ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 171 за 2018 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

Андреев С.С., Дбар С.А.,
Лацис А.О., Лихачев И.В.,
Плоткина Е.А., Фиалко Н.С.

Типовая структура программы одномерного моделирования динамики цепочки ДНК и ее схемная реализация

Рекомендуемая форма библиографической ссылки: Типовая структура программы одномерного моделирования динамики цепочки ДНК и ее схемная реализация / С.С.Андреев [и др.] // Препринты ИПМ им. М.В.Келдыша. 2018. № 171. 16 с. doi:10.20948/prepr-2018-171
URL: http://library.keldysh.ru/preprint.asp?id=2018-171

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

С.С.Андреев, С.А.Дбар, А.О.Лацис, И.В.Лихачев,
Е.А.Плоткина, Н.С.Фиалко

Типовая структура программы одномерного моделирования динамики цепочки ДНК и ее схемная реализация

Москва — 2018

Андреев С.С., Дбар С.А., Лацис А.О. , Лихачев И.В., Плоткина Е.А., Фиалко Н.С.

Типовая структура программы одномерного моделирования динамики цепочки ДНК и ее схемная реализация

Многие задачи моделирования динамики одномерных молекулярных цепочек ДНК имеют сходную и очень простую структуру циклов и зависимостей данных в расчетных формулах. В сочетании с высокой удельной вычислительной нагрузкой это делает упомянутые задачи идеальными для ускорения счета путем схемной реализации на FPGA. Однако, как показал опыт первых таких реализаций, ошибки при попытках описать эффективные схемы такого рода на языке высокого уровня также имеют очень простой и сходный для разных задач (и разных разработчиков) характер. В нашей работе мы постарались описать приемы эффективного построения таких схем, не привязываясь к конкретной задаче моделирования.

Ключевые слова: моделирование динамики одномерных молекулярных цепочек ДНК, FPGA, схемная реализация вычислений.

Sergey Sergeevich Andreev, Svetlana Alekseevna Dbar, Aleksey Ottovich Lacis, Elena Aronovna Plotkina, Ilya Vasilyevich Likhachev, Nadezhda Sergeevna Fialko

Typical structure of the program for dynamics of one-dimensional DNA molecular chain simulation and its hardware implementation

Many molecular dynamics simulation of one-dimensional models of DNA dynamics simulation of one-dimensional of DNA codes have similar and very simple structure of loops and data dependencies. Together with high computational density, this makes those tasks ideal for acceleration via hardware implementation in FPGA. But, according to the experience of the first of such implementations, the mistakes done by different implementors for the different codes are also very similar and very simple. The ways of making the hardware implementation of such codes efficient, apart from any particular code, are presented in this paper.

Key words: DNA molecular dynamics simulation, FPGA, hardware implementation of computations.

Работа выполнена при поддержке Программы Президиума РАН №26 «Фундаментальные основы создания алгоритмов и программного обеспечения для перспективных сверхвысокопроизводительных вычислений»

1. Ограничения на алгоритм, реализуемый в FPGA. Техника реализации

Основные ограничения, налагаемые спецификой гибридно-реконфигурируемого вычислителя на алгоритм, подробно рассмотрены в [1] и состоят в следующем:

- необходимость локализации вычислений с высокой удельной вычислительной нагрузкой в мелких (около 1 – 2 мегабайт) блоках данных, копируемых из памяти процессора в память ускорителя и обратно;
- необходимость глубокой конвейеризации обработки данных в пределах блока внутри ускорителя.

При невозможности уложиться в указанные ограничения реализация на FPGA формально возможна, но гарантированно приведет к замедлению расчета вместо его ускорения.

Техника реализации приложений для гибридно-реконфигурируемого вычислителя подробно описана в [1]. Суть ее в следующем.

Приложение разбивается на программную и аппаратную части, выполняющиеся, соответственно, на процессоре и на ускорителе. Связь между ними осуществляется с помощью специальной библиотеки передачи данных и запуска ускорителя [2], отдаленно напоминающей аналогичные библиотеки для взаимодействия процессора с GPGPU. Структура приложения для гибридно-реконфигурируемого вычислителя представлена на Рис.1.

Рис. 1. Структура приложения для гибридно-реконфигурируемого вычислителя.

Для того чтобы аккуратно разработать и отладить как программную, так и аппаратную части, изготавливается программная модель аппаратной части и программная модель библиотеки передачи данных. В совокупности эти две программных модели позволяют разработать и отладить программную часть приложения. Программная часть приложения теперь взаимодействует с

моделью аппаратной части посредством вызова функций библиотеки передачи данных, то есть так, как будто бы аппаратная часть уже существовала в действительности.

Теперь необходимо разработать реальную (а не моделируемую процессором) аппаратную часть и «прожечь» ее в FPGA, чтобы получился ускоритель. Для получения же приложения в целом необходимо будет добавить программную часть, разработанную на предыдущем шаге, но собранную с реальной библиотекой передачи данных, а не с ее программной моделью.

Технологий разработки аппаратной части приложения существует несколько [1,2]. Для рассматриваемого нами класса задач пригодна наиболее простая и высокоуровневая – технология синтеза схемы непосредственно по тексту реализуемого ей алгоритма на языке C++ [3]. Это означает, что текст и реальной аппаратной части приложения и ее программной модели – один и тот же, просто для моделирования на процессоре и для реального «прожига» в FPGA этот текст надо транслировать разными трансляторами.

Неверно было бы думать, что разработка аппаратной части приложения на этом этапе в целом закончена. В действительности, она только начинается.

Трансляция в схему аппаратной части в ее первоначальном виде даст, скорее всего, лишь формально работоспособный, но очень медленный вариант «ускорителя», который не ускорит, а, напротив, замедлит выполнение приложения во много раз. Чтобы схема стала быстрой, транслятору необходимо построить ее в форме достаточно глубокого конвейера. Для этого, в свою очередь, программа должна быть написана таким образом, чтобы возможность реализации критического цикла в конвейерной форме была совершенно очевидна. Но для этого, как правило, мало бывает добавить к заголовку цикла директиву «построить конвейер». Часто приходится переписывать сами циклы, например, меняя порядок вложенности. Такого рода преобразования исходного текста иногда практически не влияют на быстродействие получаемой из него **программы**, в то время как быстродействие получаемой из него **схемы** меняется в сотни раз. Текст при этом остается правильным по смыслу текстом на C++, поскольку преобразуется формально эквивалентным образом. Тем самым, из огромного числа возможных вариантов записи алгоритма на C++, практически равноценных при трансляции в программу, мы целенаправленно выбираем те, которые способны хорошо транслироваться также и в схему. В этом и заключается разработка аппаратной части приложения в рамках выбранной технологии. Чтобы убедиться, что внесенные в текст изменения действительно не меняют его смысла, его (текст) придется периодически транслировать и запускать в виде программной модели.

2. Типовая структура приложения моделирования динамики одномерной молекулярной цепочки ДНК

Имеется небольшое (порядка единиц) число входных и столько же выходных одномерных массивов типа **double**. Входные массивы **AI1**, **AI2**, ... имеют длину **N** (порядка тысяч–десятков тысяч). Выходные массивы **AO1**, **AO2**, ... имеют длину **N*NLARGE**, где смысл значения **NLARGE** объясняется ниже.

double AI1[N], AI2[N], ...
double AO1[N*NLARGE], AO2[N*NLARGE], ...

Также имеется небольшое (единицы–десятки) число скалярных входных параметров типа **double**:

double sa1, sa2, ...

и выходной массив усредненных величин типа **double**:

double avers[NLARGE*NAVERS]

где значение **NAVERS** невелико (единицы–десятки), а смысл значения **NLARGE** объясняется ниже.

Моделирование ведется шагами по времени. Всего выполняется **NLARGE** (сотни тысяч) больших шагов, каждый из которых состоит из **NSMALL** (сотни–тысячи) малых шагов и последующего вычисления усредненных величин.

Каждый малый шаг заключается в пересчете значений массивов **AI1**, **AI2**, ... по определенным, не зависящим от номера шага формулам, в которые входят, в частности, входные параметры. Новые значения **I**-х элементов массивов зависят от старых значений их элементов с индексами в диапазоне от **I-IS** до **I+IS** включительно, где значение **IS** – порядка единиц.

Каждый большой шаг завершается вычислением значений усредненных величин по текущим значениям массивов **AI1**, **AI2**, ... Всего усредненных величин в конце каждого большого шага вычисляется **NAVERS** штук. Результатом большого шага являются текущие значения **AI1**, **AI2**, ..., а также массив усредненных величин длиной **NAVERS**. Результаты больших шагов складываются в массивы **AO1**, **AO2**, ... и **avers**, соответственно, подряд, шаг за шагом.

3. Разбиение обрабатываемых данных на мелкие блоки

Для организации рассмотренной выше процедуры обработки данных в ускорителе достаточно хранить в его памяти:

- массивы исходных данных, переписываемые на каждом малом шаге, в одном экземпляре;
- массивы результатов, порождаемые на каждом большом шаге, в стольких экземплярах, на сколько больших шагов обработки за один запуск рассчитан ускоритель (как минимум в одном, но чем больше – тем, вообще говоря, лучше).

Следовательно, **AI1**, **AI2**, ... должны помещаться в памяти FPGA хотя бы два раза, а массив значений типа **double** длиной **NAVERS** – хотя бы один раз. В общем случае, если мы хотим, чтобы ускоритель выполнял за один запуск **LBLOCK** больших шагов, **AI1**, **AI2**, ... должны помещаться в памяти FPGA **LBLOCK+1** раз, а массив значений типа **double** длиной **NAVERS** – **LBLOCK** раз.

Приняв решение именно о такой организации приложения, мы можем реализовать модельный вариант функции запуска обработки, вызываемой в программной части.

Функция имеет, в модельном варианте, два входных (**a_input** и **b_input**) и три выходных (**a_output**, **b_output** и **avers_output**) массива, два скалярных входных аргумента **sa1** и **sa2**. В реальных функциях число массивов, как и число входных параметров, может быть другим. Длина входных массивов равна **n**, длина массива усредненных величин равна **navers**. При вызове функции выполняется **nlarge** больших шагов по **nsmall** малых шагов в каждом.

Для передачи данных внутри функции между программными массивами и ускорителем (или его программной моделью) используется библиотека передачи данных и запуска ускорителя [2].

```
#include <stdio.h>
#include <stdlib.h>
#include "comm.h"
#include "blank.h"
   void swblank( long nsmall, long nlarge, long n, long navers,
// sample argument arrays, put your arrays instead:
                double a_input[], double b_input[],
                double a_output[], double b_output[],
                double avers_output[],
// end put your arrays.
// sample scalar arguments, put your arguments instead:
                double sa1, double sa2
// end put your arguments
            )
{
   int arrbase, aversbase, nlcur, rc;
   unsigned long param[6];
//
   param[0] = nsmall;
   param[2] = n;
   param[3] = navers;
   param[4] = *((unsigned long*)(&sa1));
   param[5] = *((unsigned long*)(&sa2));
   logic_put( 0, 0, a_input, n );
```

```
    logic_put( 1, 0, b_input, n );
    arrbase = 0;
    aversbase = 0;
    while ( nlarge )
     {
      nlcur = nlarge;
      if ( nlcur > LBLOCK ) nlcur = LBLOCK;
      param[1] = nlcur;
      logic_put_block_reg( param, 6 );

      logic_init( 0 );
      logic_wait( 0, &rc );
      logic_get( 2, 0, a_output+arrbase, nlcur*n );
      logic_get( 3, 0, b_output+arrbase, nlcur*n );
      logic_get( 4, 0, avers_output+aversbase, nlcur*navers );
      nlarge -= nlcur;
      arrbase += nlcur*n;
      aversbase += nlcur*navers;
     }
}
```

Как легко видеть из приведенного текста, перед началом работы в ускоритель копируются входные массивы. Затем в цикле выполняется запуск ускорителя, каждый раз – на такое число больших шагов, на какое рассчитан ускоритель, исходя из того, сколько результатов больших шагов помещается в его внутренней памяти. После каждого запуска ускорителя результаты счета копируются в программные выходные массивы. Цикл завершается после выполнения заданного суммарного количества больших шагов.

4. Разработка аппаратной части приложения

4.1. Модельный исходный текст

Пример исходного текста аппаратной части модельного приложения выглядит так:

```
#include <stdio.h>
#include <stdlib.h>
#include "blank.h"
/*--vector_proc_64

    void hwblank(
    double a_input,     : ram0, in,  wblock=1,size= 1000
    double b_input,     : ram1, in,  wblock=1,size= 1000
```

```
    double a_output,     : ram2, out,  wblock=1,size= 10000
    double b_output,     : ram3, out,  wblock=1,size= 10000
    double avers_output, : ram4, out,  wblock=1,size= 100
    long nsmall,         : reg0
    long nlarge,         : reg1
    long n,              : reg2
    long navers,         : reg3
    double sa1,          : reg4
    double sa2           : reg5

)--*/
   void hwblank( long nsmall, long nlarge, long n, long navers,
// sample argument arrays, put your arrays instead:
                double a_input[N], double b_input[N],
                double a_output[N*LBLOCK], double b_output[N*LBLOCK],
                double avers_output[NAVERS*LBLOCK],
// end put your arrays.
// sample scalar arguments, put your arguments instead:
                double sa1, double sa2
// end put your arguments
             )
{
   int ilarge, ismall, i, arrbase, aversbase;
   double av1, av2, am1, a, ap1, bm1, b, bp1;
//
   arrbase = 0;
   aversbase = 0;
// Do nlarge large steps:
   for ( ilarge = 0; ilarge < nlarge; ilarge++ )
    {
// For each large step:
//
// 1. Do nsmall small steps:
     for ( ismall = 0; ismall < nsmall; ismall++ )
      {
       am1 = 0.0; a = a_input[0];
       bm1 = 0.0; b = b_input[0];
       for ( i = 0; i < n; i++ )
        {
         if ( i < (n-1) )
          {
           ap1 = a_input[i+1];
           bp1 = b_input[i+1];
```

```
            }
           else
            {
             ap1 = 0.0;
             bp1 = 0.0;
            }
           a_input[i] = (bm1+bp1)*0.5 + b*sa1;
           b_input[i] = (am1+ap1)*0.5 - a*sa2;

           am1 = a; a = ap1;
           bm1 = b; b = bp1;
          }
         }
    // 2. Copy current input to output, build avers:
        av1 = 0.0;
        av2 = 0.0;
        for ( i = 0; i < n; i++ )
         {
          a_output[arrbase+i] = a_input[i];
          av1 += a_input[i];
          b_output[arrbase+i] = b_input[i];
          av2 += b_input[i];
         }
        av1 /= n;
        av2 /= n;
        avers_output[aversbase] = av1;
        avers_output[aversbase+1] = av2;
        arrbase += n;
        aversbase += navers;
       }
    }
```

Блочный комментарий, начинающийся с "**/*--vector_proc_64**", является фактически директивой для генерации логики связи аппаратной части приложения с каналом передачи данных между процессором и ускорителем. При трансляции текста в схему эта генерация выполняется специальной утилитой [2]. При трансляции в программу (для создания программной модели аппаратной части) этот комментарий не используется.

Работа аппаратной части заключается в выполнении заданного числа больших шагов, каждый из которых состоит из заданного числа малых шагов и расчета усредненных величин. Формулы пересчета элементов массивов на каждом малом шаге, как и формулы расчета усредненных величин, в данном, модельном варианте приложения не имеют никакого физического смысла, в

реальном расчете на их место должны быть подставлены реальные формулы. Сходство модельных формул с реальными состоит в том, какие именно элементы массива старых значений используются для вычисления нового значения номер **I**. В данном случае это элементы с номерами **I-1**, **I** и **I+1**. В реальных формулах, как следует из раздела о типовой структуре приложения, должно быть что-то похожее. Для учета именно такой картины зависимости по индексам в тексте программы введены рабочие переменные **am1**, **a**, **ap1**, **bm1**, **b** и **bp1**, равные на **I**-м витке цикла значениям **a_input[I-1]**, **a_input[I]**, **a_input[I+1]**, **b_input[I-1]**, **b_input[I]** и **b_input[I+1]** соответственно. То, что в тексте программы, вместо прямого обращения к **a_input[I-1]**, **a_input[I]**, **a_input[I+1]**, **b_input[I-1]**, **b_input[I]** и **b_input[I+1]**, используются специальные рабочие переменные, не случайно. Как мы увидим далее, такой прием записи обращения к элементам массивов позволяет значительно облегчить задачу построения конвейера при трансляции текста в схему. Это тот самый случай, когда малосущественное с точки зрения программной реализации изменение формы записи оказывается при схемной реализации критически важным.

4.2. Эффективная схемная реализация

Как подробно объяснялось в [1], единственный вариант действительно эффективной схемной реализации состоит в построении по циклам исходного текста программы конвейеров минимально возможной скважности. Если некоторый цикл удалось реализовать в схеме в виде конвейера со скважностью **N**, то это означает, что время выполнения этого цикла, не считая времени разгона конвейера, составляет **N** тактов рабочей частоты на 1 виток цикла, какие бы сложные вычисления ни выполнялись в теле цикла. Лучше всего, конечно, добиться того, чтобы **N** было равно 1.

Конвейеры проще всего строить по простым циклам с линейной, относительно переменной цикла, индексацией обрабатываемых массивов. Транслятор исходного текста в схему [3] по умолчанию ни для каких циклов конвейеров не строит. Циклы, по которым следует строить конвейеры, должны быть отмечены в исходном тексте соответствующими директивами. Для этого их (циклы) необходимо сначала выбрать. В нашем модельном тексте конвейеризации подлежит цикл прохода по элементам входных массивов, с которого начинается малый шаг. После открывающей фигурной скобки этого цикла:

```
for ( i = 0; i < n; i++ )
{
```

следует написать:

```
#pragma HLS pipeline
```

Однако этого мало. Необходимо решить еще как минимум две проблемы.

Первая проблема заключается в том, что транслятор исходного текста программы в схему [3] обычно старается объединить несколько вложенных циклов в один. В результате может получиться объединенный цикл, тело которого слишком сложно для построения конвейера с минимально возможной скважностью. Чтобы этого избежать, заголовок цикла по малым шагам, объемлющего наш цикл прохода по входным массивам:

```
for ( ismall = 0; ismall < nsmall; ismall++ )
{
```

следует сопроводить директивой:

```
#pragma HLS loop_flatten off
```

Эта директива запрещает транслятору объединять указанный цикл со вложенным. В данном случае она, возможно, лишняя, поскольку циклы, вложенные не тесно, то есть с исполняемыми операторами между заголовками внешнего и внутреннего циклов, транслятор обычно и так не объединяет, но в реальных текстах эту директиву лучше все же написать.

Вторая проблема касается возможности построения по телу цикла обхода массивов не просто конвейера, а конвейера с минимально возможной скважностью. Препятствий для снижения скважности до минимально возможного значения, равного 1, может быть много. Главное из них – нехватка портов памяти (такую диагностику часто можно встретить в логах транслятора). Суть этой проблемы в том, что каждый массив программы представляется в схеме отдельным блоком индексируемой памяти, к которому за один и тот же такт рабочей частоты можно обратиться не более чем с двумя различными значениями индекса. Поэтому конструкции вроде:

```
a_input[i] = (b_input[i-1]+b_input[i+1])*0.5 + b_input[i]*sa1;
```

желательно заменить на:

```
a_input[i] = (bm1+bp1)*0.5 + b*sa1;
```

Одиночные переменные представляются в схеме отдельными регистрами, к которым можно обращаться одновременно. Но тогда встает вопрос о том, как присваивать значения этим одиночным переменным без точно таких же многочисленных обращений к массиву, из которого они берутся. Это легко сделать, если учесть, что текущее значение **b** станет на следующем витке цикла значением **bm1**, а значение **bp1** – значением **b**. Фактически на каждом витке цикла из каждого массива требуется выбрать только одно старое значение, хотя при "наивной" записи формул кажется, что их надо выбрать три. Этот прием сокращения количества обращений к элементам массива при их циклическом обходе следует обязательно применять при расчетах по реальным формулам в реальных приложениях. Он, кроме всего прочего, позволяет экономить также и

память. Без его применения для пересчета значений элементов массива на малом шаге потребовалось бы два массива – входной и выходной.

Третья проблема связана с тем, что реальные расчетные формулы могут быть сложнее в структурном отношении, чем рассматриваемый нами тривиальный модельный случай. Например, некоторые из обрабатываемых массивов могут быть не одномерными, а двумерными, и, как следствие, внутри того цикла, для которого мы решили строить конвейер, могут быть еще вложенные циклы по дополнительной размерности. Наличие этих вложенных циклов существенно усложняет для компилятора задачу построения однотактного конвейера. К счастью, в решении этой проблемы компилятору довольно легко помочь. Если такой вложенный цикл имеет независимые витки, число которых известно при компиляции, можно заставить компилятор его развернуть, применив директиву **#pragma HLS unroll**. Компилятор разворачивает цикл точно так же, как если бы программист скопировал и несколько раз вставил тело цикла, каждый раз вручную меняя значение индекса. Теперь, когда вложенные циклы развернуты, структура алгоритма снова укладывается в рамки нашей модели, а все витки развернутого вложенного цикла можно выполнять одновременно. Однако при их одновременном выполнении вновь возникает уже упоминавшаяся выше проблема нехватки портов памяти при доступе к элементам массивов: ведь доступов этих на каждом такте теперь стало во столько раз больше, сколько витков было в развернутом цикле. Эту проблему также легко решить, если одновременно с развертыванием циклов заставить компилятор разбить двумерные массивы по соответствующей размерности на несколько одномерных при помощи директивы **#pragma HLS array_partition.**

К разбиению указанным входных и выходных массивов следует подходить с осторожностью: слишком мелкое разбиение может сделать схему настолько сложной, что ее окончательный синтез станет невозможным.

Наконец, следует потребовать конвейеризации цикла усреднения и переписи входных массивов в выходные, заголовок которого тоже имеет вид:

```
for ( i = 0; i < n; i++ )
{
```

при помощи уже знакомой нам директивы:

```
#pragma HLS pipeline
```

Если скважность этого конвейера окажется равной не 1, а, скажем, 5 или 6, ничего страшного: этот конвейер запускается только один раз на каждом большом шаге. Конвейеризовать этот цикл, однако, все же нужно: не конвейеризованный совсем, он может заметно снизить итоговое быстродействие.

4.3. Проблема нехватки ресурсов FPGA

При синтезе вычислительного конвейера для каждого знака арифметической операции в составе реализуемых этим конвейером формул строится отдельное вычислительное устройство [1]. При этом затрачивается некоторый объем оборудования в составе FPGA. Для реализации очень сложных формул оборудования может не хватить. Выясняется это не при трансляции исходного текста программы в схему, а позже, при окончательном синтезе схемы. Чтобы все же реализовать необходимый вычислительный конвейер в пределах имеющейся FPGA, его надо уместить в меньший объем оборудования. Это возможно, если искусственно повысить скважность главного конвейера, то есть написать директиву конвейеризации цикла в виде:

#pragma HLS pipeline II=2

В результате транслятор исходного текста программы в схему построит более экономную по объему используемого оборудования (и более медленную) схему вычислительного конвейера со скважностью 2 такта. Иногда может потребоваться увеличить скважность еще больше, задав, например, II=3.

5. Опыт разработки реальных приложений

В предыдущем разделе мы показали, какую форму следует придать типовому алгоритму моделирования ДНК, чтобы аппаратная реализация его вычислительного ядра была эффективной. Не следует полагать, что незначительные отклонения от этих правил вызовут незначительные же потери быстродействия по сравнению с "идеальным" случаем. В лучшем случае потери быстродействия будут в 2-3 раза, что сделает мало осмысленной всю затею с аппаратной реализацией. В худшем – потери составят десятки (если не сотни) раз, и, вместо ускорения по сравнению с программной реализацией, получится многократное замедление. В этом разделе рассмотрим кратко типичные отклонения от приведенной выше формы представления интересующих нас алгоритмов, которые можно встретить в исходных текстах программ, не рассчитанных изначально на аппаратную реализацию.

5.1. Неудачный порядок вложенности циклов

Выше, в подразделе 4.1, мы привели исходный текст аппаратной части приложения. Посмотрим внимательно на структуру записанных в нем циклов, опуская некоторые мелкие подробности:

```
// Выполняем nlarge больших шагов:
   for ( ilarge = 0; ilarge < nlarge; ilarge++ )
    {
// Для каждого большого шага:
   // 1. Выполняем nsmall малых шагов:
```

```
    for ( ismall = 0; ismall < nsmall; ismall++ )
    {
            // Внутри каждого малого шага, в цикле по i:
            for ( i = 0; i < n; i++ )
            {
                    // Выполняем некоторые вычисления

                    // с i-ми элементами всех массивов:

                    // Например, такие:
                    a_input[i] = (bm1+bp1)*0.5 + b*sa1;
                    b_input[i] = (am1+ap1)*0.5 - a*sa2;
            }
    }
// 2. Копируем текущие значения входных массивов

// в выходные, вычисляем усредненные величины:

     for ( i = 0; i < n; i++ )
     {
    .....
      }
     }
}
```

Рассмотрим цикл по **ismall** и вложенный в него цикл по **i**. В этом цикле вычисляются i-е значения двух массивов — в реальных задачах таких массивов, i-е значения которых получаются по разным формулам, гораздо больше. Многие программисты оформляют проход по каждому такому массиву в отдельную функцию. Внутренний цикл по **i** тогда оказывается внутри функции, и таких циклов становится столько, сколько раз всего вызывались эти функции. Например, около десятка. Каждый такой спрятанный в функцию цикл, конечно, будет отмечен директивой «pipeline», и для каждого такого цикла будет построен конвейер. Беда лишь в том, что, для каждого витка внешнего (по **ismall**) цикла таких конвейеров придется «прокрутить» десять штук, в то время как можно было бы обойтись всего одним. В заведомой потере быстродействия в 10 раз в данном случае виноват не компилятор, а нерациональный порядок обхода массивов.

5.2. Лишние обращения к массивам при последовательной выборке элементов

Этот вопрос мы уже затрагивали выше, в подразделе 4.2, объясняя присутствие в тексте рабочих переменных **bm1** и **bp1**. Последовательный проход по массиву с шагом 1 не требует выборки из массива более одного

значения за один шаг. Однако при наивной записи такие обращения в тексте появятся. Естественная реакция компилятора — пожаловаться на нехватку портов памяти, а естественная реакция программиста – «спросить» у какой-нибудь поисковой системы, что бы могла означать такая диагностика. Узнав, что при нехватке портов памяти массивы рекомендуется разбивать на слои (**#pragma HLS partition**), программист начнет громоздить все более причудливые директивы, при достаточно призрачных шансах на успех.

5.3. Проблема датчика случайных чисел

В программах описанного здесь вида достаточно типично использование генератора псевдослучайной последовательности, причем, возможно, не одной. Такие генераторы почти всегда рекурсивны, что не дает конвейеризовать их, если только вычисление очередного элемента последовательности по предыдущему занимает более одного такта. Проблема эта хорошо известна, на тему изготовления качественных генераторов псевдослучайных последовательностей с рекурсивным ядром, выполнимым схемно за 1 такт, имеется обширная литература. В качестве примера можно привести [4]. В настоящей работе эта сторона проблемы не рассматривается, хотя всю ее важность мы в полной мере осознаем.

5.4. Выводы

Рассмотренный нами компилятор [3] принципиально отличается от типичного современного компилятора языка C++ в очень важном отношении.

Компилятор, генерирующий программный код для процессора, действительно «знает» гораздо лучше программиста, как ему строить объектную программу, и вполне эффективно изолирует программиста от конкретных деталей ее построения. О компиляторе [3] и процессе построения им схемы ничего подобного сказать нельзя. Используя обманчиво привычную и понятную форму записи, этот компилятор, тем не менее, требует от разработчика умения мыслить как схемотехник, понимания, как должна быть устроена эффективно работающая схема.

Выработка основ такого понимания для частного случая достаточно узкого класса приложений, тем не менее, вполне возможна, что мы и попытались показать в нашей работе.

6. Благодарности

Авторы выражают благодарность Шигаеву Алексею Сергеевичу за идею схожей задачи, реализуемой на FPGA, и помощь в обсуждении результатов.

Литература

1. Андреев С.С., Дбар С.А., Лацис А.О., Плоткина Е.А. Как и почему могут быть использованы на практике суперкомпьютеры на базе FPGA. М., РАН, 2017. ISBN 978-5-906906-61-8

2. Андреев С.С., Дбар С.А., Лацис А.О., Плоткина Е.А. Гибридный реконфигурируемый вычислитель. Руководство программиста. URL: http://www.kiam.ru/MVS/research/fpga/progman/ Дата обращения 04.03.2018.

3. Vivado Design Suite User Guide. High-Level Synthesis. Ug902 (v2014.1) May 30, 2014.
URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf Дата обращения 10.10.2017г.

4. https://ru.wikipedia.org/wiki/Регистр_сдвига_с_линейной_обратной_связью Дата обращения 29.07.2017.

Оглавление