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

Препринты ИПМ • Препринт № 34 за 2021 г.

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

С.С. Андреев, С.А. Дбар,
А.О. Лацис, Е.А. Плоткина

О применении технологий высокоуровневого синтеза к схемной реализации вычислений

Рекомендуемая форма библиографической ссылки: О применении технологий высокоуровневого синтеза к схемной реализации вычислений / С.С. Андреев [и др.] // Препринты ИПМ им. М.В.Келдыша. 2021. № 34. 19 с. https://doi.org/10.20948/prepr-2021-34
https://library.keldysh.ru/preprint.asp?id=2021-34

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, Е.А.Плоткина

О применении технологий высокоуровневого синтеза к схемной реализации вычислений

Москва — 2021

Андреев С.С., Дбар С.А., Лацис А.О., Плоткина Е.А.
О применении технологий высокоуровневого синтеза к схемной реализации вычислений

Аннотация

Схемная реализация вычислений в FPGA – один из путей преодоления современного архитектурного кризиса в суперкомпьютерной отрасли. Путь этот заслуженно считается одним из самых многообещающих, но при этом и одним из самых сложных. Разработка приложения для гибридного реконфигурируемого вычислителя на FPGA подразумевает решение целого ряда задач, каждая из которых сложна и трудоемка сама по себе. "Почетное первое место" по сложности и трудоемкости среди этих частных задач занимает задача логического проектирования вычислительной схемы, реализующей аппаратно ускоряемую часть приложения. В настоящей работе мы исследуем именно эту частную задачу.

Ключевые слова: технологии высокоуровневого синтеза, FPGA, схемная реализация вычислений.

Sergey Sergeevich Andreev, Svetlana Alekseevna Dbar, Aleksey Ottovich Lacis, Elena Aronovna Plotkina
On the Application of High-level Synthesis Technologies to the Circuit Implementation of Calculations

Annotation

Circuit implementation of FPGA computing is one of the ways to overcome the current architectural crisis in the supercomputing industry. This path is deservedly considered one of the most promising, but at the same time one of the most difficult. Developing an application for a hybrid reconfigurable computer on an FPGA involves solving a number of tasks, each of which is complex and time-consuming in itself. The" honorable first place " in terms of complexity and labor intensity among these particular tasks is occupied by the problem of logical design of a computational scheme that implements the hardware-accelerated part of the application. In this paper, we investigate this particular problem.

Key words: high-level synthesis technologies, FPGA, hardware implementation of computations.

Введение

Схемная реализация вычислений в FPGA - один из путей преодоления современного архитектурного кризиса в суперкомпьютерной отрасли [1]. Путь этот заслуженно считается одним из самых многообещающих, но при этом и одним из самых сложных.

Разработка приложения для гибридного реконфигурируемого вычислителя на FPGA подразумевает решение целого ряда задач, каждая из которых сложна и трудоемка сама по себе [1]. "Почетное первое место" по сложности и трудоемкости среди этих частных задач занимает задача логического проектирования вычислительной схемы, реализующей аппаратно ускоряемую часть приложения. В настоящей работе мы исследуем именно эту частную задачу.

Современная технология разработки цифровых схем вообще, а не только схем вычислительного характера, подразумевает описание их логики на некотором формальном языке, называемом ***языком описания оборудования*** (**H**ardware **D**escription **L**anguage, **HDL**). HDL внешне напоминает язык программирования, и решает, в принципе, ту же задачу - позволяет формально описать некоторую вычислительную процедуру с целью дальнейшей трансляции этого описания в машинный формат. Отличие HDL от языка программирования состоит в том, что результатом трансляции является не машинный код, а схема, в нашем случае имеющая вид прошивки для конфигурирования FPGA.

Многолетняя практика показывает, что такие HDL, как VHDL и Verilog, успешно используются профессиональными схемотехниками, но практически непостижимы для математиков, обычно разрабатывающих вычислительные приложения. В [1,2] мы довольно подробно рассматривали причину такого положения дел. Так или иначе, непригодность этих языков для разработки схем с той же легкостью, с какой обычно разрабатываются программы, является экспериментально установленным фактом. В этой связи принято говорить о "недостаточно высоком уровне" упомянутых HDL. Именно отсутствие "достаточно высокоуровневых" HDL принято считать главной (а то и единственной) причиной сложности разработки схем вычислительного характера.

В качестве альтернативы предлагаются языки описания оборудования более высокого уровня, получившие название ***языков HLS*** (**H**igh-**L**evel **S**ynthesis, высокоуровневый синтез: ***синтезом*** называют трансляцию текста на HDL в графовое представление схемы). Особенно привлекательно выглядит в этой роли язык C++, расширенный набором специальных прагм. Выпущенный несколько лет назад компилятор из C++ с прагмами в VHDL от фирмы Xilinx - Vivado HLS [3] - пользуется все возрастающей популярностью среди схемотехников как более удобная, высокоуровневая альтернатива VHDL. Казалось бы, проблема HDL, подходящего для математиков - разработчиков вычислительных приложений, с появлением такого компилятора решена раз и

навсегда. Теперь сложность разработки схем вычислительного характера должна примерно сравняться со сложностью разработки программ - ведь и описания схем, и программы записываются теперь на одном и том же языке. Исследованию того, насколько этим надеждам суждено сбыться, и посвящена настоящая работа.

1. Фундаментальное свойство вычислителей нетрадиционной архитектуры

Во всех без исключения работах, посвященных квантовой информатике, последняя противопоставляется информатике классической. Понятно, что классический программист — это кто-то, твердо владеющий "основами", но, возможно, не очень хорошо знающий "новшества".

Абсолютно **все** применяемые сегодня на практике технологии программирования, от таких древних и фундаментальных, как Фортран-4, до таких изощренных и временами трудно изучаемых, как функциональные языки, или языки программирования нетрадиционных суперкомпьютерных архитектур, **относятся, без сомнения, к области классической информатики**. Все эти технологии имеют в своей основе следующие тезисы:

- детерминированная арифметическая и логическая обработка чисел, где под числом понимается двоичное целое конечной разрядности, возможно, интерпретируемое как вещественное с плавающей точкой;
- хранение обрабатываемых данных в некоторой адресуемой памяти.

Все различия классических, как мы теперь знаем, архитектур между собой касаются исключительно способа записи процедур обработки числовых данных, допускающие или подразумевающие те или иные формы параллельного доступа к данным и выполнения вычислений. Сама обработка, сам способ получения одних чисел из других, вообще говоря, не зависит от того, выполняется ли императивная программа или функциональная, работает ли традиционный процессор, GPGPU или спецвычислитель на ПЛИС. Если мы по окончании расчета мысленно "прокрутим назад" вычислительную процедуру, мы увидим, что выходные данные получены из входных, грубо говоря, по одним и тем же формулам, какой бы вид классического компьютера ни использовался.

Цель использования вычислителей нетрадиционной архитектуры состоит в радикальном повышении эффективности вычислений, по сравнению с использованием процессоров общего назначения, в некоторой разумной метрике. Вопросы выбора подходящей метрики подробно рассматривались в [1]. Здесь и далее мы будем для краткости говорить просто об "ускорении" вычислений по сравнению с "обычным" процессором.

Фундаментальное свойство всех известных сегодня вычислителей нетрадиционной архитектуры, состоит в **отсутствии автоматизма ускорения** при переносе на них приложений с процессора общего назначения.

Рассмотрим задачу ускорения программно реализованного приложения путем переноса его на более быстрый процессор общего назначения. Некоторая работа с исходным текстом для наилучшей адаптации его к новому процессору может потребоваться, но, скорее всего, не обязательна. Основной объем ускорения будет получен **автоматически**, просто за счет самого факта использования более быстрого процессора. Ускорится при этом более или менее любая программа, каким бы способом она ни была написана изначально. Единственным условием ускорения, таким образом, является наличие на более быстром процессоре компилятора того языка программирования, на котором написана программа.

Теперь рассмотрим задачу ускорения программно реализованного приложения путем переноса его части на вычислитель нетрадиционной архитектуры. Пусть при этом подлежащая переносу часть приложения выделена правильно, а компилятор языка, на котором написана программа, на нетрадиционном вычислителе существует. Казалось бы, мы вправе ожидать хоть сколько-нибудь заметного **ускорения** более или менее любой программы - ведь мы переносим ее на **более быстрый** вычислитель? Вопрос риторический. Ясно, что вместо ускорения мы получим замедление, иногда - в десятки раз.

Причина в том, что язык, на котором написана программа (пусть для определенности это будет C++) ориентирован именно на процессор общего назначения (это так, сколько бы мы ни говорили о том, что текст на C++ - "просто программа, ни на что не ориентированная"). Благодаря этой неявной, но очень сильной, глубинной связи языка и процессора, практически любой из многих возможных способов записи алгоритма в виде программы оказывается пригодным для эффективной реализации на процессоре. Программист и процессор в буквальном смысле говорят на одном языке, и потому понимают друг друга с полуслова. Иное дело - вычислитель нетрадиционной архитектуры. Он по определению программируется в иных, нетрадиционных, терминах. Правильно было бы "объясняться" с ним на специально созданных для этой архитектуры нетрадиционных языках, но разработчик вычислительных приложений почти никогда такими языками не владеет. Пытаясь же "разговаривать" с таким вычислителем на C++, программист уподобляется человеку, разъясняющему смысл лирического стихотворения или "соль" анекдота собеседнику, который владеет языком этого стихотворения (анекдота) на уровне "Маша мыла раму". Задача эта не безнадежна, но объяснять придется, тщательно подбирая слова. Далеко не всякое объяснение, в принципе правильное, будет понято таким собеседником. Возвращаясь от метафоры разъяснения "соли" анекдота к нашей программе, мы вынуждены будем отметить, что:

- среди многих возможных способов записи алгоритма в виде программы для вычислителей нетрадиционных архитектур можно выделить способы "хорошие" и "плохие". Разница в быстродействии между ними может достигать нескольких порядков, в зависимости от степени нетрадиционности архитектуры

вычислителя;

- вероятность того, что программист сам по себе, случайно, выберет при первоначальном написании программы именно "хороший" для данной архитектуры способ, на практике равна нулю.

Именно эта ситуация и называется отсутствием автоматизма ускорения. Программа не ускорится, пока мы не перепишем ее, пусть формально и на том же языке, но в совершенно другом стиле, причем сознательно "подгоняя" стиль под конкретный вид нетрадиционной архитектуры. Чем более нетрадиционна архитектура, тем более вычурным и противоестественным будет правильный именно для нее стиль программирования на традиционном языке. Правильные программные реализации одного и того же алгоритма на одном и том же языке для GPGPU, в одном случае, и для FPGA, в другом, могут иметь друг с другом очень мало общего. При этом обе они будут являться правильными программами на C++, делающими одно и то же, и имеющими примерно одинаковое быстродействие на процессоре общего назначения.

Отсутствие автоматизма ускорения крайне печально для тех прикладных программистов, которые не могут или не хотят заниматься кропотливым переписыванием своих программ, пусть и с целью их многократного ускорения. Перед теми же немногими, кто готов пытаться, встает сложная и увлекательная задача конкретного исследования вопроса о том, как именно, и до какой степени, можно использовать традиционный язык программирования фон-неймановского типа для программирования нетрадиционных архитектур. В нашем случае речь должна идти об использовании C++ с прагмами для описания логики эффективно реализуемых в FPGA цифровых схем вычислительного характера.

2. Какие схемы мы хотим строить, и что этому мешает

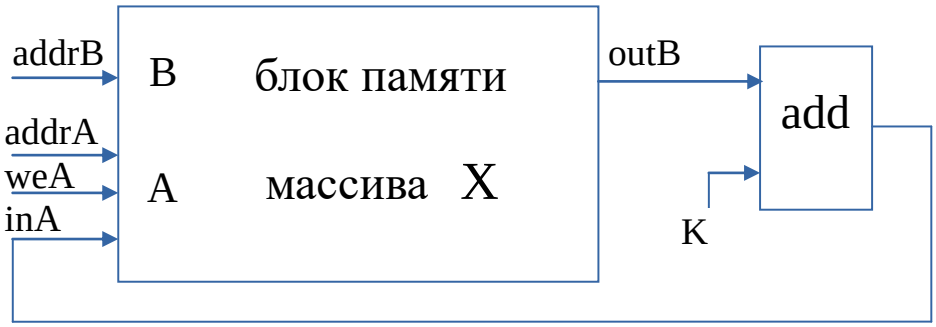
В [1,2] мы подробно останавливались на том, какого рода схемы вычислительного характера имеет смысл строить с целью ускорения приложений.

Жертвовать эффективностью ради удобства программиста - обычное дело в традиционном программировании. Будь у нас такая возможность, мы просто приняли бы, как должное, то качество цифровой схемы, которое получается при компиляции программы, написанной по умолчанию. К сожалению или к счастью, в области высокопроизводительных вычислений так поступать нельзя. Если мы пожертвуем в угоду удобству программиста так много производительности, что быстродействие нашей схемы не превысит быстродействия обычной программы, то вся затея заведомо потеряет смысл. Учитывая же, что схемы, реализуемые в FPGA, имеют на порядок меньшую рабочую частоту, чем "настоящие" процессоры, приходим к выводу, что нам годятся не всякие приемы аппаратного ускорения вычислений, а только самые эффективные из них.

Прежде чем перейти к конкретным примерам, хотелось бы пояснить, что и почему является источником ускорения эффективных схем, реализуемых в FPGA.

Специфика аппаратуры FPGA позволяет объединять цикл чтения данных, вычисления над ними и запись результатов в синхронный конвейер, который управляется тактовым генератором. Время работы конвейера измеряется в тактах. На рисунке ниже показан пример самого простого конвейера, управляющего потоком вычислений, реализующий следующий цикл:

```
for ( j=0; j<N; j++ ) X[j] = X[j] + K;
```



Данный конвейер состоит из двух элементов: блока памяти массива X и компонента сложения вещественных чисел **add**, а также регистров доступа данных к этим элементам. Память 2-х портовая с двумя независимыми портами A и B. Для каждого массива в схеме создается свой блок памяти. Каждый элемент конвейера и регистры доступа воспринимают входные данные по переднему фронту импульса тактового генератора.

Работа конвейера начинается с подачи нулевого адреса на шину addrB. На выходе данные outB появляются через такт после подачи адреса. Т.е. время выборки числа из памяти — 2 такта. Время работы оператора сложения add, допустим, 5 тактов. Время от выборки числа из памяти до записи результата в память называется латентностью конвейера (в данном случае это 7 тактов). Запись в память, соответственно, должна открыться через 7 тактов подачей нулевого адреса на шину addrA и установкой в 1 признака записи weA.

В данном цикле адреса чтения на шине addrB можно менять на каждом такте. Естественно, с задержкой в 7 тактов необходимо менять и адреса записи на addrA. Время, т. е. число тактов, между двумя последовательными выборками данных называется скважностью конвейера (или пропускная способность конвейера). В данном случае скважность = 1 и конвейер называется однотактным.

Время выполнения всего цикла равно: N*ch + lat, где

N – длина массива;

ch – скважность;

lat – латентность.

Если длина массива на порядки больше латентности, а скважность равна 1, то время выполнения будет равно N тактов, независимо от количества вычислений. А если для данных использовать векторную память шириной V, то время выполнения будет равно N/V тактов. Для векторной памяти будет построено V синхронных конвейеров, которые могут работать одновременно.

Таким образом, как показывает практика, эффективной схемой является прием реализации программных циклов в виде длинных синхронных конвейеров минимально возможной скважности и максимально возможной ширины [1,2,3]. Рассмотрим на примере Vivado HLS [3] и некоторых перспективных для реализации в FPGA приложений, что именно и как может помешать компилятору "увидеть" в программном цикле однотактный конвейер и успешно реализовать его. Будем при этом различать две ситуации:

- написать требуемый алгоритм на C++ таким образом, чтобы компилятор построил по этой записи конвейер, в принципе можно, хотя сделать это вручную, возможно, и нелегко;
- записать то, что надо, на C++ невозможно, реализация конвейера возможна только при записи вычислительной процедуры на более подходящем языке.

2.1. Пример 1. Суммирование массива вещественных чисел

Рассмотрим следующий фрагмент программы:

```
double s, *a;
....
s = 0.0;
for ( int i = 0; i < N; i++ ) s += a[i];
```

То, что в цикле выполняется именно суммирование, в данном случае неважно: это могло бы быть, например, вычисление скалярного произведения, или поиск максимального значения в массиве. Проблему при построении конвейера создает то, что значение **s**, полученное на текущем витке цикла, используется уже на следующем витке, без задержки. Это в корне отличается, например, от такой ситуации:

```
for ( int i = 0; i < N; i++ ) a[i] = exp( cos( b[i]/c ) );
```

в которой на каждом витке цикла выполняется гораздо больше вычислений, но их результат не используется уже на следующем витке, что и позволяет легко реализовать однотактный конвейер.

В случае суммирования, на первый взгляд, минимально возможная скважность конвейера равна латентности операции сложения вещественных чисел, что составляет обычно 4 - 5 тактов. В действительности суммирование со скоростью один элемент массива за такт реализовать можно и при такой

латентности единичного сумматора.

Пусть, для определенности, латентность сложения вещественных чисел равна 5 тактам. Тогда для реализации однотактного в среднем накопительного суммирования достаточно использовать 5 элементарных накопительных сумматоров. Очередной элемент массива подается каждый такт на сумматор, номер которого выбирается циклически, с периодом 5:

a[0] подается на сумматор **№0**

a[1] подается на сумматор **№1**

a[2] подается на сумматор **№2**

a[3] подается на сумматор **№3**

a[4] подается на сумматор **№4**

a[5] подается на сумматор **№0,**

a[6] подается на сумматор **№1,**

....

Таким образом, между последовательными актами подачи очередного элемента массива на элементарный накопительный сумматор с конкретным номером проходит как раз 5 тактов. В конце цикла образуются 5 частичных сумм, которые необходимо сложить между собой. Это можно сделать, например, на пирамиде сумматоров.

Компилятор Vivado HLS сам такую схему не построит, и это легко понять: ведь предложенный прием меняет порядок суммирования элементов массива, то есть построенная указанным способом схема была бы не эквивалентна исходной программе на C++. Что же делать разработчику схемы, который согласен на изменение порядка суммирования массива, но хотел бы выполнить его в 5 раз быстрее, чем предлагает компилятор?

В форумах пользователей Xilinx можно найти рекомендации на этот счет. Они сводятся к тому, чтобы записать на C++ в явном виде описанный выше "фокус" с циклическим использованием пяти сумматоров для накопления частичных сумм. Это требует, в свою очередь, написать вместо одного цикла по **i** два вложенных цикла, примерно так:

```
double s[5];
.....
for ( int i = 0; i < N; i += 5 )
 {
#pragma HLS pipeline
    for ( j = 0; j < 5; j++ )
     {
```

```
#pragma HLS unroll
            s[j] += a[i+j];
            .....
```

Получившийся в итоге довольно вычурный текст, богато сдобренный прагмами, обладает следующими недостатками:

- в нем в явном виде присутствует константа 5 - латентность суммирования вещественных чисел, которую авторы приводимых на форумах рекомендаций, фактически, "берут с потолка", "подсматривают" задним числом в протоколе компиляции, и которая впоследствии имеет право измениться;
- он становится запретительно сложным, если рассматривать не "игрушечный" фрагмент программы с суммированием массива в чистом виде, а реальный код, в котором присутствует некоторая сложная логика, и на ее фоне, в дополнение к ней - суммирование;
- он зачастую не работает, поскольку предлагаемая форма записи все равно является для компилятора не прямым указанием сделать нечто, а чем-то вроде "намека", который в зависимости от версии компилятора и от контекста может быть либо понят компилятором, либо нет.

Подводя итог, мы вынуждены отнести рассмотренный пример в категорию "записать то, что надо, на C++ невозможно".

2.2. Пример 2. Развертывание циклов с переменными границами

Одним из перспективных для ускорения на FPGA классов приложений являются алгоритмы обработки изображений, в частности - вычисление двумерных сверток [1]. Однократное применение ядра свертки записывается на C++ как два вложенных цикла, каждый из которых выполняется небольшое (обычно 3 - 7) число раз. Для эффективной реализации эти циклы должны быть полностью развернуты, но это возможно только при фиксированном, известном во время компиляции, числе витков цикла. Если же хочется изготовить схему, в которой число витков цикла (размер ядра свертки) задается значением переменной, приходится писать что-то вроде:

```
    for ( int i = 0; i < MMAX; i++ ) {
#pragma HLS unroll
        for ( j = 0; j < NMAX; j++ )  {
#pragma HLS unroll
            if ( (i < m) && (j<n)) {
                ......
```

Текст, несомненно, много выиграл бы в читабельности, если бы можно было указать в качестве границ циклов **m** и **n**, а не **MMAX** и **NMAX**, а значения

MMAX и **NMAX** задать в соответствующей прагме. Такая прагма в компиляторе Vivado HLS даже предусмотрена, но она игнорируется при синтезе, а служит только для получения более красивого представления будущей схемы при редактировании исходного текста в IDE [3]!

Этот пример, безусловно, относится к категории "написать требуемый алгоритм на C++ таким образом, чтобы компилятор построил по этой записи конвейер, в принципе можно".

2.3. Пример 3. Ложно-многочисленные обращения к элементам массива

Этот пример мы кратко рассматривали в [4]. Рассмотрим такой фрагмент программы:

```
for ( int i = 1; i < (N-1); i++ ) b[i] = (a[i-1]+a[i]+a[i+1])*0.3;
```

Построение конвейера в данном случае тривиально. Однако, если мы захотим сделать его однотактным, нам придется на каждом такте трижды обращаться к различным элементам массива **a**, что невозможно, поскольку память FPGA, используемая для реализации массивов, обычно имеет всего 2 порта доступа. Компилятор вынужден будет увеличить скважность конвейера, и выдаст в логе соответствующую диагностику. В руководствах Xilinx рекомендуется в таких случаях прибегать к циклическому расслоению массива, задавая его при помощи соответствующих прагм при объявлении. Рекомендация, мягко говоря, не безупречная: расслоение массива заметно усложняет схему, прибегать к нему в данном примитивном случае - непозволительная роскошь. Куда проще использовать тот факт, что массив, к трем разным элементам которого приходится обращаться на каждом такте, проходится последовательно:

```
ap = a[0];
ac = a[1];
for ( int i = 1; i < (N-1); i++ )
 {
  b[i] = (ap+ac+a[i+1])*0.3;
  ap = ac; ac = a[i+1];
 }
```

Действительно необходимое число обращений к разным элементам массива **a** на каждом такте, как видим, равно одному, а не трем. Честно говоря, совершенно непонятно, почему компилятор не делает сам настолько простое оптимизирующее преобразование, но - не делает.

В данном случае дефицит портов доступа к памяти оказался ложным, но во многих других случаях он оказывается истинным. Расслоение массива бывает

еще необходимо, когда результат записывается в тот же массив, откуда идет чтение, чтобы обезопасить старые данные для текущей итерации.

Кроме расслоения массива, которое, как мы отмечали выше, ведет к заметному усложнению схемы, для получения дополнительных портов доступа на чтение массивы иногда размножают. Это используется в циклах с вычислениями многоточечных шаблонов для двухмерных или трехмерных массивов. В этих случаях на каждом такте необходимо иметь 9, 11 и более элементов одного массива. В нашем случае это выглядело бы примерно так:

- наряду с массивом **a** объявить массив **a1** того же типа и размера, что и **a**;
- все присваивания вида "**a[i] =**" заменить на "**a1[i]=a[i]=**";
- в правой части операторов присваивания вместо примерно половины обращений к **a[i]** использовать **a1[i]**.

Поскольку такое оптимизирующее преобразование сопряжено с удвоением объема необходимой памяти, ждать от компилятора, чтобы он применил это преобразование автоматически, было бы неразумно. Проблема в том, что и по запросу он его применять не умеет - просто нет такого запроса.

Эти два примера, безусловно, также относятся к категории "написать требуемый алгоритм на C++ таким образом, чтобы компилятор построил по этой записи конвейер, в принципе можно".

2.4. Пример 4. Ложная зависимость от еще не выполненных вычислений при объединении нескольких циклов в один

Рассмотрим такой фрагмент программы:

```
void proc( double a[1000], double b[1000], double c[1000], double d[1000] )
{
   int i;
   {
#pragma HLS loop_merge
   for ( i = 0; i < 1000; i++ )
    {
#pragma HLS pipeline
     c[i] = a[i] + 0.5*(((i==0)?0.0:b[i-1])+((i==999)?0.0:b[i+1]));
    }
   for ( i = 0; i < 1000; i++ )
    {
#pragma HLS pipeline
     d[i] = a[i] + 0.5*(((i==0)?0.0:c[i-1])+((i==999)?0.0:c[i+1]));
    }
   for ( i = 0; i < 1000; i++ )
    {
#pragma HLS pipeline
     a[i] = b[i] + 0.5*(((i==0)?0.0:d[i-1])+((i==999)?0.0:d[i+1]));
```

```
    }
   }
}
```

Каждый из трех циклов по **i** тривиально конвейеризуется. Время работы каждого из этих конвейеров, с точностью до "краевых эффектов", составляет 1000 тактов. Время это будет потрачено трижды: сначала отработает первый цикл, затем - второй, после него - третий. Объединив эти три цикла в один, мы могли бы построить единый конвейер, и потратить время, равное 1000 тактов, не три раза, а всего один раз [4]. При этом мы столкнемся с чисто технической, но очень неприятной трудностью. Рассмотрим второй из циклов, в котором вычисляются новые значения элементов массива **d**. Видим, что **d[i]** зависит от **c[i+1]**. Если вычислять значения элементов **c** предварительно отдельным циклом, как это написано выше, то никакой проблемы в этом нет: ведь к моменту выполнения второго цикла первый уже отработал, и все необходимые нам значения элементов **c** уже вычислены. Если же мы попытаемся объединить первый и второй циклы, то при вычислении **d[i]** у нас возникнет потребность в значении **c[i+1]**, которое появится только на следующем витке цикла по **i**. Проблема тривиально решается "сдвигом фазы": на **i**-м витке следует вычислять **c[i]** и **d[i-1]**. При этом не будем забывать о необходимости, вообще говоря, использовать еще и прием экономии числа доступов к элементам массива, описанный выше, в предыдущем подразделе. А также о том, что третий цикл также содержит "сдвиг фазы" относительно второго. Ну и, конечно же, о том, что в реальных задачах не только число сдвигов фазы, но и объем формул, в которые все эти сдвиги необходимо аккуратно и безошибочно подставить, гораздо больше, чем в приведенном здесь "игрушечном" примере. Принимая во внимание все это вместе, приходим к выводу, что совсем простые в принципе преобразования текста программы на практике вряд ли удастся выполнить без ошибок, не говоря уже о полной потере читабельности и годности к сопровождению получившегося текста.

В компиляторе Vivado HLS имеется прагма, предписывающая объединение нескольких последовательных циклов в один, и в данном примере она была использована с экспериментальными целями (**#pragma HLS loop_merge**). Конечно же, объединять наши три цикла в один компилятор отказался, сославшись на наличие "зависимостей" между ними.

Здесь мы явно имеем дело со случаем "написать требуемый алгоритм на C++ таким образом, чтобы компилятор построил по этой записи конвейер, в принципе можно, хотя сделать это вручную, возможно, и нелегко".

2.5. Предварительные выводы

Критика в адрес разработчиков компилятора Vivado HLS - самое последнее, ради чего были подробно рассмотрены примеры, приведенные в этом разделе. Если бы проблема сводилась очевидным образом к конечному числу недоделок в сравнительно новом компиляторе, мы уже давно жили бы в

прекрасном мире вычислительной схемотехники и высокоуровневого синтеза. Как показывает опыт, список прагм и оптимизаций, которых "совершенно категорически не хватает" компилятору традиционного языка для нетрадиционной архитектуры, обычно оказывается бесконечным. Другие разработчики, опираясь на опыт попыток реализации в FPGA других численных методов, указали бы на критическую необходимость совершенно других, столь же "простых и очевидных", прагм и оптимизирующих преобразований. Урок же, который мы действительно хотели бы извлечь из наших рассмотрений, лежит в несколько другой плоскости.

Полезность транслятора C++ в схему как инструмента действительно быстрой разработки общего логического каркаса схемы сомнений не вызывает.

Тот факт, что возможностей C++, даже в сочетании с богатым набором прагм, недостаточно для выражения многих приемов эффективного построения схем, также вряд ли вызывает сомнение.

Исходя из этого, необходимо понять, как именно следует расширить и дополнить возможности системы HLS. Рассмотрев приведенные примеры, мы можем сделать это в общих чертах. Легко видеть, что система HLS нуждается в дополнении как "снизу", так и "сверху".

Дополнение "снизу" должно заключаться в придании системе возможностей вставлять в схему, разработанную в целом на C++, отдельные фрагменты, закодированные вручную на VHDL (или на других языках, транслируемых в VHDL), наподобие ассемблерных вставок в Фортранные программы, популярных среди программистов в 60-70-е годы.

Дополнение "сверху" должно представлять собой препроцессор, возможно, даже интерактивный, почти наверняка - проблемно-ориентированный, реализующий в автоматическом и/или автоматизированном режиме преобразования текста на C++ из категории "написать требуемый алгоритм на C++ таким образом, чтобы компилятор построил по этой записи конвейер, в принципе можно, хотя сделать это вручную, возможно, и нелегко". Если при дополнении "снизу" появится необходимость в неудобных для написания и восприятия человеком языковых конструкциях, прикрыть их подходящей формой записи мог бы тот же препроцессор.

3. Вставки фрагментов на VHDL в схему на C++

В традиционном программировании вставка фрагмента на языке ассемблера в программу на C++ - обычное дело. Для этого достаточно оформить вставляемый ассемблерный фрагмент как отдельную функцию, и вызвать ее из текста на C++. При компиляции текста на C++ в текст на VHDL при помощи Vivado HLS функции превращаются в компоненты, а их вызовы - в вызовы (вставки) компонентов. Тот факт, что компонент является определением не функции, а макроса, принципиально ситуацию не усложняет. Казалось бы, можно ожидать, что вставка в описание схемы на C++ компонента, написанного на VHDL, окажется таким же простым, рутинным делом, как вызов функции,

написанной на языке ассемблера, из программы на C++.

Тем не менее, в Vivado HLS возможность вставки в схему компонентов, написанных на VHDL, не предусмотрена. Не встречались нам и ссылки на то, что ее реализовал кто-то из сторонних пользователей: напротив, в форумах разработчиков заикнувшихся на эту тему новичков обычно сразу же одергивают более опытные коллеги.

Ничего удивительного в этом в действительности нет. Вставка в схему независимо разработанного фрагмента, неважно, на каком языке написанного, в принципе гораздо сложнее, чем аналогичное действие для традиционной программы. Это объясняется тем, что схема обладает гораздо большей степенью "внутренней связности", чем программа, и спецификация интерфейса между вызывающей частью схемы и вставляемым в нее инородным фрагментом может оказаться очень непростым делом. Понимая это, разработчики Vivado HLS приняли решение эту действительно очень сложную задачу не только не решать, но даже и не ставить: компилятор из C++ в VHDL вообще не поддерживает раздельную компиляцию.

Нам такой подход представляется слишком ограничительным. Далее мы рассмотрим несколько частных случаев, в которых вставка инородного фрагмента в схему все же возможна, и для каждого такого частного случая научимся ее выполнять.

Очевидное препятствие на нашем пути - отсутствие возможности раздельной компиляции. Впрочем, способ преодоления этого препятствия тоже довольно очевиден. Когда нам потребуется вставить в схему на C++ компонент, написанный на VHDL, мы сначала запишем этот компонент в виде функции на C++. В результате ее компиляции получится компонент на VHDL. Если мы позаботимся о том, чтобы его интерфейс с вызывающей частью схемы был прост и предсказуем, то ничто не может помешать нам просто выбросить текст на VHDL, изготовленный компилятором Vivado HLS, и заменить его своим, с тем же внешним интерфейсом, но с более эффективной "начинкой".

3.1. Вставка компонента, активируемого асинхронно

Итак, мы компилируем при помощи Vivado HLS текст на C++, который состоит из вызывающей программы и вызываемой функции. В результате получается текст на VHDL, в котором вызывающей программе соответствует внешняя часть схемы, вызываемой функции соответствует компонент, а вызову функции - вставка компонента. Нам хотелось бы заменить компонент на VHDL, изготовленный компилятором, на свой, написанный на VHDL вручную. Для этого необходимо сохранить внешний интерфейс компонента, на который рассчитана вызывающая часть схемы. Если бы мы имели дело с обычным компилятором, порождающим не описание схемы, а программу, нужная нам замена выполнялась бы тривиально. Объясняется эта тривиальность тем, что вызывающая программа и вызываемая функция по определению работают строго поочередно. В схеме в принципе вполне возможны ситуации, когда

вызываемая функция еще работает, а вызывающая программа тоже уже начала работать, и в их одновременном доступе к одним и тем же переменным проявились те или иные некорректности.

Но "возможны" - не значит "есть". Компилятору Vivado HLS ничто не запрещает порождать схемы, в которых внешняя часть и вставленный в нее компонент работают строго поочередно, подобно вызывающей программе и вызываемой функции. Если они (части схемы) при этом синхронизируются явными сигналами запуска и завершения работы, то есть всегда ожидают завершения работы друг друга, то замена изготовленного компилятором компонента на свой, разработанный независимо, становится такой же простой задачей, как вызов ассемблерной функции из программы на C++.

Условия, при соблюдении которых компилятор Vivado HLS изготавливает именно такие схемы, оказались неожиданно простыми. Достаточно того, чтобы вызов функции не попадал в область действия прагм "**pipeline**" и "**dataflow**", а сама функция не содержала побочных эффектов, кроме присваиваний новых значений выходным параметрам. Впрочем, имеется и пара "подводных камней".

Чтобы компилятор не "заоптимизировал" объявление компонента в порождаемой схеме, объявление вызываемой функции должно содержать прагму:

```
#pragma HLS inline=off
```

Из тех же соображений нельзя делать тело предназначенной "на выброс" функции на C++ пустым: компилятор это обнаружит, и просто не станет изготавливать ни компонент, ни его вставку. Если же тело функции "на выброс" сделать не совсем пустым, но тривиальным, ограничившись написанием чего-то вроде:

```
a = 0;
```

то компилятор начнет выбрасывать из интерфейса создаваемого компонента ненужные, по его мнению, сигналы, которые в окончательном, рабочем варианте компонента, напротив, нужны.

Нам неизвестен простой и внятный способ заставить компилятор Vivado HLS этого не делать, кроме написания в качестве тела функции "на выброс" пусть и неэффективной, но правильной по смыслу версии алгоритма на C++. Это не только упрощает отладку, но и гарантирует, что реально используемые сигналы в выбрасываемом и окончательном вариантах компонента - одни и те же. Конкретный интерфейс компонента, активируемого асинхронно, подробно описан в [3], и соответствует типу интерфейса **ap_ctrl_hs** для блоков.

Теперь следует признать, что в проведенных только что рассуждениях мы допустили некоторые упрощения, требующие дополнительных пояснений. Например, мы сознательно оставили без комментариев применение к частям схемы понятия "работают по очереди", прекрасно понимая, что все части схемы, в отличие от команд традиционной программы, всегда работают, и никакого "по очереди", строго говоря, просто не может быть.

Говоря о поочередной работе внешней части схемы и вставленного в нее компонента, мы, конечно, имели в виду, что они строго поочередно осуществляют доступ к совместно используемым регистрам и блокам памяти (в терминах программы - к переменным и массивам). Таковыми являются параметры-массивы и выходные скалярные параметры вызываемой функции.

Как мы знаем из [2], возможность записи в один и тот же регистр из разных источников в разные моменты времени обеспечивается мультиплексированием. Но тогда возникает совершенно естественный вопрос о том, будут ли необходимые мультиплексоры построены во внешней части схемы, изготовленной компилятором Vivado HLS, или же, напротив, внешняя часть схемы рассчитана на наличие этих мультиплексоров в вызываемом компоненте. В последнем случае разработчику компонента, кодируемого вручную, придется изготавливать как необходимые мультиплексоры, так и логику управления ими.

В Vivado HLS по этому вопросу принято исключительно разумное решение, как будто специально рассчитанное на реализацию конструируемой нами возможности вставки независимо разработанных компонентов:

- за мультиплексирование сигналов, соответствующих одиночным выходным параметрам, отвечает целиком и полностью вызывающая часть схемы;

- при передаче массива в качестве параметра, мультиплексоры доступа к регистрам управления соответствующим блоком памяти также находятся в вызывающей части схемы, но в управлении ими участвует вызываемый компонент. Для этого среди сигналов, реализующих передачу массива в качестве параметра, имеется сигнал "**CE**" (**C**hip **E**nable), активирующий доступ к регистрам управления соответствующим блоком памяти со стороны вызываемого компонента. Эти сигналы для всех массивов-параметров вызываемый компонент имеет право активировать только в периоды своей активности. Внешняя часть схемы, со своей стороны, не пытается активировать свои ветви мультиплексора доступа к этим блокам памяти, пока вызываемый компонент активен.

Таким образом, вставку компонента, активируемого асинхронно, оказалось гораздо проще выполнить, чем объяснить. Изготовление инородного компонента, призванного заменить компонент, изготовленный компилятором Vivado HLS, сводится к реализации довольно простой временной диаграммы на небольшом числе управляющих сигналов. Простота взаимодействия с вызывающей частью схемы в данном случае обеспечивается наличием явных периодов активности компонента, которые начинаются и заканчиваются явными же сигналами.

Рассмотренная нами возможность позволяет "грубо" заменять большие куски логики, с компиляцией которых Vivado HLS справился плохо, на разработанные вручную. Для тонкого вмешательства в генерируемую компилятором схему эта возможность не годится.

Рассмотрим случай цикла, выполняющегося **K** раз, в котором выполняются некоторые сложные вычисления, а их результаты на каждой итерации цикла накопительно суммируются и полученное значение используется на следующей итерации цикла в качестве параметра. Пусть для этих "сложных вычислений" компилятор Vivado HLS смог построить однотактный конвейер, что позволяет выполнить все вычисления без накопительного суммирования за **N** тактов. Включение в этот конвейер накопительного суммирования, как мы знаем из примера №1, сразу увеличит скважность конвейера до 5 тактов, то есть, с точностью до "краевых эффектов", время выполнения цикла составит **5N** тактов, а всех вычислений **5*N*K**.

Мы можем сократить внутренний цикл до **2N**, если один синхронный конвейер заменим на два. В первом конвейере за N тактов произведем "сложные вычисления" и запишем их во вспомогательный массив. А во втором — вызовем написанный вручную на VHDL компонент, который за N тактов просуммировал бы вспомогательный массив в темпе один такт на слагаемое.

Хотелось бы, однако, выполнять суммирование в том же конвейере, что и вычисления, чтобы обойтись суммарным временем в **N** тактов, и без вспомогательного массива. К сожалению, техника вставки компонентов, активируемых асинхронно, не позволяет нам сделать это. Для достижения этой цели нам потребуется более сложная техника, а именно - техника внедрения инородной логики в части схемы, управляемые синхронно, только по глобальному таймеру, то есть не имеющие явных сигналов пуска и готовности. Грубо говоря, вместо дисциплины взаимного согласования с партнером каждого своего шага нам необходимо научиться «ходить в ногу», т.е. создавать вставки компонентов, активируемых **синхронно**.

4. Благодарности

Авторы благодарны Смольянову Юрию Павловичу за идею этой работы, за ценные замечания.

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 (v2018.3) December 20, 2018. URL:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives Дата обращения 8.01.2019г.

4. 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 Дата обращения 8.01.2019г.

Оглавление