Проектирование расширения обработки сигналов для универсального микропроцессора

А. А. ЛЕСНЫХ
Московский государственный университет им. М. В. Ломоносова

И. А. ШИРОКОВ
Научно-исследовательский институт системных исследований РАН
e-mail: shirokov@gmail.com

УДК 519.852+519.712.2

Ключевые слова: параллельное программирование, потоковое программирование, системы обработки сигналов.

Аннотация

В статье описывается расширение цифровой обработки сигналов для универсального 64-разрядного процессора К 64-СМП, разрабатываемого в НИИСИ РАН. Это расширение позволило увеличить производительность микропроцессора на задачах обработки сигналов до трёх раз. Также описывается схема потокового программирования процессоров, которая использовалась при проектировании расширения.

Abstract

A. A. Lesnykh, I. A. Shirokov, Designing digital signal processing extension for multipurpose microprocessor, Fundamentalnaya i prikladnaya matematika, vol. 12 (2006), no. 8, pp. 159—180.

The digital signal processing extension for the multipurpose 64 bit processor K 64-SMP designed by the Institute for System Research of the Russian Academy of Science is discussed. The proposed extension increases the processor efficiency for signal processing by a factor of 3. We discuss also the stream programming scheme utilized in the extension programming.

Схема потокового программирования

Первоначально методы потокового программирования разрабатывались в НИИСИ РАН при проектировании суперЭВМ с вычислительными узлами на основе сопроцессора вещественной арифметики СЭВМ-6 (см. [2]). В основе этих методов лежат исследования работ [3—6,8—10]. В дальнейшем схема потокового программирования активно использовалась при проектировании различных специализированных и универсальных микропроцессоров, разрабатываемых в НИИСИ РАН, в том числе и при проектировании расширения обработки сигналов для процессора К 64-СМП. Мы приводим описание основных принципов схемы потокового программирования.

Фундаментальная и прикладная математика, 2006, том 12, № 8, с. 159—180.

Реализация для одного конвейеризованного вычислительного узла

Излагаемая далее схема потокового программирования (ПП) строится для следующей модели арифметического устройства (АУ). АУ имеет конвейер длины n, любая команда выполняется за n тактов, причём каждый такт на АУ можно запускать новую команду.

Схема ПП позволяет вести на АУ одновременно n потоков вычислений, причём с точки зрения каждого потока вычисления ведутся на АУ без конвейера, результат любой команды одного потока доступен уже для следующей команды этого потока на следующем такте. Реализуется схема следующим образом. Время использования АУ делится между потоками: первый поток использует АУ только на тактах с номерами nk, второй поток — на тактах с номерами $nk+1$, третий — $nk+2$ и т. д. Таким образом, между двумя последовательными командами одного потока проходит ровно n тактов, за это время завершится выполнение предыдущей команды и будет доступен её результат.

В качестве примера рассмотрим реализацию одной строки программы sPPM (см. [2, п. 5.2]) на сопроцессоре вещественной арифметики ВУ СЭВМ-6 (см. [2, приложение 5]). На сопроцессоре имеются четыре пары сложитель-умножитель. В качестве одного вычислительного устройства будем рассматривать одну такую пару. Заметим, что длина конвейера сложителя равна 2, однако наличие FIFO на два элемента на его выходе позволяет считать, что длина конвейера равна 4. Таким образом, пара сложитель-умножитель полностью укладывается в модель АУ, описанную выше: длина конвейера АУ равна 4 и каждый такт на вход АУ можно давать новую команду. На одной паре сложитель-умножитель во всех четырёх потоках будем выполнять одни и те же вычисления, но для разных данных.

Реализуем на паре сложитель-умножитель следующую строку на фортране:

```
dffuse(i) = difcon * abs (ddxux + ddyuy + ddzuz).
```

Сначала разложим эту строку на элементарные команды — команды, которые могут выполняться непосредственно на сопроцессоре:

```
ddxux + ddyuy
AF + ddzuz
1 X MO
difcon * MO
MO -> dffuse(i)
```

Здесь использованы следующие обозначения: `AF` — выходной регистр FIFO сложителя; `MO` — выходной регистр умножителя; `+`, `*`, `X`, `->` — операции сложения, умножения, умножения знака на модуль, пересылки соответственно.

Повторив каждую команду четыре раза, получаем следующую программу для одной пары сложитель-умножитель:

```
 1   ddxux_1 + ddyuy_1
 2   ddxux_2 + ddyuy_2
 3   ddxux_3 + ddyuy_3
 4   ddxux_4 + ddyuy_4
 5   AF + ddzuz_1
 6   AF + ddzuz_2
 7   AF + ddzuz_3
 8   AF + ddzuz_4
 9   1 X MO
10   1 X MO
11   1 X MO
12   1 X MO
13   difcon_1 * MO
14   difcon_2 * MO
15   difcon_3 * MO
16   difcon_4 * MO
17   MO -> dffuse(i)_1
18   MO -> dffuse(i)_2
19   MO -> dffuse(i)_3
20   MO -> dffuse(i)_4
```

Здесь `name_i` — это переменная `name` потока `i`.

Реализация для нескольких арифметических узлов

Схема ПП хорошо подходит для параллельных вычислений, когда имеется несколько одинаковых АУ, соответствующих модели, описанной выше, и когда необходимо выполнить одинаковые вычисления над разными данными. В таком случае все потоки, выполняемые на одном АУ, одинаковы (как в предыдущем примере) и программы, выполняемые на различных АУ, также одинаковы. Тем самым достигается хорошая сбалансированность вычислений: все АУ начинают и заканчивают работать одновременно.

В качестве примера можно указать реализацию на сопроцессоре вещественной арифметики ВУ СЭВМ-6 алгоритма быстрого преобразования Фурье (см. [7, п. 1]). В этой реализации различные пары сложитель-умножитель и различные потоки на одной паре просчитывают различные итерации внешнего цикла.

Реализация для нескольких арифметических узлов при наличии разделяемого общего ресурса

Пусть k АУ с конвейером длины n конкурируют за общий ресурс, к которому можно обращаться каждый такт. Тогда, чтобы не возникало конфликтов между различными потоками на различных АУ, схему ПП для нескольких АУ, описанную в предыдущем пункте, необходимо подправить следующим обра-

зом. Во-первых, время использования общего ресурса делится между потоками: 1-е АУ обращается к общему ресурсу только на тактах с номерами $nkl,\ldots,$ $nkl+n-1$, 2-е АУ — на тактах с номерами $nkl+n,\ldots,$ $nkl+2n-1$, 3-е АУ — на тактах $nkl+2n,\ldots,$ $nkl+3n-1$ и т. д. Во-вторых, так как на различных АУ выполняются одинаковые программы, а время использования общего ресурса разделено указанным образом, то программы на различных АУ запускаются не одновременно, а с задержкой в n тактов: 2-е АУ начинает работу через n тактов после 1-го, 3-е — через n тактов после 2-го и т. д. Наконец, программа для одного потока, команды которой затем будут повторены по n раз для разных потоков на одном АУ, пишется таким образом, чтобы обращение к общему ресурсу происходило только в командах с номерами $kl+1$. Тогда после повторения каждой команды n раз для различных потоков АУ будет обращаться к общему ресурсу только в отведённое ему время. Эта схема в случае $n=2$ и $k=3$ показана на рис. 1. Показана программа для одного потока, программа для одного АУ и потактовая диаграмма работы всех АУ. Команды, имеющие возможность обращаться к общему ресурсу, выделены жирным шрифтом.

В качестве примера рассмотрим делитель сопроцессора вещественной арифметики ВУ СЭВМ-6 (см. [2, приложение 6]). Можно считать, что он является разделяемым между различными парами сложитель-умножитель общим ресурсом. В данном случае $n=4$ и $k=4$. Тогда программа для одного потока пишется так, чтобы команда деления встречалась только в командах с номерами $4l+1$. Вычисления на различных парах начинаются с задержкой в четыре такта.

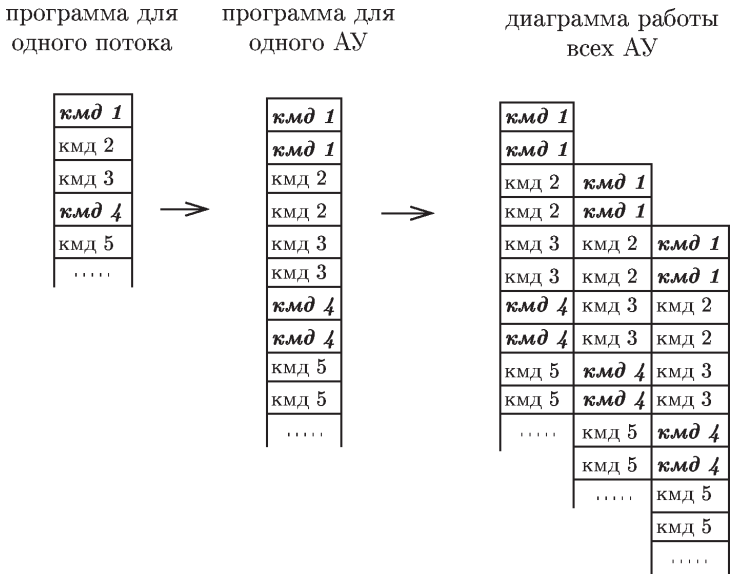

Рис. 1. Пример реализации схемы ПП

Реализация для вычислительных узлов с косвенной адресацией памяти

Использование схемы ПП возможно и в случае, если адресация памяти производится посредством регистров адреса и автоинкремента. Ниже мы покажем, какие изменения необходимо внести в схему ПП, чтобы иметь произвольный доступ к памяти в типичных ситуациях работы с памятью. Будем предполагать, что все потоки на одном АУ выполняют одинаковые вычисления, как в примере на рис. 1.

Доступ к памяти в линейных программах

В линейной (без циклов) программе доступ к памяти организуется достаточно простым образом, так как адреса переменных, к которым необходимо обращаться, постоянны, а в наборе команд обычно имеются спецкоманды, позволяющие записывать постоянные значения в регистры адреса. Во-первых, одинаковые переменные разных потоков располагаются в памяти в соседних ячейках. Если какая-либо переменная 1-го потока располагается в ячейке с номером A, то соответствующая переменная 2-го потока располагается в ячейке с номером $A+1$, 3-го — с номером $A+2$ и т. д. (рис. 2). Во-вторых, автоинкременты адресов устанавливаются равными 1. В-третьих, перед каждой командой, которая обращается к какой-нибудь ячейке памяти (заметим, что следующие за ней $n-1$ команд тоже обращаются к памяти к следующим ячейкам), ставится спецкоманда установки регистра адреса на нужный адрес. Тогда эта команда (и следующие за ней $n-1$ команды, так как автоинкремент равен 1) будет обращаться к нужной ячейке памяти. Таким образом, на каждые n команд, обращающихся к памяти, имеем одну дополнительную спецкоманду, устанавливающую адрес.

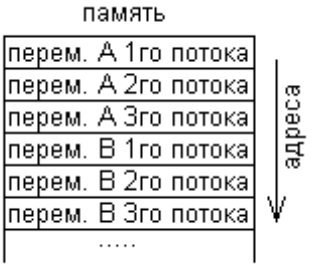

Рис. 2. Организация хранения данных

Доступ к памяти в циклах

В циклах задача доступа к нужным ячейкам памяти усложняется, так как адрес ячейки может зависеть от номера текущей итерации. Покажем, как решается эта задача при некоторых дополнительных ограничениях.

Прежде всего заметим, что цикл можно развернуть в линейную программу. Тогда доступ к памяти организуется так, как описано выше. Описанная же ниже схема доступа к памяти применяется при программировании циклов с помощью команды REPEAT (или аналогичной).

В наборе команд АУ обычно имеются специальные команды, позволяющие записывать фиксированные числа в регистры адресов и автоинкрементов. Кроме того, имеются команды, позволяющие прибавлять к этим регистрам фиксированное число. Эти команды и будут использоваться для организации доступа к нужным ячейкам памяти.

Чтобы показать, как организуется хранение данных в памяти и работа с управляющими регистрами, рассмотрим следующий пример:

```
do 100 i=1,20
c(i)=a(i)+b(i)*c(i)
100 continue
```

Пусть необходимо выполнить четыре таких цикла для различных массивов a, b и c и все эти массивы хранятся в памяти сопроцессора. Реализуем эту программу на паре сложитель-умножитель. В элементарных командах эта программа выглядит следующим образом:

```
do 100 i=1,20
b(i) * c(i)
a(i) + MO
AF -> c(i)
100 continue
```

Программа с повторенными в соответствии со схемой потокового программирования командами имеет вид

```
 1  REPEAT#12#20
 2  b(i)_1 * c(i)_1
 3  b(i)_2 * c(i)_2
 4  b(i)_3 * c(i)_3
 5  b(i)_4 * c(i)_4
 6  a(i)_1 + MO
 7  a(i)_2 + MO
 8  a(i)_3 + MO
 9  a(i)_4 + MO
10  AF -> c(i)_1
11  AF -> c(i)_2
12  AF -> c(i)_3
13  AF -> c(i)_4
```

Команда REPEAT#12#20 повторяет следующие за ней 12 команд 20 раз.

Разместим перед началом работы сопроцессора данные в памяти так, как показано на рис. 3, и добавим в программу команды, обеспечивающие доступ к памяти по необходимым адресам:

```
 1  R1=80 R2=160 W1=160 incR1=1 incR2=1 incW1=1
 2  REPEAT#14#20
 3  b(i)_1[r1] * c(i)_1[r2]
 4  b(i)_2[r1] * c(i)_2[r2]
 5  b(i)_3[r1] * c(i)_3[r2]
 6  b(i)_4[r1] * c(i)_4[r2]
 7  R1 += -84
 8  a(i)_1[r1] + MO
 9  a(i)_2[r1] + MO
10  a(i)_3[r1] + MO
11  a(i)_4[r1] + MO
12  R1 += 80
13  AF -> c(i)_1[w1]
14  AF -> c(i)_2[w1]
15  AF -> c(i)_3[w1]
16  AF -> c(i)_4[w1]
```

Здесь использованы следующие обозначения: `R1`, `R2` — регистры адреса чтения первого и второго полутактов соответственно, `W1` — регистр адреса записи первого полутакта, `incR1`, `incR2`, `incW1` — регистры автоинкрементов соответствующих адресов. В квадратных скобках показано, на каком полутакте происходит чтение/запись. Например, `a(i)_1[r1]` означает, что происходит чтение (`read`) переменной `a(i)` первого потока на первом полутакте.

Поясним работу этой программы. Во-первых, как и в примере в предыдущем пункте, указанная схема хранения данных и единичные значения автоинкрементов позволяют не выполнять спецкоманды для установки адресов перед каждой командой, в которой происходит доступ к памяти. Достаточно выполнить спецкоманду один раз перед командой первого потока, и команды остальных потоков будут также обращаться к памяти по нужным адресам.

Во-вторых, в командах программы на первом полутакте встречается чтение как массива `a`, так и массива `b`. В строках 7 и 12 происходит инкремент соответствующего регистра (`R1`) таким образом, чтобы этот регистр указывал по очереди на массивы `a` и `b`.

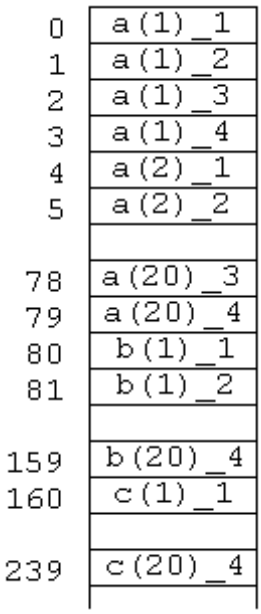

Рис. 3. Организация хранения данных

Из этого примера легко понять общую схему работы с памятью. Автоинкремент равен 1, а данные для разных потоков располагаются так же, как в примере. Перед обращением первого потока вычислений к некоторой ячейке памяти происходит инкремент нужного управляющего регистра. Величина инкремента выбирается так, чтобы доступ к памяти происходил по нужному адресу. Отметим, что данная схема применима только в том случае, когда величина инкремента не зависит от номера итерации, т. е. когда используемые в цикле переменные находятся в памяти на фиксированных, не зависящих от номера итерации цикла расстояниях. Например, цикл

```
do 100 i=1,20
c(i)=a(i+1)+b(i-2)*c(i+3)
100 continue
```

можно реализовать по этой схеме, так как расстояния между переменными `c(i)`, `a(i+1)`, `b(i-2)`, `c(i+3)` постоянны и не зависят от номера итерации. А к циклу

```
do 100 i=1,20
c(i)=a(i)+b(2*i)*c(i)
100 continue
```

применить описанную схему нельзя, так как расстояние между некоторыми переменными, например `a(i)` и `b(2*i)`, зависит от номера итерации `i`.

Нерегулярный доступ к памяти в циклах

Для эффективной работы с нерегулярным доступом к памяти в наборе команд АУ должны быть команды, позволяющие сохранять (`saved`), а затем восстанавливать (`loadd`) регистры адресов и автоинкрементов. Более того, желательно наличие команды `swapd`, позволяющей восстанавливать значения этих регистров с одновременным сохранением текущего содержимого регистров. С помощью указанных команд можно эффективно реализовать на АУ предыдущий пример, к которому не применима описанная выше схема доступа к памяти.

Итак, пусть необходимо выполнить четыре цикла

```
do 100 i=1,20
c(i)=a(i)+b(2*i)*c(i)
100 continue
```

для четырёх различных троек массивов `a`, `b` и `c` и все эти массивы хранятся в памяти сопроцессора. В элементарных командах этот цикл выглядит следующим образом:

```
do 100 i=1,20
b(2*i) * c(i)
a(i) + МО
```

```
AF -> c(i)
100 continue
```

Программа с повторенными командами имеет вид

```
 1  REPEAT#12#20
 2  b(2*i)_1 * c(i)_1
 3  b(2*i)_2 * c(i)_2
 4  b(2*i)_3 * c(i)_3
 5  b(2*i)_4 * c(i)_4
 6  a(i)_1 + MO
 7  a(i)_2 + MO
 8  a(i)_3 + MO
 9  a(i)_4 + MO
10  AF -> c(i)_1
11  AF -> c(i)_2
12  AF -> c(i)_3
13  AF -> c(i)_4
```

Размещение массивов в памяти такое же, как в предыдущем примере (см. рис. 3). Добавим в программу команды, обеспечивающие доступ к памяти по необходимым адресам:

```
 1  R1=0 R2=160 W1=160 incR1=1 incR2=1 incW1=1
 2  saved
 3  R1=84
 4  REPEAT#15#20
 5  b(2*i)_1[r1] * c(i)_1[r2]
 6  b(2*i)_2[r1] * c(i)_2[r2]
 7  b(2*i)_3[r1] * c(i)_3[r2]
 8  b(2*i)_4[r1] * c(i)_4[r2]
 9  R1 += 4
10  swapd
11  a(i)_1[r1] + MO
12  a(i)_2[r1] + MO
13  a(i)_3[r1] + MO
14  a(i)_4[r1] + MO
15  swapd
16  AF -> c(i)_1[w1]
17  AF -> c(i)_2[w1]
18  AF -> c(i)_3[w1]
19  AF -> c(i)_4[w1]
```

Поясним работу программы. В строке 1 устанавливаются управляющие регистры, причём регистр `R1`, в отличие от предыдущего примера, указывает на массив `a`. Следующей командой регистры сохраняются, а в строке 3 регистр `R1`

направляется на массив b. После команды swapd в строке 10 регистр R1 вновь указывает (остальные регистры также восстанавливаются, но они нас здесь не интересуют) на массив a, с которым и ведётся работа в строках 11—14. В строке 15 прежнее значение регистра R1 восстанавливается, и он вновь указывает на массив b.

Можно сказать, что команда swapd обеспечивает два адресных контекста, между которыми можно переключаться. Если в цикле присутствуют две переменные, расстояние между которыми зависит от номера итерации цикла, то один контекст связывается с одной переменной, другой — с другой. Например, цикл

```
do 100 i=1,20
c(i) = a(i-2)*d(2*i+3) + b(2*i)*c(i)
100 continue
```

реализуется по этой схеме так, что один контекст связывается с переменными c(i) и a(i-2), а другой — с переменными d(2*i+3) и b(2*i). Цикл

```
do 100 i=1,20
c(i) = a(i) + b(21-i)
100 continue
```

также можно реализовать по этой схеме: один контекст связывается с переменными c(i) и a(i), другой — с переменной b(21-i). Но цикл

```
do 100 i=1,20
c(i) = a(2*i) + b(21-i)
100 continue
```

нельзя реализовать по этой схеме, так как необходимо три адресных контекста: один — связанный с переменной c(i), другой — с переменной a(2*i), третий — с переменной b(21-i).

Проектирование расширение обработки сигналов

В данном разделе рассматриваются некоторые варианты усовершенствования 64-битных микропроцессорных архитектур с целью увеличения производительности на задачах обработки сигналов. Хотя описываемые усовершенствования применимы к широкому классу универсальных 64-битных процессоров, в данной статье рассматривается конкретная реализация для архитектуры микропроцессора К 64-СМП. Реализация всех усовершенствований позволила увеличить производительность процессора К 64-СМП на задачах обработки сигналов в три раза. При проектировании этого расширения мы опирались на принципы потокового программирования, описанные выше.

1. Описание усовершенствований

С точки зрения задач обработки сигналов вычислительный блок процессора К 64-СМП позволяет параллельно выполнять следующие операции:

- умножение с накоплением 64-битных чисел (a*b+c);
- загрузку/выгрузку 64-битного слова.

1.1. Усовершенствование 1

Одновременная обработка PS-команд. В этом случае вычислительный блок процессора К 64-СМП в дополнение к раннее описанным возможностям позволит параллельно выполнять следующие операции:

- одновременное умножение с накоплением двух 32-битных чисел в соседних ps-секциях 64-битного слова ($a^1*b^1+c^1$, $a^2*b^2+c^2$);
- загрузка/выгрузка 64-битного слова.

1.2. Усовершенствование 2

Одновременная загрузка/выгрузка двух соседних 64-битных слов. В этом случае вычислительный блок процессора К 64-СМП в дополнение к ранее описанным возможностям позволит параллельно выполнять следующие операции:

- одновременное умножение с накоплением двух 32-битных чисел в соседних ps-секциях 64-битного слова ($a^1*b^1+c^1$, $a^2*b^2+c^2$);
- загрузка/выгрузка двух соседних 64-битных слов в/из соседних регистров процессора.

1.3. Усовершенствование 3

Одновременное вычисление операций вида a+b и a–b, при этом вычислительный блок процессора К 64-СМП будет позволять параллельно выполнять (при некоторых ограничениях, описанных ниже) следующие операции:

- два умножения с накоплением 64-битных чисел (a*b+c, a*b–c) или четыре умножения с накоплением двух 32-битных чисел в соседних ps-секциях 64-битного слова ($a^1*b^1+c^1$, $a^1*b^1-c^1$, $a^2*b^2+c^2$, $a^2*b^2-c^2$);
- загрузка/выгрузка двух соседних 64-битных слов в/из соседних регистров процессора.

1.4. Усовершенствование 4

Вычисление операции a*b, где a и b — комплексные числа в формате 32+32, за один такт. При этом вычислительный блок процессора К 64-СМП будет позволять параллельно выполнять (при некоторых ограничениях, описанных ниже) следующие операции:

- умножение двух комплексных чисел в формате 32+32 ($a^1*b^1+a^2*b^2$, $a^1*b^2-a^2*b^1$), или два умножения с накоплением 64-битных чисел

(`a*b+c`, `a*b-c`), или четыре умножения с накоплением двух 32-битных чисел в соседних ps-секциях 64-битного слова ($\texttt{a}^1\texttt{*b}^1\texttt{+c}^1$, $\texttt{a}^1\texttt{*b}^1\texttt{-c}^1$, $\texttt{a}^2\texttt{*b}^2\texttt{+c}^2$, $\texttt{a}^2\texttt{*b}^2\texttt{-c}^2$);

- загрузка/выгрузка двух соседних 64-битных слов в/из соседних регистров процессора.

2. Преобразование Фурье

Обычно в задачах обработки сигналов имеется поток данных, которые должны быть подвергнуты преобразованию Фурье. Поэтому для повышения производительности микропроцессора допустимо одновременно выполнять несколько преобразований Фурье. Ниже рассматривается задача параллельного выполнения четырёх преобразований Фурье.

Алгоритм быстрого преобразования Фурье (БПФ) с прореживанием по времени сводится к вычислению выражений вида

$$\begin{pmatrix} A \\ B \end{pmatrix} \longmapsto \begin{pmatrix} A + B \times W \\ A - B \times W \end{pmatrix},$$

где A, B и W — комплексные числа. Данная операция обычно называется бабочкой Фурье.

Обозначим $A_\mathrm{r} = \operatorname{Re} A$, $A_\mathrm{i} = \operatorname{Im} A$, $B_\mathrm{r} = \operatorname{Re} B$ и т. д. Бабочка Фурье записывается в виде

$$\begin{pmatrix} A_\mathrm{r} & A_\mathrm{i} \\ B_\mathrm{r} & B_\mathrm{i} \end{pmatrix} \longmapsto \begin{pmatrix} A_\mathrm{r} + (B_\mathrm{r} \times W_\mathrm{r} - B_\mathrm{i} \times W_\mathrm{i}) & A_\mathrm{i} + (B_\mathrm{r} \times W_\mathrm{i} + B_\mathrm{i} \times W_\mathrm{r}) \\ A_\mathrm{r} - (B_\mathrm{r} \times W_\mathrm{r} - B_\mathrm{i} \times W_\mathrm{i}) & A_\mathrm{i} - (B_\mathrm{r} \times W_\mathrm{i} + B_\mathrm{i} \times W_\mathrm{r}) \end{pmatrix}.$$

Одновременно с вычислением и загрузкой/выгрузкой данных процессор К 64-СМП позволяет вычислять адрес и выполнять предварительную выборку слова из памяти. Это позволяет обойти проблемы, связанные с нерегулярным доступом к данным, возникающие при обработке преобразования Фурье.

Для достижения максимальной производительности процессора используется схема потокового программирования: процессор одновременно выполняет четыре различных преобразования Фурье. Предполагается, что все данные для четырёх преобразований Фурье находятся в L1-кэше, а 32-битные данные лежат в кэше следующим образом: $\texttt{A(1)}^1$, $\texttt{A(1)}^2$, $\texttt{A(1)}^3$, $\texttt{A(1)}^4$, $\texttt{A(2)}^1$, $\texttt{A(2)}^2$, $\texttt{A(2)}^3$, $\texttt{A(2)}^4$, $\texttt{A(3)}^1$,..., где $\texttt{A(i)}^\texttt{j}$ — `i`-й элемент массива `A` для преобразования Фурье номер `j`.

3. Оценка производительности предлагаемых архитектур

3.1. Исходная архитектура

Возможности архитектуры:

- умножение с накоплением 64-битных чисел (`a*b+c`);
- загрузка/выгрузка одного 64-битного слова.

В регистровом файле имеются четыре порта на чтение и три на запись (4/3).

Вычисления организуются следующим образом (обработка элементов с индексами k и $k+1$, B_{i}^{k}, B_{i}^{k+1}, W_{i}, W_{r} уже загружены на предыдущей итерации (см. такты 7—10)):

1	$B_{\mathrm{r}}^{k} \to R_{04}$;	$R_{00} * R_{02} \to R_{10}$ $(B_{\mathrm{i}} * W_{\mathrm{i}})$;	
2	$B_{\mathrm{r}}^{k+1} \to R_{05}$;	$R_{01} * R_{02} \to R_{11}$ $(B_{\mathrm{i}} * W_{\mathrm{i}})$;	
3	$A_{\mathrm{i}}^{k} \to R_{06}$;	$R_{04} * R_{03} - R_{10} \to R_{12}$ $(C_{\mathrm{i}} = B_{\mathrm{r}} * W_{\mathrm{r}} - B_{\mathrm{i}} * W_{\mathrm{i}})$;	
4	$A_{\mathrm{i}}^{k+1} \to R_{07}$;	$R_{05} * R_{03} - R_{11} \to R_{13}$ $(C_{\mathrm{i}} = B_{\mathrm{r}} * W_{\mathrm{r}} - B_{\mathrm{i}} * W_{\mathrm{i}})$;	
5	$A_{\mathrm{r}}^{k} \to R_{08}$;	$R_{00} * R_{03} \to R_{10}$ $(B_{\mathrm{i}} * W_{\mathrm{r}})$;	
6	$A_{\mathrm{r}}^{k+1} \to R_{09}$;	$R_{01} * R_{03} \to R_{11}$ $(B_{\mathrm{i}} * W_{\mathrm{r}})$;	
7	$B_{\mathrm{i}}^{k+2} \to R_{00}$;	$R_{04} * R_{02} + R_{10} \to R_{14}$ $(C_{\mathrm{r}} = B_{\mathrm{r}} * W_{\mathrm{i}} + B_{\mathrm{i}} * W_{\mathrm{r}})$;	
8	$B_{\mathrm{i}}^{k+3} \to R_{01}$;	$R_{05} * R_{02} + R_{11} \to R_{15}$ $(C_{\mathrm{r}} = B_{\mathrm{r}} * W_{\mathrm{i}} + B_{\mathrm{i}} * W_{\mathrm{r}})$;	
9	$W_{\mathrm{i}} \to R_{02}$;	$R_{06} + R_{12} \to R_{16}$ $(A_{\mathrm{i}} + C_{\mathrm{i}})$;	
10	$W_{\mathrm{r}} \to R_{03}$;	$R_{07} + R_{13} \to R_{17}$ $(A_{\mathrm{i}} + C_{\mathrm{i}})$;	
11		$R_{06} - R_{12} \to R_{18}$ $(A_{\mathrm{i}} - C_{\mathrm{i}})$;	$R_{16} \to \mathtt{Mem}(A_{\mathrm{i}}^{k})$;
12		$R_{07} - R_{13} \to R_{19}$ $(A_{\mathrm{i}} - C_{\mathrm{i}})$;	$R_{17} \to \mathtt{Mem}(A_{\mathrm{i}}^{k+1})$;
13		$R_{08} + R_{14} \to R_{16}$ $(A_{\mathrm{r}} + C_{\mathrm{r}})$;	$R_{18} \to \mathtt{Mem}(A_{\mathrm{r}}^{k})$;
14		$R_{09} + R_{15} \to R_{17}$ $(A_{\mathrm{r}} + C_{\mathrm{r}})$;	$R_{19} \to \mathtt{Mem}(A_{\mathrm{r}}^{k+1})$;
15		$R_{08} - R_{14} \to R_{18}$ $(A_{\mathrm{r}} - C_{\mathrm{r}})$;	$R_{16} \to \mathtt{Mem}(B_{\mathrm{i}}^{k})$;
16		$R_{09} - R_{15} \to R_{19}$ $(A_{\mathrm{r}} - C_{\mathrm{r}})$;	$R_{17} \to \mathtt{Mem}(B_{\mathrm{i}}^{k+1})$;
17			$R_{18} \to \mathtt{Mem}(B_{\mathrm{r}}^{k})$;
18			$R_{19} \to \mathtt{Mem}(B_{\mathrm{r}}^{k+1})$.

Итого: **18** тактов на две бабочки Фурье;
20 выполненных операций;
2 простейшие операции выполняются за один такт (`a*b+c`).

Эффективность: **55,6 %.**

3.2. Усовершенствование 1

Возможности архитектуры:

- умножение с накоплением двух 32-битных чисел в соседних ps-секциях 64-битного слова ($\mathtt{a}^1\mathtt{*b}^1\mathtt{+c}^1$, $\mathtt{a}^2\mathtt{*b}^2\mathtt{+c}^2$);
- загрузка/выгрузка 64-битного слова (двух 32-битных слов из соседних секций 64-битного слова).

В регистровом файле имеются четыре порта на чтение и три на запись (4/3).

Вычисления организуются следующим образом (обработка элементов с индексами k, $k+1$, $k+2$ и $k+3$, B_{i}^{k}, B_{i}^{k+1}, B_{i}^{k+2}, B_{i}^{k+3}, W_{i}, W_{r} уже загружены на предыдущей итерации (см. такты 7—10)):

1	$B_{\mathrm{r}}^{k},\ B_{\mathrm{r}}^{k+1} \to R_{04};$	$R_{00} * R_{02} \to R_{10}\ \ (B_{\mathrm{i}} * W_{\mathrm{i}});$	(2/1)
2	$B_{\mathrm{r}}^{k+2},\ B_{\mathrm{r}}^{k+3} \to R_{05};$	$R_{01} * R_{02} \to R_{11}\ \ (B_{\mathrm{i}} * W_{\mathrm{i}});$	(2/1)
3	$A_{\mathrm{i}}^{k},\ A_{\mathrm{i}}^{k+1} \to R_{06};$	$R_{04} * R_{03} - R_{10} \to R_{12};$	(3/2)
4	$A_{\mathrm{i}}^{k+2},\ A_{\mathrm{i}}^{k+3} \to R_{07};$	$R_{05} * R_{03} - R_{11} \to R_{13};$	(3/2)
5	$A_{\mathrm{r}}^{k},\ A_{\mathrm{r}}^{k+1} \to R_{08};$	$R_{00} * R_{03} \to R_{10}\ \ (B_{\mathrm{i}} * W_{\mathrm{r}});$	(2/1)
6	$A_{\mathrm{r}}^{k+2},\ A_{\mathrm{r}}^{k+3} \to R_{09};$	$R_{01} * R_{03} \to R_{11}\ \ (B_{\mathrm{i}} * W_{\mathrm{r}});$	(2/1)
7	$B_{\mathrm{i}}^{k+4},\ B_{\mathrm{i}}^{k+5} \to R_{00};$	$R_{04} * R_{02} + R_{10} \to R_{14};$	(3/3)
8	$B_{\mathrm{i}}^{k+6},\ B_{\mathrm{i}}^{k+7} \to R_{01};$	$R_{05} * R_{02} + R_{11} \to R_{15};$	(3/3)
9	$W_{\mathrm{i}},\ W_{\mathrm{i}} \to R_{02};$	$R_{06} + R_{12} \to R_{16}\ \ (A_{\mathrm{i}} + C_{\mathrm{i}});$	(2/1)
10	$W_{\mathrm{r}},\ W_{\mathrm{r}} \to R_{03};$	$R_{07} + R_{13} \to R_{17}\ \ (A_{\mathrm{i}} + C_{\mathrm{i}});$	(2/1)
11	$R_{16} \to \mathtt{Mem}(A_{\mathrm{i}}^{k}, A_{\mathrm{i}}^{k+1});$	$R_{06} - R_{12} \to R_{18}\ \ (A_{\mathrm{i}} - C_{\mathrm{i}});$	(3/2)
12	$R_{17} \to \mathtt{Mem}(A_{\mathrm{i}}^{k+2}, A_{\mathrm{i}}^{k+3});$	$R_{07} - R_{13} \to R_{19}\ \ (A_{\mathrm{i}} - C_{\mathrm{i}});$	(3/2)
13	$R_{18} \to \mathtt{Mem}(A_{\mathrm{r}}^{k}, A_{\mathrm{r}}^{k+1});$	$R_{08} + R_{14} \to R_{16}\ \ (A_{\mathrm{r}} + C_{\mathrm{r}});$	(3/1)
14	$R_{19} \to \mathtt{Mem}(A_{\mathrm{r}}^{k+2}, A_{\mathrm{r}}^{k+3});$	$R_{09} + R_{15} \to R_{17}\ \ (A_{\mathrm{r}} + C_{\mathrm{r}});$	(3/1)
15	$R_{16} \to \mathtt{Mem}(B_{\mathrm{i}}^{k}, B_{\mathrm{i}}^{k+1});$	$R_{08} - R_{14} \to R_{18}\ \ (A_{\mathrm{r}} - C_{\mathrm{r}});$	(3/1)
16	$R_{17} \to \mathtt{Mem}(B_{\mathrm{i}}^{k+2}, B_{\mathrm{i}}^{k+3});$	$R_{09} - R_{15} \to R_{19}\ \ (A_{\mathrm{r}} - C_{\mathrm{r}});$	(3/1)
17	$R_{18} \to \mathtt{Mem}(B_{\mathrm{r}}^{k}, B_{\mathrm{r}}^{k+1});$		(1/1)
18	$R_{19} \to \mathtt{Mem}(B_{\mathrm{r}}^{k+2}, B_{\mathrm{r}}^{k+3});$		(1/1)

Итого: **18** тактов на четыре бабочки Фурье;

40 выполненных операций;

4 простейшие операции выполняются за один такт ($\mathtt{a}^1\mathtt{*b}^1\mathtt{+c}^1$, $\mathtt{a}^2\mathtt{*b}^2\mathtt{+c}^2$).

Эффективность: 55,6 %.

Замечание. На каждом такте происходят пересылки данных. Следовательно, невозможно повысить производительность процессора на данной задаче без усовершенствования операций ввода/вывода. Более того, при осуществлении записи данных из внешней памяти в кэш невозможен обмен данными между внутренними регистрами процессора и L1-кэшем, поэтому необходимо оставить несколько тактов процессора, свободных от работы с L1-кэшем.

3.3. Усовершенствование 2

Возможности архитектуры:

- умножение с накоплением двух 32-битных чисел в соседних ps-секциях 64-битного слова ($\mathtt{a}^1\mathtt{*b}^1\mathtt{+c}^1$, $\mathtt{a}^2\mathtt{*b}^2\mathtt{+c}^2$);
- загрузка/выгрузка двух соседних 64-битных слов (четырёх соседних 32-битных слов из двух соседних 64-битных слов).

В регистровом файле имеются четыре порта на чтение и три на запись (4/3).

Вычисления организуются следующим образом (обработка элементов с индексами k, $k+1$, $k+2$ и $k+3$, в скобках указана используемость портов

ввода/вывода в регистровом файле, все данные для текущей итерации уже загружены (см. такты 9—14)):

1	$R_{20} \to \mathtt{Mem}(B_{\mathrm{i}}^{k-4}, B_{\mathrm{i}}^{k-3})$;	$R_{00} * R_{02} \to R_{10}$ $(B_{\mathrm{i}} * W_{\mathrm{i}})$;	(4/1)
	$R_{21} \to \mathtt{Mem}(B_{\mathrm{i}}^{k-2}, B_{\mathrm{i}}^{k-1})$;		
2	$R_{22} \to \mathtt{Mem}(B_{\mathrm{r}}^{k-4}, B_{\mathrm{r}}^{k-3})$;	$R_{01} * R_{02} \to R_{11}$ $(B_{\mathrm{i}} * W_{\mathrm{i}})$;	(4/1)
	$R_{23} \to \mathtt{Mem}(B_{\mathrm{r}}^{k-2}, B_{\mathrm{r}}^{k-1})$;		
3		$R_{04} * R_{03} - R_{10} \to R_{12}$	(3/1)
4		$R_{05} * R_{03} - R_{11} \to R_{13}$	(3/1)
5		$R_{00} * R_{03} \to R_{10}$ $(B_{\mathrm{i}} * W_{\mathrm{r}})$;	(2/0)
6		$R_{01} * R_{03} \to R_{11}$ $(B_{\mathrm{i}} * W_{\mathrm{r}})$;	(2/0)
7		$R_{04} * R_{02} + R_{10} \to R_{14}$	(3/2)
8		$R_{05} * R_{02} + R_{11} \to R_{15}$	(3/2)
9	$B_{\mathrm{i}}^{k+4},\ B_{\mathrm{i}}^{k+5} \to R_{00}$;	$R_{06} + R_{12} \to R_{16}$ $(A_{\mathrm{i}} + C_{\mathrm{i}})$;	(2/2)
	$B_{\mathrm{i}}^{k+6},\ B_{\mathrm{i}}^{k+7} \to R_{01}$;		
10	$W_{\mathrm{i}},\ W_{\mathrm{i}} \to R_{02}$;	$R_{07} + R_{13} \to R_{17}$ $(A_{\mathrm{i}} + C_{\mathrm{i}})$;	(2/2)
	$W_{\mathrm{r}},\ W_{\mathrm{r}} \to R_{03}$;		
11	$B_{\mathrm{r}}^{k+4}, B_{\mathrm{r}}^{k+5} \to R_{04}$;	$R_{06} - R_{12} \to R_{18}$ $(A_{\mathrm{i}} - C_{\mathrm{i}})$;	(2/3)
12	$B_{\mathrm{r}}^{k+6},\ B_{\mathrm{r}}^{k+7} \to R_{05}$;	$R_{07} - R_{13} \to R_{19}$ $(A_{\mathrm{i}} - C_{\mathrm{i}})$;	(2/3)
13	$A_{\mathrm{r}}^{k+4},\ A_{\mathrm{r}}^{k+5} \to R_{24}$;	$R_{08} + R_{14} \to R_{20}$ $(A_{\mathrm{r}} + C_{\mathrm{r}})$;	(2/3)
	$A_{\mathrm{r}}^{k+6},\ A_{\mathrm{r}}^{k+7} \to R_{25}$;		
14	$A_{\mathrm{i}}^{k+4},\ A_{\mathrm{i}}^{k+5} \to R_{06}$;	$R_{09} + R_{15} \to R_{21}$ $(A_{\mathrm{r}} + C_{\mathrm{r}})$;	(2/3)
	$A_{\mathrm{i}}^{k+6},\ A_{\mathrm{i}}^{k+7} \to R_{07}$;		
15	$R_{16} \to \mathtt{Mem}(A_{\mathrm{i}}^{k}, A_{\mathrm{i}}^{k+1})$;	$R_{08} - R_{14} \to R_{22}$ $(A_{\mathrm{r}} - C_{\mathrm{r}})$;	(4/1)
	$R_{17} \to \mathtt{Mem}(A_{\mathrm{i}}^{k+2}, A_{\mathrm{i}}^{k+3})$;		
16	$R_{18} \to \mathtt{Mem}(A_{\mathrm{r}}^{k}, A_{\mathrm{r}}^{k+1})$;	$R_{09} - R_{15} \to R_{23}$ $(A_{\mathrm{r}} - C_{\mathrm{r}})$;	(4/1)
	$R_{19} \to \mathtt{Mem}(A_{\mathrm{r}}^{k+2}, A_{\mathrm{r}}^{k+3})$		

Итого: **16** тактов на четыре бабочки Фурье;
40 выполненных операций;
4 простейшие операции выполняются за один такт ($\mathtt{a}^1\mathtt{*b}^1\mathtt{+c}^1$, $\mathtt{a}^2\mathtt{*b}^2\mathtt{+c}^2$).

Эффективность: **62,5 %.**

3.4. Усовершенствование 3

Возможности архитектуры:

- четыре умножения с накоплением двух 32-битных чисел в соседних ps-секциях 64-битного слова ($\mathtt{a}^1\mathtt{*b}^1\mathtt{+c}^1$, $\mathtt{a}^1\mathtt{*b}^1\mathtt{-c}^1$, $\mathtt{a}^2\mathtt{*b}^2\mathtt{+c}^2$, $\mathtt{a}^2\mathtt{*b}^2\mathtt{-c}^2$);

- загрузка/выгрузка двух соседних 64-битных слов (четырёх соседних 32-битных слов из двух соседних 64-битных слов). Возможно одновременное выполнение операций $\mathtt{a} \pm \mathtt{b}$ и выгрузка двух соседних 64-битных слов или одновременное выполнение операций $\mathtt{a} \pm \mathtt{b}$ и загрузка одного 64-битного слова.

В регистровом файле имеются четыре порта на чтение и три на запись (4/3).

Вычисления организуются следующим образом (обработка элементов с индексами k, $k+1$, $k+2$ и $k+3$, в скобках указана используемость портов ввода/вывода в регистровом файле, все данные для текущей итерации уже загружены (см. такты 5—10)):

1	$R_{20} \to \mathtt{Mem}(B_{\mathrm{i}}^{k-4}, B_{\mathrm{i}}^{k-3})$; $R_{21} \to \mathtt{Mem}(B_{\mathrm{i}}^{k-2}, B_{\mathrm{i}}^{k-1})$;	$R_{00} * R_{02} \to R_{10}$ $(B_{\mathrm{i}} * W_{\mathrm{i}})$;	(4/2)
2	$R_{22} \to \mathtt{Mem}(B_{\mathrm{r}}^{k-4}, B_{\mathrm{r}}^{k-3})$; $R_{23} \to \mathtt{Mem}(B_{\mathrm{r}}^{k-2}, B_{\mathrm{r}}^{k-1})$;	$R_{01} * R_{02} \to R_{11}$ $(B_{\mathrm{i}} * W_{\mathrm{i}})$;	(4/2)
3	$A_{\mathrm{i}}^{k+4},\ A_{\mathrm{i}}^{k+5} \to R_{24}$; $A_{\mathrm{i}}^{k+6},\ A_{\mathrm{i}}^{k+7} \to R_{25}$;	$R_{04} * R_{03} - R_{10} \to R_{12}$;	(3/3)
4	$A_{\mathrm{r}}^{k+4},\ A_{\mathrm{r}}^{k+5} \to R_{26}$; $A_{\mathrm{r}}^{k+6},\ A_{\mathrm{r}}^{k+7} \to R_{27}$;	$R_{05} * R_{03} - R_{11} \to R_{13}$;	(3/3)
5	$B_{\mathrm{r}}^{k+4}, B_{\mathrm{r}}^{k+5} \to R_{28}$; $B_{\mathrm{r}}^{k+6},\ B_{\mathrm{r}}^{k+7} \to R_{29}$;	$R_{00} * R_{03} \to R_{10}$ $(B_{\mathrm{i}} * W_{\mathrm{r}})$;	(2/2)
6		$R_{01} * R_{03} \to R_{11}$ $(B_{\mathrm{i}} * W_{\mathrm{r}})$;	(2/0)
7		$R_{04} * R_{02} + R_{10} \to R_{14}$	(3/2)
8		$R_{05} * R_{02} + R_{11} \to R_{15}$	(3/2)
9	$W_{\mathrm{i}},\ W_{\mathrm{i}} \to R_{02}$; $W_{\mathrm{r}},\ W_{\mathrm{r}} \to R_{03}$;	$R_{06} \pm R_{12} \to R_{16},\ R_{17}$ $(A_{\mathrm{i}} \pm C_{\mathrm{i}})$;	(2/2)
10	$B_{\mathrm{i}}^{k+4},\ B_{\mathrm{i}}^{k+5} \to R_{00}$; $B_{\mathrm{i}}^{k+6},\ B_{\mathrm{i}}^{k+7} \to R_{01}$;	$R_{07} \pm R_{13} \to R_{18},\ R_{19}$ $(A_{\mathrm{i}} \pm C_{\mathrm{i}})$;	(2/2)
11	$R_{16} \to \mathtt{Mem}(A_{\mathrm{i}}^{k}, A_{\mathrm{i}}^{k+1})$; $R_{17} \to \mathtt{Mem}(A_{\mathrm{i}}^{k+2}, A_{\mathrm{i}}^{k+3})$;	$R_{08} \pm R_{12} \to R_{20},\ R_{21}$ $(A_{\mathrm{r}} \pm C_{\mathrm{r}})$;	(4/3)
12	$R_{18} \to \mathtt{Mem}(A_{\mathrm{r}}^{k}, A_{\mathrm{r}}^{k+1})$; $R_{19} \to \mathtt{Mem}(A_{\mathrm{r}}^{k+2}, A_{\mathrm{r}}^{k+3})$	$R_{09} \pm R_{13} \to R_{22},\ R_{23}$ $(A_{\mathrm{r}} \pm C_{\mathrm{r}})$;	(4/3)

Итого: **12** тактов на четыре бабочки Фурье;
40 выполненных операций;
6 простейших операций выполняются за один такт ($\mathtt{a}^1\mathtt{*b}^1 \pm \mathtt{c}^1$, $\mathtt{a}^2\mathtt{*b}^2 \pm \mathtt{c}^2$).

Эффективность: **62,5 %** (**83,3 %** без учёта повышения пиковой производительности).

3.5. Другие усовершенствования

Помимо усовершенствований 1—3, рассматривался целый ряд других альтернатив, однако в силу различных причин они были отвергнуты. Среди таких альтернатив необходимо упомянуть о команде выполнения комплексного умножения и команде выполнения всей бабочки Фурье. К сожалению, данные усовершенствования либо не приводили к существенному увеличению производительности модуля, либо были технически сложны с точки зрения инженеров.

При добавлении возможности вычисления произведения двух комплексных чисел не происходит существенного повышения производительности, так как операция $\pm$ помещает результаты в соседние регистры. В результате мы не получаем выигрыша в скорости и всё равно вынуждены сортировать данные перед началом обработки. Однако внедрение данного усовершенствования потребовало бы дополнительных вычислительных устройств в составе математического сопроцессора.

Для обхода данной проблемы предлагалось ввести команду, вычисляющую всю бабочку Фурье. При этом возможна ситуация, при которой после такой команды нельзя выполнять никакую команду математического сопроцессора. Однако само это усовершенствование потребовало бы одновременной загрузки трёх и выгрузки двух комплексных чисел, а для эффективного использования вновь введённой команды потребовалось бы усовершенствовать кэш первого уровня (обеспечив возможность одновременной работы кэша с памятью и с процессором). На данном этапе эти усовершенствования сделать технически сложно, поэтому мы не рассматривали эти альтернативы детально. Однако потенциально подобные усовершенствования способны повысить производительность процессора на задачах обработки сигналов более чем в полтора раза.

3.6. Результаты

В табл. 1 кратко суммируются полученные результаты.

Замечание. Реализация всех усовершенствований позволяет повысить производительность процессора на задачах обработки сигналов в три раза.

4. Производительность процессора K 64-CMП на задачах обработки сигналов

До сих пор рассматривалось не быстрое преобразование Фурье в целом, а лишь выполнение одной бабочки Фурье. В данном разделе рассматривается производительность процессора на преобразовании Фурье в предположении, что все данные уже находятся в L1-кэше. L1-кэш процессора K 64-CMП составляет

Таблица 1. Производительность процессора на преобразовании бабочки Фурье при работе с L1-кэшем

	Время вычисления четырёх бабочек Фурье (тактов)	Эффективность[1] (%)	Пиковая производительность[3] (MFlops)
Исходный вариант	36	55,6	800
Усовершенствование 1	18	55,6	1600
Усовершенствование 2	16	62,5	1600
Усовершенствование 3	12	55,6 (83,32[2])	2400 (1600[4])

16 Кб (4 по 4 Кб), т. е. может содержать 2 K (4 по 512) комплексных чисел. Используя данные табл. 1, получаем результаты, представленные в табл. 2.

Замечание. Преобразование Фурье большего размера будет рассмотрено ниже, так как в этом случае данные не помещаются в L1-кэш.

Таблица 2. Производительность процессора на потоке БПФ при работе с L1-кэшем

Размер FFT	Количество проходов	Число тактов процессора на обработку четырёх преобразований Фурье в потоке	Пропускная способность по данным (количество комплексных чисел, обрабатываемых в секунду)
16	4	384	66,7
32	5	960	53,3
64	6	2,25 K	44,4
128	7	5,25 K	38,1
256	8	12 K	33,3
512	9	27 K	29,6

[1]Эффективность вычисляется по формуле $\text{eff} = \frac{40}{xy} \cdot 100\ \%$, где 40 — количество простейших операций, которые необходимо выполнить для обработки четырёх бабочек Фурье, x — количество тактов, затрачиваемых на обработку четырёх бабочек Фурье, y — количество простейших операций, которые могут быть выполнены одновременно ($y = 2$ для строки 1, $y = 4$ для строк 2 и 3, $y = 6$ для строки 4).

[2]В скобках приведена эффективность без учёта того, что при добавлении возможности совмещения операций $\mathtt{a} + \mathtt{b}$ и $\mathtt{a} - \mathtt{b}$ повышается пиковая производительность, т. е. эффективность вычисляется по формуле $\text{eff} = \frac{40}{xy} \cdot 100\ \%$, $x = 12$, $y = 4$.

[3]Здесь и дальнейшем производительность вычисляется в предположении, что процессор работает на частоте 400 МГц.

[4]В скобках приведена пиковая производительность без учёта параллельной обработки команд $\mathtt{a} + \mathtt{b}$ и $\mathtt{a} - \mathtt{b}$.

5. Производительность мезонинного модуля на основе процессора К 64-СМП

5.1. Преобразование Фурье

В этом разделе рассматривается производительность мезонинного модуля на основе микропроцессора К 64-СМП на преобразовании Фурье. Предполагается, что данные попадают на мезонинный модуль по каналу RapidIO, на мезонинном модуле находится память SRAM объёмом 2 Мб и скоростью доступа 64×200 МГц.

На загрузку четырёх слов длины 64 бита из памяти SRAM в L1-кэш необходимо затратить 12 тактов процессора, при этом необходимо предусмотреть 4 такта непосредственно на запись данных в кэш, в течение которых не происходит обмена данными между кэшем и процессором. Это приведёт к увеличению времени обработки бабочки Фурье с 12 тактов до 13 тактов (так как в алгоритме существует лишь 3 такта без обменов данными между кэшем и процессором). В дальнейшем будем предполагать, что на обработку четырёх бабочек Фурье затрачивается 13, а не 12 тактов процессора, хотя это увеличение происходит только в случае параллельной обработки данных и загрузки данных из памяти SRAM.

Для обработки четырёх бабочек Фурье необходимо загрузить восемь комплексных чисел; это можно сделать лишь за 24 такта. Однако для преобразования Фурье необходимо выполнить несколько проходов по вектору. Самое короткое из рассматриваемых преобразований Фурье (длины 16) требует четырёх проходов, поэтому за время обработки можно как загрузить, так и выгрузить все необходимые данные. При увеличении размера преобразования Фурье количество проходов увеличивается, следовательно, увеличивается и время обработки, а затраты на загрузку/выгрузку остаются прежними. При длине вектора меньше 512 можно загружать две пачки преобразований Фурье по четыре вектора в каждой. В этом случае вторая пачка загружается во время обработки первой. При длине вектора, превышающей 256 комплексных чисел, две пачки перестают умещаться в кэш.

Перед началом обработки данные необходимо отсортировать в соответствии со схемой, описанной в разделе 2. На сортировку четырёх комплексных векторов длины N, находящихся в L1-кэше, необходимо затратить $10 \times N$ тактов (одна загрузка по 128 бит и четыре выгрузки по 32 бита на перемещение четырёх действительных чисел). Сортировка данных происходит в L1-кэше непосредственно перед и сразу после окончания обработки. Производительность при коротком преобразовании Фурье представлена в табл. 3.

Теперь рассмотрим случай, если данные не помещаются в L1-кэше размера 16 Кб. В этом случае предлагается перейти к двумерному преобразованию Фурье.

Таблица 3. Производительность мезонинного модуля при работе с памятью: короткое БПФ

Размер FFT	Обработка четырёх FFT в потоке (тактов)	Загрузка/ выгрузка данных для четырёх FFT (тактов)	Сортировка четырёх векторов (тактов)	Пропускная способность по данным без учёта сортировки (МГц)	Пропускная способность по данным с учётом сортировки (МГц)
16	416	384	160	61,5	34,8
32	1040	768	320	49,2	30,5
64	2,44 K	1,5 K	640	41,0	27,0
128	5,69 K	3 K	1,25 K	35,1	24,4
256	13 K	6 K	2,5 K	30,8	22,2

Алгоритм работы.

1. Данные загружаются в L1-кэш.
2. Обработка данных из L1-кэша, промежуточные результаты работы сохраняются в кэше без записи в память. Параллельно происходит загрузка данных для следующего преобразования Фурье.
3. Данные выгружаются из кэша в некэшируемую область памяти в транспонированном порядке. Параллельно происходит обработка следующего преобразования Фурье.

Производительность при длинном преобразовании Фурье представлена в табл. 4.

Таблица 4. Производительность мезонинного модуля при работе с памятью: длинное БПФ

Размер FFT	Обработка четырёх FFT в потоке (тактов)	Сортировка четырёх векторов (тактов)	Пропускная способность по данным без учёта сортировки (МГц)	Пропускная способность по данным с учётом сортировки (МГц)
$512 = 16 \times 32$	29,25 K	5 K	27,4	20,4
$1\text{ K} = 32 \times 32$	65 K	10 K	24,6	18,8
$2\text{ K} = 32 \times 64$	143,1 K	20 K	22,4	17,5
$4\text{ K} = 64 \times 64$	312,4 K	40 K	20,5	16,3
$8\text{ K} = 64 \times 128$	676,5 K	80 K	18,9	14,9
$16\text{ K} = 128 \times 128$	1456,6 K	160 K	17,6	14,4

5.2. Модельная задача

Описание задачи: на вход подаётся поток комплексных отсчётов длины 2 К в формате $32 + 32$ бита. После этого выполняются следующие действия:

- свёртка вектора длины 2 К с весовой функцией (прямое БПФ, умножение на весовую функцию, обратное БПФ);
- накопление матрицы из 64 векторов длины 2 К каждый;
- обработка столбцов получившейся матрицы (длины 64);
- пороговая фильтрация.

Перед началом обработки данных необходимо провести сортировку. Согласно табл. 4 на сортировку 64 векторов длины 2 К уходит $20\ \text{К} \times 16 = 320\ \text{К}$ тактов.

На выполнение четырёх свёрток векторов длины 2 К с весовой функцией необходимо затратить 286,2 К тактов, следовательно, на обработку 64 таких свёрток необходимо затратить $16 \times 286{,}2\ \text{К} = 4579\ \text{К}$ тактов. При этом данные сохраняются в транспонированном порядке.

2 К преобразований Фурье над столбцами матрицы длины 64 выполняется за $512 \times 2{,}44\ \text{К} = 1249\ \text{К}$ тактов.

Пороговая фильтрация занимает 512 К тактов (по четыре операции над каждым числом, из них две выполняются одновременно).

Наконец, обратная сортировка занимает 320 К тактов.

Итого: $320\ \text{К} + 4579\ \text{К} + 1249\ \text{К} + 512\ \text{К} + 320\ \text{К} = 6980\ \text{К}$ тактов (пропускная способность по данным: 7,3 МГц).

Времена обработки различных этапов представлены на рис. 4.

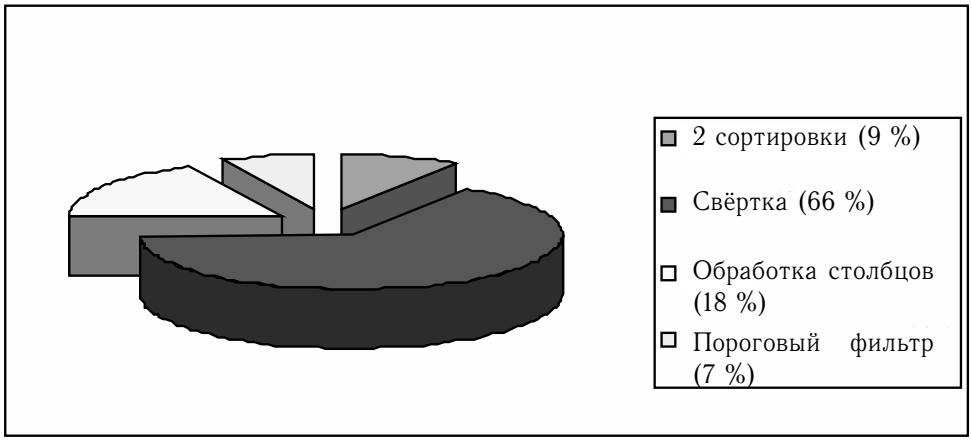

Рис. 4. Времена обработки различных этапов

Авторы работы благодарны А. Г. Кушниренко за многочисленные плодотворные дискуссии в ходе работы над схемой потокового программирования и проектирования расширения процессора К 64-СМП.

Литература

[1] Бахвалов Н. С. Численные методы. — М.: Наука, 1973.

[2] Бетелин В. Б., Бобков С. Г., Зендрикова С. А., Кравченко А. А., Кушниренко А. Г., Николаев В. К. Теоретические оценки эффективности суперЭВМ с распределённой памятью. — М.: НИИСИ РАН, 2003.

[3] Вьюкова В. В., Галатенко В. А., Самборский С. В., Шумаков С. М. Описание VLIW-архитектуры в оптимизирующем постпроцессоре // Информационная безопасность. Инструментальные средства программирования. Базы данных / Под ред. В. Б. Бетелина. — М.: НИИСИ РАН, 2001.

[4] Вьюкова В. В., Галатенко В. А., Самборский С. В., Шумаков С. М. О проблеме оптимизации кода для процессорных архитектур с явным параллелизмом. — М.: ИПУ РАН, 2001.

[5] Корнеев В. В. Параллельные вычислительные системы. — М.: Нолидж, 1999.

[6] Кушниренко А. Г., Лебедев Г. В. Программирование для математиков. — М.: Наука, 1988.

[7] Лесных А. А., Широков И. А. Оценки производительности суперЭВМ на основе сопроцессора вещественной арифметики на задачах обработки сигналов. — М.: НИИСИ РАН, 2005.

[8] Dongarra J. J. Performance of various computers using standard linear equations software. — Technical report UT-CS-89-85. — Knoxville: University of Tennessee, 1989. — http://www.netlib.org/benchmark/performance.ps.

[9] Dongarra J. J., Duff I. S., Sorensen D. C., Van der Vorst H. A. Numerical Linear Algebra for High-Performance Computers. — Philadelphia: SIAM, 1998. — (Software — Environments — Tools; Vol. 7).

[10] Hwang K., Xu Z. Scalable parallel computing. — McGraw-Hill, 1998.