# Методы распараллеливания программ в оптимизирующем компиляторе для ВК семейства Эльбрус

***В.Ю.Волконский***, *ОАО «ИНЭУМ им. И.С.Брука», нач. отделения, vol@mcst.ru;*
***А.В.Брегер***, *ЗАО «МЦСТ», научн. сотр., breger@mcst.ru;*
***А.В.Грабежной***, *ЗАО «МЦСТ», нач. сектора, grab_av@mcst.ru;*
***А.В.Ермолицкий***, *ЗАО «МЦСТ», ст. научн. сотр., era@mcst.ru;*
***Л.Е.Муханов***, *ЗАО «МЦСТ», ст. научн. сотр., mukhanov@mcst.ru;*
***М.И.Нейман-заде***, *ЗАО «МЦСТ», зам. нач. отделения, muradnz@mcst.ru;*

## 1. Введение

Большинство современных микропроцессорных архитектур использует различные методы повышения производительности исполняемых программ за счет их распараллеливания, а именно:

- конвейеризация исполнения операций – разбиение процесса исполнения на стадии (такты) и одновременное исполнение операций, находящихся на разных стадиях конвейера;
- параллельное (одновременное) исполнение нескольких операций, находящихся на одной стадии конвейерного исполнения;
- применение одной операции к нескольким данным одновременно;
- поддержка многопоточного исполнения внутри одного процессорного ядра, на нескольких процессорных ядрах или в многопроцессорной системе, работающей на общей памяти;
- поддержка неоднородной многоядерной или многопроцессорной системы, в которой ядра могут работать одновременно, используя при этом локальную или общую память.

Эти методы распараллеливания требуют поддержки в оптимизирующих компиляторах, т.к. обычные языки программирования, такие как C, C++, Fortran (есть операции над векторами), являются языками последовательного программирования. В работе рассматриваются методы распараллеливания программ в компиляторах, которые ориентируются на аппаратную поддержку указанных видов параллелизма, особенно коротких векторов и потоков управления на общей памяти. Для исследования используется оптимизирующий компилятор с языков C, C++, Fortran, реализованный для микропроцессорных архитектур «Эльбрус» и «МЦСТ-R» (совместима с архитектурой SPARC) [1]. Обе архитектуры поддерживают конвейерное параллельное исполнение операций, набор целочисленных и вещественных операций над короткими векторами, а

также многоядерность и многопроцессорность на общей памяти с когерентным доступом. Результаты представлены для архитектуры «Эльбрус» (и вычислительного комплекса (ВК) «Эльбрус-3М» [2]), т.к. она обеспечивает наиболее полную аппаратную поддержку данных видов параллелизма, и поддержку со стороны оптимизирующего компилятора, применяемую в основном к циклам, т.к. на них приходится значительное время работы программ.

## 2. Программная конвейеризация

Архитектура микропроцессора «Эльбрус» поддерживает одновременное выполнение до 23 операций за такт. Операции выполняются в конвейере, поэтому с учетом всех особенностей аппаратуры на различных стадиях исполнения одновременно могут находиться несколько сот операций. Такой значительный параллелизм требует существенной поддержки со стороны оптимизирующего компилятора. Распараллеливание вычислений на уровне операций является основным методом повышения производительности компилируемых программ.

Наиболее полного использования возможностей архитектуры удается достичь при распараллеливании циклов. Для этого используется метод программной конвейеризации цикла, суть которого состоит в том, что итерации цикла совмещаются при исполнении с небольшим сдвигом по отношению друг к другу. Такой способ планирования применяется к самым внутренним циклам, которые повторяются много раз (от нескольких десятков до сотен и более).

Программная конвейеризация цикла позволяет в несколько раз повысить скорость его исполнения по сравнению с последовательным выполнением итераций. Она применима для любой конвейерной архитектуры, но, как правило, для реализации требуется программная раскрутка цикла и программная открутка пролога и эпилога. Кроме этого для сокращения потерь от промахов в кэш-памяти требуется дополнительная программная предварительная подкачка данных. Именно так реализована конвейеризация циклов для архитектуры «МЦСТ-R».

В архитектуре «Эльбрус» программная конвейеризация имеет *мощную аппаратную поддержку*, включающую асинхронную предварительную подкачку данных, базируемые регистры, управление прологом и эпилогом. Конвейеризация циклов с аппаратной поддержкой дает дополнительный значительный прирост производительности (в **2,5-3** раза на задачах с интенсивной обработкой массивов в циклах) по сравнению с чисто программными методами конвейеризации.

Для конвейеризации циклов в компиляторе реализуются несколько видов анализа зависимостей, в первую очередь анализ зависимостей по данным, включая зависимости между обращениями в память на разных итерациях цикла, а также анализ потока управления и потока данных.

Эти же виды анализа применяются для векторизации и распараллеливания программ, поэтому более подробно они рассмотрены в последующих разделах.

### 3. Автоматическая векторизация

В архитектуре микропроцессора «Эльбрус», также как и в большинстве современных микропроцессоров, присутствуют короткие векторные инструкции. Суть этих инструкций заключается в параллельном исполнении нескольких одинаковых операций над векторами упакованных данных. Как показывает практика, использование векторных инструкций позволяет значительно увеличить производительность процессора на задачах, в которых присутствует параллелизм на уровне данных.

В качестве примера рассмотрим цикл вычисления максимума:

```
unsigned char *a, *b,
*c;
…
for( i = 0; i < N; i++ )
{
  if( a[i] > b[i] )
    c[i] = a[i];
  else
    c[i] = b[i];
}
```

```
for( i = 0; i < N; i+=8 )
{
 c[i:i+7] =
  PMAXUB( a[i:i+7],
b[i:i+7]);
}
```

Векторная версия данного цикла (справа) исполняется значительно быстрее скалярной версии (слева) за счёт параллельного исполнения 8 итераций скалярного цикла за одну итерацию векторного (ради простоты считаем, что N кратно 8). Векторная инструкция `PMAXUB` принимает в качестве аргументов два вектора, содержащих по 8 байтовых элементов, и выдает в качестве результата 8-байтовый вектор. i-ый элемент результата вычисляется как максимум из i-ых элементов аргументов.

Внедрение векторных инструкций в код программы посредством ассемблерных вставок либо вызовов специальных библиотечных функций представляет собой достаточно трудоёмкую задачу, требующую высокой квалификации разработчиков программного обеспечения. Поэтому в оптимизирующем компиляторе для архитектуры Эльбрус встроена мощная система автоматической векторизации [3,4], позволяющая в большинстве случаев избавить программиста от необходимости ручной вставки векторных инструкций в код программы. Эта система автоматически находит участки программы, которые могут быть векторизованы, и заменяет в них скалярные инструкции векторными. При этом от программиста не требуется каких-либо дополнительных усилий.

### 3.1. Базоые средства векторизации

Процесс автоматической векторизации в оптимизирующем компиляторе для архитектуры Эльбрус состоит из следующих фаз: (1) канонизация промежуточного представления, (2) вспомогательные преобразования циклов, (3) анализ и раскрутка циклов, и (4) генерация векторного кода. При этом последние три фазы плотно взаимодействуют друг с другом.

На *фазе канонизации* выполняется множество классических оптимизаций, таких как сбор общих подвыражений и минимизация разветвлений управления. Кроме того, на этой фазе распознаются идиомы специальных семантических конструкций, таких как вычисление максимума или сложение с насыщением результата; эти конструкции заменяются специальными фиктивными инструкциями. Основная цель данной фазы – упростить последующий анализ программы.

На *фазе анализа* компилятор производит анализ потоков управления, потоков данных, диапазонов значений, зависимостей по данным, а также анализ выровненности инструкций обращения к памяти. При этом последние два вида анализа являются межпроцедурными. Результатом анализа является множество участков программы, в которых присутствует параллелизм на уровне данных.

Анализ зависимостей по данным представляет собой серию проверок, сложность которых прогрессивно увеличивается. Вначале компилятор пробует определить независимость пары обращений к памяти с помощью простых проверок, имеющих сложность O(1). Если простые проверки не дают положительного результата, запускаются более сложные. В конечном счете, компилятор решает задачу пересечения адресов как систему линейных диофантовых уравнений с помощью метода Фурье-Моцкина.

Крайне важным для системы автоматической векторизации является анализ выровненности инструкций обращения к памяти. В архитектуре Эльбрус обращение в память по невыровненному адресу приводит к возникновению исключительной ситуации либо к блокировке конвейера (в зависимости от режима исполнения программы). При векторизации происходит расширение формата инструкций обращений к памяти, в результате чего они могут стать невыровненными. Для того чтобы избежать этого, компилятор в таких случаях использует специальную технику векторизации, что влечет за собой значительное снижение эффективности конечного кода. Анализ выровненности позволяет определить, станет ли инструкция невыровненной в результате векторизации. Таким образом, данный анализ позволяет в ряде случаев избежать высоких накладных расходов, возникающих при векторизации обращений к памяти.

Если анализ цикла определил, что он может быть векторизован, компилятор далее раскручивает этот цикл и заменяет в нём группы скалярных инструкций соответствующими векторными инструкциями.

В качестве примера рассмотрим следующий цикл. Будем считать, что на фазе анализа компилятор установил выровненность и независимость указателей **a** и **b** (цикл слева). Тогда компилятор раскручивает цикл (в центре) и заменяет пару скалярных операций умножения векторной операцией PFMUL, пару скалярных операций чтения из массива **a[]** – векторным чтением, пару записей в массив **b[]** – векторной операцией записи (цикл справа). При этом инвариантный скаляр **x** заменяется векторным **X**, значение которого вычисляется перед циклом.

```
float *a, *b, x;
…
for( i=0; i<N;
i++)
{
  b[i] = a[i]*x;
}
```

```
N′ = (N/2)*2;
for( i=0; i<N′;
i+=2)
{
  b[i] = a[i]*x;
  b[i+1] =
a[i+1]*x;
}
if( N′ < N )
{
  b[i] = a[i]*x;
}
```

```
N′ = (N/2)*2;
X = {x, x};
for( i=0; i<N′;
i+=2)
{
  b[i:i+1] =
    PFMUL(
a[i:i+1], X);
}
if( N′ < N )
{
  b[i] = a[i]*x;
}
```

### 3.2. Вспомогательные преобразования

Наряду со статическим анализом, оптимизирующий компилятор для архитектуры Эльбрус позволяет получать информацию о компилируемой задаче динамически, во время её исполнения. Важность динамических проверок сложно переоценить, поскольку без них невозможно получение высокопроизводительного векторного кода для большинства реальных задач.

В случае если статический анализ не позволяет определить независимость обращений к памяти в цикле, компилятор может создать копию цикла и динамическую проверку зависимостей по данным, передающую управление на одну из версий цикла. Далее версия цикла без зависимостей векторизуется, а версия с зависимостями остаётся скалярной.

Аналогичным образом компилятор поступает в случае, когда статический анализ не позволяет определить выровненность некоторых обращений к памяти в цикле. В этом случае создается копия цикла и динамическая проверка, передающая управление на одну из версий цикла в зависимости от выровненности обращений к памяти. При этом обе версии цикла могут быть векторизованы, однако цикл с выровненными обращениями к памяти векторизуется значительно более эффективно.

Таким образом, во время исполнения программы выбирается наиболее эффективная версия цикла. Кроме того, в некоторых случаях компилятор способен выровнять обращения к памяти с помощью открутки нескольких первых итераций цикла.

В случае если цикл раскручен вручную программистом, компилятор выполняет скрутку – преобразование, обратное раскрутке цикла. Это позволяет применять к циклу другие вспомогательные преобразования.

### 3.3. Векторизация рекуррентных выражений

Рекуррентным называется выражение в цикле, потребляющее свой результат, вычисленный на одной из предыдущих итераций цикла. В общем случае такие выражения принципиально не могут быть векторизованы. Тем не менее, оптимизирующий компилятор для архитектуры Эльбрус позволяет векторизовать рекуррентные выражения в важном частном случае редукции. Рассмотрим в качестве примера цикл сложения элементов массива:

```
unsigned char *a;
int x;
...
for( i=0; i<N; i++ ){
  x = x + a[i];
  if( x > 0xFF )
      x = 0xFF;
}
```

```
X[0:7] = 0;
for( i=0; i<N; i+=8 ){
  X[0:7] =
     PADDUSB( X[0:7], a[i:i+7]);
}
for( j=0; j<8; j++ ){
  x = x + X[j];
  if( x > 0xFF )
      x = 0xFF;
}
```

В исходном цикле значение x, вычисленное на i-ой итерации, используется на (i+1)-ой итерации. Векторизация такого цикла заключается в создании перед циклом инициализации векторной переменной X, хранящей частичные суммы, вычислении частичных сумм в цикле при помощи векторной инструкции сложения с насыщением результата PADDUSB, и суммировании частичных сумм после основного цикла.

### 3.4. Векторизация циклов с разветвлениями управления

Компилятор для архитектуры Эльбрус позволяет векторизовать циклы с произвольными разветвлениями управления. Для этого используются инструкции векторного сравнения, вырабатывающие битовые маски, элементы которых заполнены единичными либо нулевыми битами в зависимости от результата сравнения отдельных элементов сравниваемых векторов.

В качестве примера рассмотрим цикл вычисления минимума двух массивов:

| | |
|---|---|
| ```int *a, *b, *c;```<br>```...```<br>```for( i=0; i<N; i++ )```<br>```{```<br>```  if( a[i] > b[i] )```<br>```    c[i] = b[i];```<br>```  else```<br>```    c[i] = a[i];```<br>```}``` | ```for( i=0; i<N; i+=8 )```<br>```{```<br>```  P[0:1] =```<br>```   PCMPGTW( a[i:i+1], b[i:i+1] );```<br>```  X[0:1] =```<br>```   PAND( b[i:i+1], P[0:1] );```<br>```  Y[0:1] =```<br>```   PANDN( a[i:i+1], P[0:1] );```<br>```  c[i:i+1] =```<br>```   POR( X[0:1], Y[0:1] );```<br>```}``` |

Для векторизации такого цикла используется операция векторного сравнения «больше» PCMPGTW, работающая с векторами, состоящими из двух слов. Полученная в результате векторного сравнения маска P[0:1] используется в векторных операциях PAND, PANDN и POR, выполняющих логические операции «и», «и-не» и «или» соответственно.

Кроме этого, компилятор способен векторизовать циклы, содержащие выходы не по счётчику. Суть векторизации таких циклов заключается в реорганизации кода внутри цикла и создании специального компенсирующего кода, исполняемого при выходе из цикла не по счётчику.

### 3.5. Экспериментальные результаты

Эффективность системы автоматической векторизации в оптимизирующем компиляторе для архитектуры Эльбрус проверялась на задачах стандартных тестовых пакетов SPEC. Максимальный прирост производительности составил **83%**, **61%**, **117%** и **11%** на задачах из пакетов SPEC CINT92, SPEC CFP95, SPEC CINT95 и SPEC CINT2000, соответственно.

Кроме того, эффективность автоматической векторизации исследовалась на функциях высокопроизводительной библиотеки векторных вычислений EML. Для исследования использовались 373 функции этой библиотеки, реализующие наиболее распространённые операции над векторами и матрицами. Средний прирост производительности за счёт автоматической векторизации этих функций составил **52%** и **37%** в случае выровненных и невыровненных входных данных, соответственно. Скорость работы отдельных функций удалось повысить почти в **10 раз**.

## 4. Автоматическое распараллеливание

В архитектуре микропроцессора Эльбрус имеются средства поддержки когерентного доступа в общую память для многоядерных и многопроцессорных систем. Поддержка этих средств в оптимизирую-

щем компиляторе позволяет дополнительно повысить производительность программ, в которых используется интенсивная обработка информации в циклах.

### 4.1. Общее описание функциональности

Разработанная в оптимизирующем компиляторе техника автоматического распараллеливания [5,6] позволяет проводить распараллеливание последовательных задач, реализованных на языках C/C++. В рамках данной техники компилятор выделяет участки последовательной программы, которые могут быть распараллелены. На данный момент в компиляторе реализована только техника автоматического распараллеливания циклов, так как в большинстве вычислительных задач наибольшую выгоду приносит распараллеливание циклов. В разработанной технике циклы, подходящие для распараллеливания, вырезаются компилятором в отдельные процедуры. В данные процедуры передается управляющий параметр (идентификатор потока), с помощью которого определяется исполняемая ветка и остальные параметры процедуры, передающиеся через стек. Использование техники выреза циклов в отдельные процедуры позволяет отказаться от использования дополнительных инструкций во внутреннем представлении компилятора, с помощью которых обозначается параллельный участок (подобный подход использован в компиляторе Intel). Недостаток такого подхода заключается в том, что возникает необходимость адаптировать все оптимизации компилятора для работы с введенными инструкциями. В результате, данная адаптация может негативно сказаться на производительности отдельных оптимизаций.

Автоматическое распараллеливание может проводиться на любое количество потоков. В частности, для комплекса Эльбрус Е3М распараллеливание проводится на два потока. На рис. 1 показана схема исполнения распараллеленной программы. В данной схеме используются интерфейсы “EPL_INIT”, “EPL_SPLIT”, “EPL_JOIN”, “EPL_SEND_SYNCHR”, “EPL_SYNCHR”, “EPL_FINAL”, которые являются частью разработанной библиотеки поддержки автоматического распараллеливания. Основное назначение данных интерфейсов заключается в создании (удалении) потоков и синхронизации действий между ними. Например, интерфейс “EPL_INIT” создает второй поток, который после создания дожидается указателя на код и соответствующего разрешения от основного потока (“EPL_SPLIT”) на его исполнение (интерфейс “EPL_RECEIVE_EVENT”). В свою очередь первый поток после исполнения своей части распараллеленного цикла дожидается окончания исполнения второй части цикла во втором потоке (“EPL_WAIT_SYNCHR”).

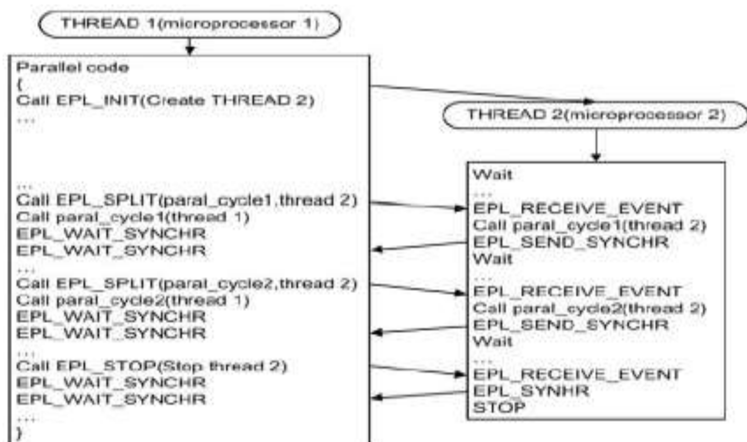


***Рис. 1.*** *Схема исполнения распараллеленной программы*

## 4.2. Техника автоматического распараллеливания

Все циклы распараллеливаются по базовой индуктивности, которая определяет количество итераций цикла.

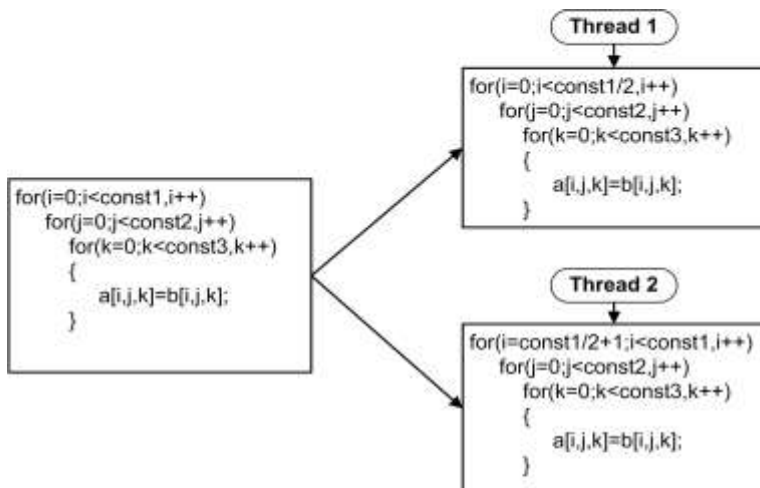


***Рис. 2.*** *Пример распараллеленного гнезда циклов*

В левой части рис. 2 приведено гнездо циклов, состоящее из трех вложенных в друг друга циклов. На данном рисунке переменные i, j, k соответствуют базовым индуктивностям этих циклов. Базовая индуктивность должна быть представлена в следующей форме:

induct_var = oper(induct_var,const)

В данном соотношении “induct_var” соответствует переменной базовой индуктивности, “const” соответствует константе, а “oper” соответствует операции сложения или вычитания. Таким образом, при распараллеливании верхняя граница базовой индуктивности (которая может быть и переменной в процедуре цикла, но обязательно инвариантной внутри цикла) делится на два. В результате первая половина цикла исполняется на первом потоке, а вторая половина цикла исполняется на втором потоке. Необходимо отметить, что разработанная техника автоматического распараллеливания позволяет распараллеливать как отдельный цикл, так и целое гнездо циклов. На рис. 2 приведен пример распараллеленного гнезда циклов по базовой индуктивности охватывающего цикла (i).

Для поиска и корректного распараллеливания циклов используются несколько типов анализа, которые базируются на использовании аналитических структур оптимизирующего компилятора:

- Анализ потока управления (Control-flow analysis);
- Анализ потока данных (Dataflow analysis);
- Анализ указателей (Pointer analysis) и зависимостей внутри цикла (Loop dependence analysis).

Анализ потока управления используется для нахождения сводимых циклов. Цикл является сводимым, если в нем есть только одна точка входа. Также в технике автоматического распараллеливания рассматриваются только циклы с одной обратной CFG-дугой и одной CFG-дугой, являющейся выходом из цикла.

Анализ потока данных позволяет определять особенности потока передачи данных в цикле. Цикл может быть распараллелен без дополнительных преобразований, если в нем имеется только одна редукция, соответствующая базовой индуктивности, и отсутствует какая-либо передача данных из цикла по графу потока данных. В случае если имеется одна или несколько побочных редукций, то при применении разработанной техники распараллеливается не только базовая индуктивность, но и данные редукции. В случае наличия передачи данных из цикла по графу потока данных в вырезанной процедуре, исполняемой параллельно, создаются операции сохранения этих данных в памяти. В исходной процедуре, которая содержала вырезанный цикл, эти данные восстанавливаются.

Анализ указателей и зависимостей внутри цикла позволяет выявлять меж-итерационные зависимости между операциями, которые обращаются к одному и тому же участку памяти и тем самым препятствуют распараллеливанию цикла по базовой индуктивности.

## 4.3. Библиотека поддержки для автоматического распараллеливания

Для поддержки исполнения распараллеленных программ была создана библиотека libepl, предоставляющая компилятору упрощенный интерфейс для управления потоками. Данная библиотека управляет запуском и остановкой вспомогательных потоков, распределением заданий между потоками, распределением потоков по вычислительным ядрам и синхронизацией между ними. Реализовано несколько вариантов синхронизации, переключение между которыми производится перед запуском распараллеленной программы. Наибольшую производительность в задачах с короткими параллельными участками показывает синхронизация при помощи активного ожидания (spinlock), обеспечивающая накладные расходы на уровне нескольких сот тактов на один параллельный участок, что позволяет распараллеливать гнезда циклов или даже одиночные циклы, время исполнения которых измеряется, начиная от тысячи тактов. Благодаря минимальным накладным расходам удается распараллеливать больше циклов и таким образом сокращать ту часть кода, которая исполняется последовательно.

Для целей отладки и измерения производительности создано две вспомогательные версии библиотеки, одна из которых проверяет корректность работы автоматического распараллеливания, а другая измеряет длительность исполнения параллельных участков программы и накладные расходы на синхронизацию и организацию многопоточной обработки.

Кроме этого были созданы библиотеки динамической поддержки интерфейса OpenMP для программного распараллеивания с расширениями OpenMP для языков C, C++ и Fortran, а также библиотеки поддержки распараллеливания на системах с распределенной памятью. В отличие от библиотеки libepl эти библиотеки используют стандартные средства распараллеливания, не удерживающие поток на ядре, что приводит заметному увеличению времени, необходимому для синхронизации. С другой стороны, использование расширений OpenMP снабжает компилятор необходимыми подсказками относительно возможности распараллеливания циклов, которые без таких подсказок не всегда удается распараллелить только с помощью компилятора.

## 4.4. Результаты применения автоматического распараллеливания

Для оценки эффективности автоматического распараллеливания использовались задачи пакетов SPEC95 и SPEC2000. В рамках данного тестирования было проведено три различных запуска каждой из задач. В первом запуске осуществлялось исполнение последовательной версии задачи, скомпилированной с использованием пиковых опций. Во вто-

ром запуске осуществлялось исполнение автоматически распараллеленной задачи, скомпилированной с использованием тех же пиковых опций. В третьем запуске осуществлялось одновременное исполнение ранее скомпилированной последовательной версии на разных ядрах комплекса Эльбрус Е3М. Третий запуск позволяет сделать оценку максимально-возможного ускорения автоматически распараллеленной задачи, учитывая влияние подсистемы памяти.

***Табл. 1***. *Статистика исполнения задач пакета SPEC95 и SPEC2000*

| BENCH | REAL | MEM_IMP | EXEC | L1 MISS | APB MISS | NOP | NO_COM |
|---|---|---|---|---|---|---|---|
| tomcatv | **1,37** | 1,75 | **1,48** | 1,46 | 1,40 | 1,37 | 1,36 |
| swim | **1,49** | 1,5 | 1,96 | 1,94 | **1,49** | 1,49 | 1,48 |
| mgrid | **1,61** | 1,87 | 1,98 | **1,85** | **1,65** | 1,63 | 1,62 |
| hydro2d | **1,30** | 1,47 | 1,71 | 1,68 | **1,26** | 1,25 | 1,25 |
| art | **1,37** | 1,63 | **1,46** | 1,38 | 1,45 | 1,41 | 1,41 |

Можно было бы предположить, что при полном распараллеливании задачи и отсутствии конфликтов при обращении в память параллельный вариант будет в два раза быстрее последовательного варианта. Но в действительности коэффициенты ускорения распараллеленных задач оказались значительно меньше 2 (табл. 2, колонка REAL – для автоматического распараллеливания и колонка MEM_IMP – для одновременного запуска двух одинаковых задач). Это объясняется тем, что при распараллеливании задачи поток запросов в память увеличивается, что приводит к *увеличению конфликтов внутри подсистемы памяти*, а следовательно и к замедлению распараллеленной версии задачи. Причем конфликты снижают производительность системы не только при распараллеливании одной задачи, но и при одновременном исполнении двух одинаковых задач.

Для анализа данной ситуации была собрана детальная статистика об исполнении последовательных и распараллеленных версий задач (Табл. 1). В данной таблице:

- поле “REAL” соответствует реальному ускорению распараллеленной задачи по сравнению с последовательным вариантом;
- поле “MEM_IMP” соответствует оценке максимально-возможного ускорения автоматически распараллеленной задачи, учитывая влияние подсистемы памяти;
- поле “EXEC” соответствует отношению количества исполненных широких команд в последовательной версии к количеству широких команд, исполненных в распараллеленной версии в первом потоке;
- поле “L1 MISS” соответствует полю “EXEC”, дополнительно учитывающему количество блокировок из-за промахов в L1 кэш в последовательной версии и в  первом потоке распараллеленной версии при исполнении операций чтения;

- поле “APB MISS” соответствует полю “L1 MISS”, дополнительно учитывающему количество блокировок, возникших в основном из-за APB промахов и промахов в L2 кэш, в последовательной версии и в первом потоке распараллеленной версии;
- поле “NOP” соответствует полю “APB MISS”, дополнительно учитывающему количество пустых команд, исполненных в последовательной версии и в первом потоке распараллеленной версии;
- поле “NO COMMAND” соответствует полю “NOP”, дополнительно учитывающему количество блокировок из-за промахов в L1 кэш команд в последовательной версии и в первом потоке распараллеленной версии.

Прирост производительности ВК при одновременном исполнении двух одинаковых задач на самой лучшей конфигурации составил, в среднем, **1,63**, что значительно меньше 2. Но этот результат характерен только для задач со столь интенсивным потоком обращений в память

Абсолютный прирост производительности за счет распараллеливания на пяти рассмотренных задачах составляет, в среднем, **1,42**. Эффективность автоматического распараллеливания оптимизирующим компилятором приведенных задач можно оценить на основе коэффициентов поля “EXEC”, так как именно этот показатель позволяет соотнести количество широких команд, исполненных в последовательной версии и в распараллеленной версии. Для задач “swim” и “mgrid” данный коэффициент близок к 2. Это значит, что задачи были почти полностью распараллелены. В случае задач “tomcatv” и “hydro2d” подобного результата не удалось достичь, так как значимая часть времени исполнения данных задач уходит на исполнение функций ввода/вывода, которые не могут быть распараллелены компилятором. В случае задачи “art” такой коэффициент был получен из-за того, что не все циклы удалось распараллелить. В среднем, эффективность автоматического распараллеливания достигает **77%** по сравнения с одновременным исполнением двух задач, что подтверждает, в целом, эффективность автоматического распараллеливания.

Таким образом, поле “EXEC” определяет максимально-возможное ускорение распараллеленной задачи по сравнению с последовательным вариантом. Данные ускорения не удалось достичь на практике из-за влияния различных видов блокировок при работе с памятью, что подтверждается значениями полей “L1 MISS” и “APB MISS”. Истинная природа этих блокировок, скорее всего, связана с влиянием всей подсистемы памяти в целом, что является предметом будущего исследования.

## 5. Заключение

В работе рассмотрены методы программной конвейеризации, автоматической векторизации и распараллеливания программ в оптимизирующем компиляторе. Эти методы демонстрируют высокую эффективность и позволяют существенно повышать производительность программ, в которых используются форматы данных, позволяющие использовать операции над короткими векторами, и циклы, допускающие распараллеливание на потоки управления.

Максимальный прирост производительности за счет автоматической векторизации составил **83%**, **61%**, **117%** и **11%** на задачах из пакетов SPEC CINT92, SPEC CFP95, SPEC CINT95 и SPEC CINT2000, соответственно. Средний прирост производительности на 373 функциях библиотеки, реализующие наиболее распространённые операции над векторами и матрицами, за счёт их автоматической векторизации составил **52%** и **37%** в случае выровненных и невыровненных входных данных, соответственно. Скорость работы отдельных функций удалось повысить почти в **10 раз**.

Абсолютный прирост производительности за счет распараллеливания на пяти задачах из пакетов SPEC95, SPEC2000 составляет для двухпроцессорного ВК «Эльбрус-3М», в среднем, **1,42**, или **77%** от одновременного исполнения двух задач.

**Литература**

1. Кузьминский М.Б. Куда идет «Эльбрус». //Открытые системы, No. 7, 2011.
2. Ким А.К., Волконский В.Ю., Груздов Ф.А., Михайлов М.С., Парахин Ю.Н., Сахин Ю.Х., Семенихин С.В., Слесарев М.В., Фельдман В.М. Микропроцессорные вычислительные комплексы с архитектурой «Эльбрус» и их развитие, // Современные информационные технологии и ИТ-образование. Сборник докладов 3-й международной научно-практической конференции, Москва, 6-9 декабря 2008. С.12-31.
3. Ермолицкий, А., Шлыков, С. Автоматическая векторизация выражений оптимизирующим компилятором. Приложение к журналу “Информационные технологии” , No. 11, 2008.
4. Волконский, В., Ермолицкий, А., Ровинский, Е. Развитие метода векторизации циклов при помощи оптимизирующего компилятора. // Высокопроизводительные вычислительные системы и микропроцессоры. Сборник научных трудов ИМВС РАН. Вып. 8, 2005. С. 34-56.
5. Mukhanov L., Ilyin P., Ermolitsky A., Grabezhnoy A., Shlykov S., Breger A. Thread-level Automatic Parallelization in the Elbrus Optimizing Compiler. // Parallel and Distributed Computing and Systems (PDCS-2010), The IASTED International Conference on Informatics 2010 series. 2010.
6. Волконский В.Ю., Грабежной А.В., Муханов Л.Е., Нейман-заде М.И. Исследование влияния подсистемы памяти на производительность распараллеленных программ. //Вопросы радиоэлектроники, сер. ЭВТ, 2011, вып. 3. С.22-37.