$c_0 + c_1 x_{1,1} x_{1,2} ... x_{1,k_1} + ... + c_n x_{n,1} x_{n,2} ... x_{n,k_n}$$

где c_0, c_1, ..., c_n - некоторые константы, а $x_{i,j}$ - переменные.

Будем считать, что поток данных программы представлен в форме *статического единственного присваивания* (Static Single Assignment, SSA) [12, 13, 14, 15]. В этом случае для любой переменной существует единственная вырабатывающая ее инструкция. Для этого в представление вводятся *φ-функции*, размещаемые в точках схождения потоков управления. φ-функция является псевдооперацией, выбирающей нужное значение переменной среди множества значений, приходящих по разным потокам управления. Тогда справедливо следующее

Утверждение 1 Адрес любой инструкции обращения к памяти может быть представлен в виде ps-формы, где в качестве переменных используются результаты некоторых инструкций или φ-функций.

В качестве примера рассмотрим цикл, содержащий инструкцию записи (рис. 5a). Адрес инструкции записи в массив **A** может быть представлен в виде ps-формы $a0 + 2Mi + 2j$, где $a0$ - адрес начала массива **A**.

```
short A[N][M];
...
for( i = 0; i < N; i++ )
    for( j = 0; j < M; j++ )
        A[i][j] = ...;
                                    a)
```

Рис. 5: Пример вычисления ps-формы адреса инструкции

Покажем, каким образом ps-формы могут быть использованы для определения эквивалентности выровненности двух кортежей. Будем считать, что в компиляторе реализован анализ выровненности, позволяющий определить выровненность значения любого регистра промежуточного представления, равную максимальному числу $A = 2^n$, на которое делится без остатка значение регистра. Используя эту информацию можно легко определить выровненность адреса инструкции по его ps-форме. Так, выровненность адреса равна минимальной выровненности слагаемых соответствующей ps-формы. В свою очередь выровненность слагаемого ps-формы равна произведению выровненностей множителей этого слагаемого. Два кортежа имеют одинаковую выровненность L_v, если разность ps-форм адресов первых их инструкций выровнена на L_v (т.е. кратна L_v).

Описанный выше анализ ps-форм позволяет получить группы кортежей с одинаковой выровненностью. В ряде случаев это позволяет значительно уменьшить количество динамических проверок выровненности, поскольку выравнивающее преобразование, примененное к одному из кортежей группы, автоматически выравнивает все кортежи из этой группы. Таким образом, в рамках рассматриваемого алгоритма группа является элементарным объектом, к которому применяются выравнивающие преобразования.

На *фазе выравнивающих преобразований* выполняется ранжирование групп по их весу и выравнивание групп с наибольшим весом. Ранжирование необходимо в случае, когда в цикле есть несколько групп инструкций, поскольку в этом случае все группы выровнять невозможно. Вес группы является суммой весов инструкций, входящих в группу.

Невыровненная группа с наибольшим весом выравнивается с помощью открутки итераций, поскольку это единственное преобразование, позволяющее гарантированно выровнять группу инструкций (другие преобразования позволяют лишь проверить выровненность группы). Открутка итерации цикла для выравнивания одной группы делает невыровненными все остальные группы цикла. Открутка не имеет смысла, если группа с наибольшим весом является выровненной.

Выровненность остальных групп инструкций цикла, оставшихся невыровненными после открутки итераций, проверяется при помощи создания копий цикла и динамических проверок выровненности, описанных в разделе 3.3. Если ограничение на количество ко-

пий цикла не позволяет выровнять таким образом все группы, то выравниваются только группы с наибольшим весом.

```
float a, *b;
...
for( i = 0; i < n; i++ ) {
    b[i] = b[i] + a[i] * c0;
    b[i+1] = b[i+1] + a[i] * c1;
    b[i+2] = b[i+2] + a[i] * c2;
    b[i+3] = b[i+3] + a[i] * c3;
}
```

а)

```
float *a, *b;
...
for( i = 0; i < n; i++ ) {
    | b[i]   | = | b[i]   | + | a[i] | * c1
    | b[i+1] | = | b[i+1] | + | a[i] | * c2
    | b[i+2] | = | b[i+2] | + | a[i] | * c3
    | b[i+3] | = | b[i+3] | + | a[i] | * c4
}
```

б)

```
i = 0;
misalign_b = (unsigned)(&b[i]) & 0x7;
if( misalign_b != 0 ) {
    b[i] = b[i] + a[i] * c0;
    b[i+1] = b[i+1] + a[i] * c1;
    b[i+2] = b[i+2] + a[i] * c2;
    b[i+3] = b[i+3] + a[i] * c3;
    i++;
}
for( ; i < n; i++ ) {
    /* b выровнен */
    b[i] = b[i] + a[i] * c0;
    b[i+1] = b[i+1] + a[i] * c1;
    b[i+2] = b[i+2] + a[i] * c2;
    b[i+3] = b[i+3] + a[i] * c3;
}
```

в)

```
misalign_a = (unsigned)(&a[i]) & 0x7;
if( misalign_a == 0 ) {
    for( ; i < n; i++ ) {
        /* a и b выровнены */
        b[i] = b[i] + a[i] * c0;
        b[i+1] = b[i+1] + a[i] * c1;
        b[i+2] = b[i+2] + a[i] * c2;
        b[i+3] = b[i+3] + a[i] * c3;
    }
} else {
    for( ; i < n; i++ ) {
        /* b выровнен */
        b[i] = b[i] + a[i] * c0;
        b[i+1] = b[i+1] + a[i] * c1;
        b[i+2] = b[i+2] + a[i] * c2;
        b[i+3] = b[i+3] + a[i] * c3;
    }
}
```

г)

Рис. 6: Пример выравнивания инструкций цикла

В качестве примера работы алгоритма выравнивания рассмотрим цикл на рис. 6а. Исходный цикл содержит 8 инструкций чтения и 4 инструкции записи, при этом выровненность указателей **a** и **b** неизвестна. На первой фазе алгоритма выполняется поиск кортежей в цикле, в результате инструкции обращения к памяти объединяются в три кортежа: первый - из 4 записей в массив **b**, второй - из 4 чтений из массива **b**, третий - вырожденный кортеж единичной длины, включающий в себя 4 инструкции чтения из массива **a** (рис. 6б). На следующей фазе алгоритма найденные кортежи объединяются в 2 группы: кортежи инструкций обращения к **b** имеют одинаковую выровненность и поэтому объединяются в одну группу, оставшийся кортеж входит в другую группу. Наконец, группа из двух кортежей выравнивается с помощью открутки итерации цикла (рис. 6в), и далее выровненность группы из одного кортежа проверяется динамически (рис. 6г).

Стоит отметить, что в группу могут входить инструкции чтения/записи, обращающиеся к разным массивам. В качестве примера рассмотрим цикл сложения двух матриц размером 513x513 (рис. 7). ps-формы адресов инструкций обращения к памяти в данном цикле имеют вид: $addr(X[i][j]) = X0 + 4 * 513 * i + 4 * j$, где X обозначает один из мас-

```
float A[513][513], B[513][513], C[513][513];
...
for( i = 0; i < 513; i++ ) {
    for( j = 0; j < 513; j++ ) {
        A[i][j] = B[i][j] + C[i][j];
    }
}
```

Рис. 7: Пример цикла, все инструкции обращения к памяти которого входят в одну группу

сивов (**A**, **B** или **C**), а $X0$ - адрес начала соответствующего массива. Рассмотрим любые две инструкции обращения к памяти, например, запись в массив **A** и чтение из массива **B**. Разность адресов этих инструкций $addr(A[i][j]) - addr(B[i][j]) = A0 - B0$ кратна размеру вектора L_v, поскольку адреса начала локальных массивов выровнены на L_v. Таким образом, выровненность этих двух инструкций совпадает и они попадают в одну группу. Рассуждая аналогичным образом для других пар инструкций легко увидеть, что все инструкции цикла входят в одну группу, так что для выравнивания всех инструкций цикла достаточно одного выравнивающего преобразования (открутка итераций цикла). Подобная ситуация встречается в задаче 171.swim из пакета тестов SPEC FP2000, где один из горячих циклов содержит инструкции обращения к 9 двумерным массивам, при этом все инструкции обращения к памяти входят в одну группу.

3.5 Частичная открутка итераций цикла

Открутка итераций цикла, описанная в разделе 3.2, способна выровнять кортеж только если следующее уравнение имеет целочисленное решение x для всех m, кратных формату его инструкций:

$$m + Sx = 0 \bmod L_v \tag{2}$$

где m - величина невыровненности первой инструкции кортежа, S - шаг адреса инструкций кортежа, x - количество откручиваемых итераций цикла, L_v - длина векторных регистров (все параметры измеряются в числе элементов векторных регистров). Из теории чисел известно, что уравнение (2) имеет решение для произвольного m, если S и L_v являются взаимнопростыми. Поскольку для всех современных архитектур $L_v = 2^n$, то выравнивание путем открутки итераций цикла возможно только для кортежа инструкций с нечетным шагом S.

Заметим, что в описанных в литературе выравнивающих преобразованиях рассматриваются только отдельные инструкции (суть вырожденные кортежи), шаг адреса которых равен формату инструкции (т.е. $S = 1$), так что указанное выше ограничение для них удовлетворяется. Однако эти методы не работают для невырожденных кортежей.

В случае, когда для невыровненного кортежа уравнение (2) не имеет целочисленных решений, он может быть выровнен с помощью *частичной открутки итераций цикла.* Данное преобразования заключается в создании динамических проверок выровненности и копий цикла, и дальнейшем вынесении части нескольких первых итераций копий цикла в предцикл и части последней итерации - в постцикл. В случае, когда величина невыровненности кортежа известна статически, создание динамических проверок и

копий цикла не требуется, так что конечный код оказывается более эффективным и компактным. В результате преобразования выравниваемый кортеж инструкций оказывается выровненным во всех копиях цикла.

а)	б)

`int *dst, *src1, *src2;`
`...`
`for( i = 0; i < n; i++ ) {`
`    dst[2*i] = src1[i];`
`    dst[2*i+1] = src2[i];`
`}`
	`misalign = (unsigned)dst & 0x7;`
	`if ( misalign == 0 ) {`
	`    for( i = 0; i < n; i++ ) {`
	`        dst[2*i] = src1[i];`
	`        dst[2*i+1] = src2[i];`
	`    }`
	`} else {`
	`    dst[0] = src1[0];`
	`    for( i = 0; i < n - 1; i++ ) {`
	`        dst[2*i+1] = src2[i];`
	`        dst[2*i+2] = src1[i+1];`
	`    }`
	`    dst[2*i+1] = src2[i];`
	`}`

Рис. 8: Пример выравнивания кортежа при помощи частичной открутки итераций цикла

В качестве примера рассмотрим цикл преобразования двух массивов, содержащих действительную и мнимую части комплексных чисел, в один массив (рис. 8а). Будем считать, что выровненность всех указателей неизвестна и что группа инструкций записи имеет наибольший вес. Шаг адреса инструкций записи $S = 2$ элемента, так что обычная открутка итераций не позволяет выровнять их. В результате применения частичной открутки итераций создается проверка выровненности указателя **dst** и копия цикла, из которой вынесена часть первой и последней итерации (рис. 8б). При этом обе копии цикла содержат выровненные инструкции записи по указателю **dst**, что делает дальнейшую векторизацию этих инструкций максимально эффективной.

В общем случае количество создаваемых копий цикла зависит от формата и шага адреса выравниваемых инструкций кортежа. Максимальное количество копий цикла равно количеству элементов векторных регистров, используемых для векторизации данного кортежа. Необходимость создания множества копий обусловлено тем, что преобразование меняет сам цикл, и для различных значений величины невыровненности преобразование цикла различается. Таким образом, частичная открутка итерации цикла может приводить к значительному росту кода, что является ее основным недостатком.

4 Скрутка циклов

Большинство современных компиляторов способны раскручивать циклы. Однако, в программах нередко встречаются циклы, раскрученные программистом вручную. Подобная оптимизация, выполненная программистом, зачастую оказывается не самой эффективной для конкретной архитектуры и препятствует работе других оптимизаций. В частности, для таких циклов невозможно применение большинства вспомогательных преобразований. Одним из наиболее эффективных решений данной проблемы является выполнение *скрутки цикла* - преобразования, обратного раскрутке цикла. Оно реализуется перед

остальными вспомогательными преобразованиями и заключается в удалении созданных программистом копий тела цикла.

Формально задача скрутки цикла может быть сформулирована следующим образом. Имеется цикл, содержащий $n > 1$ выражений. Если все выражения являются копией одного выражения, полученного в результате раскрутки цикла (т.е. все выражения цикла изоморфны), необходимо удалить все копии, оставив только исходное выражение. Наибольшую сложность здесь представляет задача определения изоморфизма двух выражений, которая в общем случае является NP-полной. Быстрый алгоритм определения изоморфизма двух выражений был предложен в работе [1]. Этот алгоритм в большинстве практически важных случаев работает за время, пропорциональное количеству инструкций в анализируемых выражениях.

```
UChar yy[256];
...
j = nextSym-1;
for(; j > 3; j -= 4) {
    yy[j] = yy[j-1];
    yy[j-1] = yy[j-2];
    yy[j-2] = yy[j-3];
    yy[j-3] = yy[j-4];
}
```
а)

```
b = nextSym - 1;
n = (b - 3 + 4 - 1) / 4;
a = b - (n * 4);
for( j = b; j > a; j– )
    yy[j] = yy[j-1];
```
б)

Рис. 9: Пример скрутки раскрученного программистом цикла

В качестве примера рассмотрим один из горячих циклов задачи 256.bzip2 из пакета тестов SPEC CPU2000 (рис. 9а). Значение **nextSym** читается из памяти, поэтому статически определить выровненность инструкций обращения к памяти в цикле невозможно. Ручная раскрутка цикла препятствует выполнению вспомогательных преобразований, которые смогли бы выровнять инструкции обращения к памяти. В результате скрутки из цикла удаляются 3 копии выражения (рис. 9б). Далее инструкция записи в скрученном цикле может быть выровнена откручиванием итераций, после чего цикл может быть векторизован обычным образом. Стоит отметить, что в данном случае можно было бы не выполнять скрутку цикла, а сразу воспользоваться частичной откруткой итераций цикла для выравнивания инструкции записи. Однако при таком подходе размер конечного кода получается значительно больше, а сам код - менее эффективен.

5 Метод динамического разогревания циклов

Эффективность векторизации зависит от количества итераций цикла - если в цикле мало итераций, то накладные расходы могут нивелировать положительный эффект векторизации. Сейчас общепринятым является подход, когда при помощи профилирования или каких-либо эвристик оценивается среднее число итераций цикла, на основе которого принимается решение о целесообразности применения той или иной оптимизации. Однако такой подход оказывается неэффективным в случае, когда количество итераций цикла значительно меняется от запуска к запуску. Например, при компиляции библиотечных функций невозможно предсказать с какими параметрами будут вызваны эти функции,

и, как следствие, невозможно точно предсказать количество итераций большинства циклов в этих функциях. Таким образом, недостоверная профильная информация может приводить к отказу от векторизации либо к деградации производительности в случае применения векторизации.

В данной работе предлагается следующее решение указанной проблемы. В случае, когда компилятор не может точно определить количество итераций цикла, создается копия цикла и динамическая проверка, передающая управление на «горячую» копию цикла в случае, если число итераций цикла больше определенного порога, или на «холодную» копию цикла, если число итераций ниже этого порога. Данное преобразование называется *динамическим разогреванием циклов*.

```
f( char* dst, char* src, int len) {
    ...
    for( i = 0; i < len; i++ )
        dst[i] = src[i];
}
                                              а)
```

```
if( len > 32 ) {
        /* горячий цикл */
        for( i = 0; i < len; i++ )
            dst[i] = src[i];
} else {
        /* холодный цикл */
        for( i = 0; i < len; i++ )
            dst[i] = src[i];
}
                                              б)
```

Рис. 10: Пример динамического разогревания цикла

В качестве примера рассмотрим цикл на рис. 10а. Будем считать, что функция **f()** является библиотечной, так что никакой межпроцедурный анализ не сможет определить диапазон значений параметра **len**, задающего число итераций цикла. В результате динамического разогревания создается копия цикла (рис. 10б), так что верхний (горячий) цикл гарантированно имеет больше 32 итераций, в то время как нижний (холодный цикл) имеет не более 32 итераций. Далее к верхнему циклу могут применяться вспомогательные преобразования, позволяющие выровнять указатели **src** и **dst**, а также векторизация. Для нижнего цикла векторизация является нецелесообразной. Таким образом, в случае, когда статический анализ не может точно определить эффективность векторизации, она определяется динамически и в зависимости от числа итераций управление передается либо на векторизованную версию цикла, либо на скалярную версию. Отметим, что конкретный порог числа итераций цикла зависит от архитектуры и параметров цикла (числа инструкций в цикле, формата инструкций обращения к памяти и др.).

В общем случае число градаций «температуры» цикла может быть больше двух. Например, при трех градациях к самому горячему циклу может применяться полная векторизация, а к циклу со средней «температурой» - векторизация с меньшей длиной векторов. Такой подход нередко применяется при ручной оптимизации библиотечных функций.

6 Результаты

Рассмотренные в данной работе вспомогательные преобразования были реализованы в составе оптимизирующего компилятора для архитектуры «Эльбрус». В данном разделе приводятся результаты экспериментального исследования эффективности описанных

алгоритмов.

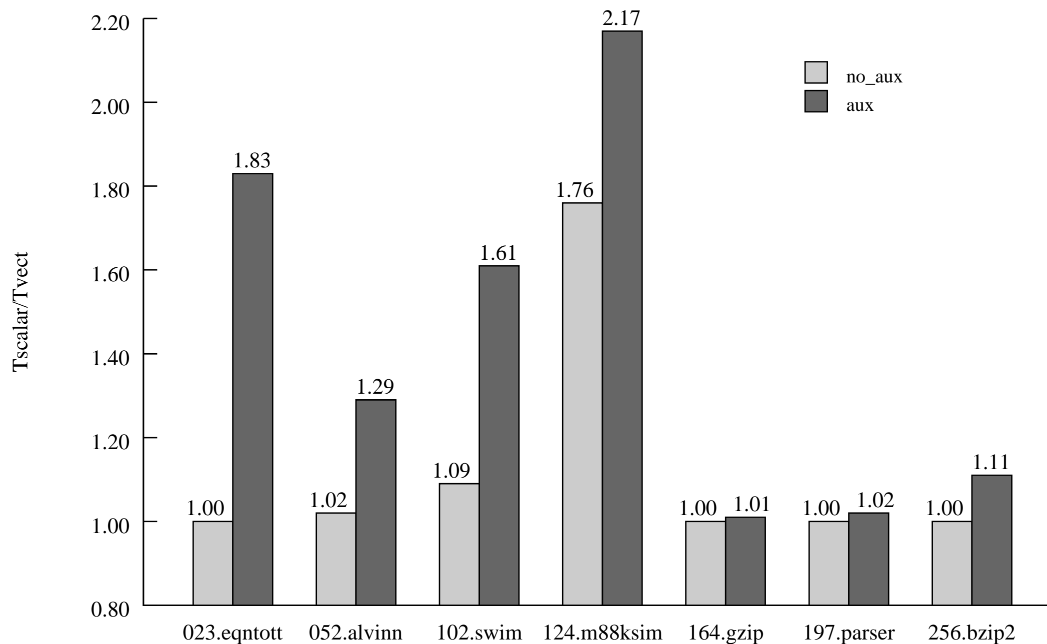

Рис. 11: Прирост производительности за счет векторизации на задачах SPEC

На рис. 11 представлено влияние предложенных вспомогательных преобразований на эффективность автоматической векторизации для пакетов тестов SPEC CINT92, SPEC CFP92, SPEC CINT95, SPEC CFP95 и SPEC CINT2000 [16]. На нем показано отношение $K = T_{scalar}/T_{vect}$ времени исполнения без векторизации (T_{scalar}) ко времени исполнения после применения векторизации (T_{vect}). Через no_aux обозначено значение K без вспомогательных преобразований, через aux - значение K в случае применения предложенных вспомогательных преобразований.

Вспомогательные преобразования позволили увеличить производительность рассмотренных задач в среднем в 1.25 раз. На отдельных задачах величина прироста производительности составила до 1.83 раз.

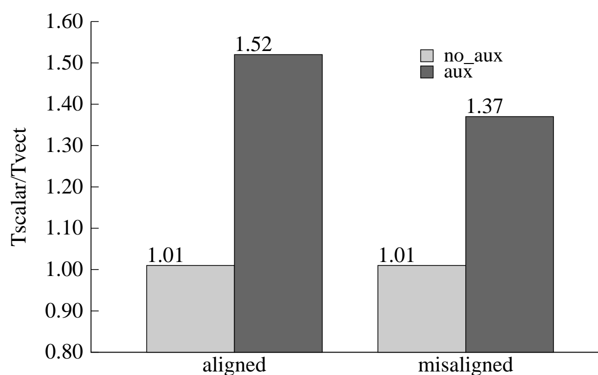

Рис. 12: Средний прирост производительности за счет векторизации на функциях EML

Кроме того, эффективность предложенных преобразований была исследована на 373 функциях, входящих в состав высокопроизводительной библиотеки векторных вычис-

лений EML(Elbrus Math Library) [17]. Данные функции реализуют наиболее распространенные операции над векторами и матрицами. На рис. 12 представлена величина прироста производительности за счет векторизации, усредненная по всем функциям EML. Замеры производительности выполнялись как для выровненных данных (обозначены как *aligned*), так и для невыровненных данных (*misaligned*). Вспомогательные преобразования позволяют значительно увеличить производительность функций EMLв основном за счет динамического разогревания циклов.

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] Krall, A., Lelait, S. Compilation techniques for multimedia processors. Int. J. Parallel Program., volume 28, № 4:pp 347–361. — 2000

[3] 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

[4] Волконский, В., Ермолицкий, А., Ровинский, Е. Развитие метода векторизации циклов при помощи оптимизирующего компилятора. Информационные технологии и вычислительные системы, , № 8:pp 34–56. — 2005

[5] Nuzman, D., Henderson, R. Multi-platform auto-vectorization. CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, pp 281–294. IEEE Computer Society, Washington, DC, USA, 2006

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

[7] Ермолицкий, А., Шлыков, С. Автоматическая векторизация выражений оптимизирующим компилятором. Приложение к журналу «Информационные технологии», , № 11. — 2008

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

[9] Wu, P., Eichenberger, A.E., Wang, A. Efficient simd code generation for runtime alignment and length conversion. CGO '05: Proceedings of the international symposium on Code generation and optimization, pp 153–164. IEEE Computer Society, Washington, DC, USA, 2005

[10] 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

[11] Дроздов, А., Корнев, Р., Боханко, А. Индексный анализ зависимостей по данным. Информационные технологии и вычислительные системы, , № 3:pp 27–37. — 2004

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

[13] Bilardi, G., Pingali, K. The static single assignment form and its computation. techreport. — 1999

[14] Bilardi, G., Pingali, K. Algorithms for computing the static single assignment form. J. ACM, volume 50, № 3:pp 375–425. — 2003

[15] Дроздов, А., Новиков, С. Эффективный алгоритм построения формы статического единственного присваивания. Информационные технологии, , № 3. — 2005

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

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