Отделение математических наук

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

Как и почему могут быть использованы на практике суперкомпьютеры на базе FPGA

Москва 2017

УДК 004.78
ББК 32.973
К16

ISBN 978-5-906906-61-8

Как и почему могут быть использованы на практике суперкомпьютеры на базе FPGA

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

(ИПМ им. М.В. Келдыша РАН)

Аннотация: *Суперкомпьютеры на базе FPGA в принципе позволяют решить многие актуальные проблемы, стоящие сегодня перед отраслью высокопроизводительных вычислений. Однако на практике их применение в качестве вычислительных систем общего назначения сталкивается с целым рядом трудностей. В работе делается попытка систематизировать эти трудности и предложить пути их решения. Особое внимание уделяется обоснованному выбору классов задач, для решения которых такие суперкомпьютеры особенно эффективны.*

Ключевые слова и фразы: *суперкомпьютер, FPGA, высокопроизводительные вычисления, программистская модель, разработка приложений.*

Sergey Sergeevich Andreev, Svetlana Alekseevna Dbar, Aleksey Ottovich Lacis, Elena Aronovna Plotkina

Abstract: *In principle, FPGA-based supercomputers can solve many serious problems the HPC industry is facing today. But the attempts to use them in general-purpose supercomputing has not yet been successful due to numerous difficulties. In this paper, we try to classify those difficulties, and to offer some solutions. Special attention is paid to the choice of applications for which those supercomputers are especially good.*

Key words and phrases: *supercomputer, FPGA, high-performance computing, programming model, application development.*

Введение

Производительность единичного процессорного ядра универсального процессора на вычислительных задачах практически не растет уже почти 15 лет. Этот факт оставляет для традиционных многопроцессорных суперкомпьютеров единственный путь роста производительности – наращивание числа процессорных ядер в системе. Развитие по этому пути имеет несколько серьезнейших фундаментальных ограничений, важнейшее (но не единственное) из которых – рост потребления энергии. Все это заставляет разработчиков перспективных суперкомпьютеров все больше задумываться не просто об их эволюционном совершенствовании в рамках старых подходов, а о смене архитектуры [1].

Замечание. Под архитектурой здесь и далее будем понимать видимую для программиста логическую организацию вычислительной системы [2].

Первой успешной попыткой создания новой архитектуры было появление гибридно-параллельных систем на базе GPGPU [2,3]. Опыт последних лет показал как несомненные успехи на этом пути, так и то, что об окончательном решении проблемы говорить пока рано. Нужны еще гораздо более эффективные архитектуры. Наиболее радикальным подходом к архитектурному совершенствованию вычислителя является прямая схемная реализация алгоритма, что технически легко реализуемо при использовании FPGA. О громадном потенциале FPGA-ускорителей говорят уже много лет [4,5], но серьезных применений таких машин для сколько-нибудь широкого круга задач нет до сих пор. В этой работе мы постараемся проанализировать проблему, а главное – показать, для каких задач и при каких условиях такие машины можно (и нужно) строить уже сегодня.

1. Чем плох универсальный процессор

Идеал улучшить нельзя. Прежде чем ставить задачу заменить успешно отработавшую, постепенно улучшаясь эволюционным путем, без малого 70 лет архитектуру чем-то радикально новым, необходимо понять, что именно в этой старой архитектуре плохо. Также для начала нелишне было бы определить сами понятия «хорошо» и «плохо», то есть ввести метрику для сравнения архитектур.

На первый взгляд, вполне разумной метрикой кажется быстродействие, измеренное в арифметических операциях с вещественными числами в секунду (во флопсах). Конкретные вычислительные устройства, например, некий графический процессор X и некий универсальный процессор Y обычно сравнивают именно так. Иначе обстоит дело, если надо сравнить архитектуры, то есть ответить на вопрос о том, стоит ли вообще разрабатывать технологию замены универсального процессора графическим, или от использования графического процессора лучше отказаться. Да, графический процессор X считает в N раз быстрее универсального процессора Y. Но почему для сравнения был выбран именно Y, а не более быстрый универсальный процессор Z? Почему, наконец, мы должны использовать именно X, а не N штук Y-ов, поставленных рядом? Измеряя только быстродействие во флопсах, на эти и подобные им вопросы ответить невозможно. Однако флопсовую метрику легко модифицировать так, чтобы она стала пригодной и для сравнения архитектур. Для этого достаточно измерять не просто быстродействие, а удельное быстродействие в расчете на ватт потребляемой электрической мощности (во флопсах на ватт, или флопах на джоуль). Сравнение в этой метрике показывает нам, какая архитектура лучше использует потребляемую электрическую мощность для совершения полезной вычислительной работы, то есть является качественно более совершенной. При условии, конечно, что сравниваемые устройства выпущены на элементной базе примерно одного технологического уровня, то есть, грубо говоря, примерно в одно и то же время. Отметим, что для средних, а особенно – для больших вычислительных систем, проблема экономии энергии как таковая стоит очень остро, так что метрика, связанная с энергетической эффективностью, актуальна и сама по себе, а не только как технический критерий совершенства архитектуры.

Определив, таким образом, понятия «хорошо» и «плохо», оценим с этой точки зрения универсальный процессор как архитектуру. Любое вычислительное устройство, и процессор – в том числе, выполняет по ходу вычислений два вида работы:

– собственно вычисления (арифметические операции с вещественными числами);

– вспомогательные действия, которые необходимы, чтобы собственно вычисления могли произойти, например, вычисления адресов операндов арифметических операций.

Объем вычислительной работы определяется, в целом, алгоритмом и численным методом, то есть формулами, по которым необходимо считать (хотя формулы, конечно, тоже можно написать по-разному). Объем вспомогательных действий, напротив, очень сильно зависит от архитектуры вычислителя. Так, при работе универсального процессора промежуточный результат между двумя арифметическими операциями может сохраняться в оперативной памяти, а может – в кеше, в адресуемом регистре процессора или же во внутреннем регистре арифметического устройства. Во всех этих случаях накладной расход на запоминание промежуточного результата после первой операции и считывание его перед второй будет разным, а сам набор упомянутых возможностей прямо определяется архитектурой. Например, число адресуемых регистров и объем кеша в разных процессорах могут быть разными, вплоть до полного их (регистров и кеша) отсутствия, а оперативная память может быть однородной или неоднородной.

В любом случае вспомогательные действия являются накладным расходом. Попробуем грубо оценить вклад этих действий в общий объем работы.

Разделение на работу «полезную» и «вспомогательную» можно трактовать двояко:

– или как разделение работы на вычислительную и коммуникационную, учитывая, что вспомогательные действия, так или иначе, связаны либо с перемещением данных между функциональными устройствами процессора, либо с подготовкой таких перемещений;

– или как разделение выполняемых процессором инструкций на полезные (арифметические) и вспомогательные (все остальные).

С первой точки зрения, полезно было бы сопоставить энергозатраты на вычислительную и коммуникационную работу, со второй – количество полезных и вспомогательных инструкций, выполняемых процессором в единицу времени, поскольку по длительности выполнения арифметические инструкции от остальных в современных процессорах не отличаются. Если мы правильно выбрали наши точки зрения, увидеть с них мы должны примерно одно и то же.

Взгляд с обеих точек зрения показывает, действительно, примерно одно и то же, а именно – удручающе низкий объем полезной (арифметической) работы по сравнению со вспомогательной (коммуникационной) [1,2,4,5].

Для того, чтобы убедиться, что вспомогательные инструкции подавляющим образом доминируют в любой разумной программе, достаточно посмотреть на ее ассемблерный код. Как следствие, отношение полезного вычислительного быстродействия во флопсах к пиковому быстродействию, иногда называемое для краткости «к.п.д. процессора», лежит для современных процессоров где-то в окрестности 10%, а на некоторых классах задач падает до долей процента (алгоритмы, на которых к.п.д. близок к единице, существуют, но чрезвычайно редки) [2].

Для сравнения энергозатрат на арифметическую и коммуникационную работу в ходе вычислений на процессоре общего назначения также есть хорошо известные цифры. Примерно 3-4 года назад один флоп стоил примерно 70 пДж, и эта величина устойчиво снижается по ходу прогресса элементной базы. Коммуникационная работа по обслуживанию одного флопа в процессоре общего назначения стоила в это время от одной до нескольких тысяч пДж, и на протяжении последних лет эта величина довольно стабильна [6].

Таким образом, вклад вспомогательных (коммуникационных) действий в общий объем вычислительной работы универсального процессора не просто велик – он огромен. Снизив этот вклад, действительно можно выиграть очень много в любой разумной метрике. Для этого необходимо спроектировать вычислительное устройство с такой архитектурой, которая позволяла бы организовать вспомогательное перемещение данных во много раз экономнее, чем это делает универсальный процессор. Наиболее распространенная и успешная на сегодня альтернативная вычислительная архитектура такого рода – графический процессор общего назначения (GPGPU) [3]. Эта архитектура в несколько раз эффективнее универсального процессора, но все же недостаточно эффективна, чтобы считать архитектурную проблему полностью закрытой [4].

Конечно, универсальный процессор обладает громадным преимуществом универсальности и простоты программирования перед всеми альтернативными архитектурами. Однако, прекращение роста вычислительного быстродействия единичного процессора (в сегодняшних терминах – процессорного ядра) лишает нас возможности «прощать» универсальному процессору его крайне низкую эффективность, вынуждает искать гораздо более эффективные архитектуры.

2. Плохие новости для программистов

Из сказанного выше следует несколько важных выводов, которые вряд ли способны обрадовать прикладных программистов, разрабатывающих программы вычислительного характера. Выводы эти – общего плана, они касаются большинства, если не всех, разрабатываемых сегодня (и завтра) новых вычислительных архитектур.

2.1. Не всякий алгоритм способен ускориться на вычислителе с новой архитектурой

Проблема не столько в том, что преимущества вычислителей с новой архитектурой перед универсальным процессором лежат в сфере организации коммуникаций, сколько в том, что никакими другими преимуществами новые архитектуры не обладают. Арифметические операции выполняются как в универсальном процессоре, так и в GPGPU одинаково хорошо. Однако далеко не всякий алгоритм можно ускорить, улучшив именно коммуникации. Улучшить разом все коммуникации, подобно тому, как когда-то рост рабочей частоты ускорял разом все вычисления, физически невозможно: для этого потребовалось бы соединить каждое функциональное устройство внутри компьютера с каждым бесконечно быстрым каналом связи, не потребляющим энергии. Значит, коммуникации придется улучшать выборочно, путем построения неоднородной, иерархической системы памяти (вспомним, как ускорилось в свое время быстродействие универсальных процессоров с появлением кеш-памяти большого объема). Но иерархическая память позволяет ускорить только те алгоритмы, где есть локальность обращения к данным. Например, случайный, равномерно распределенный доступ к памяти большого объема невозможно ускорить кешированием [5].

2.2. Никакой алгоритм не способен ускориться на вычислителе с новой архитектурой сам собой

Сложно устроенная иерархическая память (такая, например, как в GPGPU) ускоряет только те алгоритмы, в которых локальность обращения к данным не просто возможна в принципе, но явно обеспечена программистом в тексте программы путем

правильной организации циклов. В свое время появление в составе универсального процессора кеш-памяти большого объема оказалось успешным во многом благодаря тому, что это решение автоматически, без участия программиста, способно заметно ускорить более или менее произвольную программу. Участие же программиста в переделке программы с целью получить еще большее ускорение, способно давать постепенно накапливающийся эффект: немного лучшая организация некоторых циклов дает некоторое ускорение еще лучшая организация еще большего числа циклов еще немного ускоряет работу программы и так далее. Со сложно организованными системами иерархической памяти, характерными для большинства новых архитектур, такое невозможно. Если в программе для GPGPU хочется ускорить векторную обработку, поместив данные в расслоенную память (shared memory), и необходимый объем данных в эту память влезает почти, но не совсем и не всегда, получить даже минимальное ускорение не удастся [3,4,5].

Такая ситуация разительно отличается от картины мира, сформировавшейся за многие годы в головах программистов вычислительных задач, особенно – в головах «программистов поневоле», для которых программирование не является основной целью производственной деятельности. Эта картина мира основывается на следующей причинно-следственной цепочке: «Новый процессор – быстрее старого. Значит, сам факт переноса более или менее любой программы на новый процессор и дает основной объем ее ускорения. Подгонка программы под архитектурные особенности нового процессора – дело, конечно, хорошее, но уж очень хлопотное и не обязательное. Пусть моя программа ускорится не в 50, а только в 20 раз – я буду доволен. Главное – перенести программу на новый процессор физически, для чего нужен компилятор языка, на котором написана моя программа. Если такой компилятор на новом процессоре есть, то задачу переноса можно считать в принципе решенной».

Применительно к новым архитектурам в этом нехитром рассуждении не является верным ни один вывод. Механический перенос программы на новый процессор сам по себе почти никогда ее не ускоряет. Глубокая структурная перестройка программы с явным учетом архитектурных особенностей совершенно необходима. Отказ от нее уменьшит ожидаемое ускорение не на десятки процентов, а в десятки и сотни раз. Наличие компилятора языка,

на котором программа написана, облегчает ситуацию незначительно, и уж точно не является синонимом решения проблемы: программу ведь все равно придется переписывать. Инкрементальное улучшение текста программы зачастую невозможно. Словом, думать так, как описано выше, подобно ожиданию того, что не распараллеленная, последовательная программа, будучи механически перенесена на суперкомпьютер из первой десятки top-500, ускорится от этого – ну, пусть не в 100000, но хотя бы в 1000–2000 раз.

2.3. Традиционного языка программирования не будет достаточно

Упомянутая выше глубокая структурная перестройка программы нужна для того, чтобы выразить в переработанном тексте архитектурные понятия нового процессора. Например, обработку большого массива от начала до конца, возможно, потребуется заменить на последовательную обработку небольших блоков этого массива, чтобы на деле обеспечить локальность доступа к данным. В простейшем случае, если мы хотим лучше использовать кеш-память, одной этой реорганизации циклов будет достаточно. В более сложном случае (то есть при переносе программы на процессор с более сложной новой архитектурой) может потребоваться, например, скопировать обрабатываемый блок в некоторую специальную память, с которой новый процессор работает особенно хорошо. Но в традиционных языках программирования понятия «некоторая специальная память» не предусмотрено. В общем случае программисту следует быть готовым к тому, что в том языке программирования, на котором написана подлежащая переносу программа, просто нет тех понятий, без явной записи которых программа не способна ускориться. Например, в традиционном Фортране нет понятия «послать сообщение», без которого невозможно переписать программу в параллельной форме: в результате, в добавление к вполне традиционному Фортрану приходится использовать MPI, в котором необходимые понятия есть. Конечно, существуют компиляторы, позволяющие выразить недостающие понятия в виде прагм (псевдокомментариев), но сути дела это не меняет. Именно по этой причине, попытка использовать на новом процессоре традиционный язык программирования может принести не пользу, а вред. Например, языки,

применяемые для программирования GPGPU, довольно заметно отличаются от традиционных. Конкурирующая с GPGPU архитектура Xeon Phi позволяет писать программу на традиционном языке. Казалось бы, путь к ускорению программы при этом сокращается. В действительности же он становится длиннее – написать на традиционном языке программу таким образом, чтобы Xeon Phi «догадался», как ее ускорить, оказывается не проще, а сложнее, чем при помощи специального языка, на котором все необходимые приемы ускорения можно выразить явно [2,3,5].

2.4. Выводы

На протяжении длительного времени между разработчиками процессоров и прикладных программ существовало простое и понятное разделение труда: программист пишет программу, разработчик процессора обеспечивает ее быстродействие. Новый процессор по определению ускоряет старую программу. Это время прошло и больше не вернется. Возможности строить новые процессоры, все более и более ускоряющие произвольные программы просто в силу своей новизны, больше нет, и в обозримом будущем она вряд ли появится.

Теперь прикладному программисту приходится учитывать и явно отображать в тексте программы конкретные особенности новых вычислительных архитектур. Большинство из них нацелено на оптимизацию перемещения данных, но конкретный вид этой нацеленности для разных архитектур может быть разным. Программист вынужден все это знать. И не просто знать, а узнавать снова и снова, по мере появления все новых архитектур. Появления одной, господствующей длительное время, универсальной новой архитектуры не предвидится. Переход от последовательного программирования к классическому параллельному оказался не порогом, как казалось еще лет 10 назад, а первой ступенькой крутой и высокой лестницы. Освоение новых архитектур, вообще говоря, неотделимо от освоения новых по набору базовых понятий языков программирования.

Даже при условии самого глубокого и заинтересованного освоения программистом всех архитектурных тонкостей ускорить в принципе можно не всякий алгоритм, а только тот, который удается явно записать в мелкоблочной форме. Конкретные новые архитектуры могут налагать дополнительные ограничения.

Реальные программисты, особенно – «программисты поневоле», могут не выдержать указанных сложностей и отказаться от использования новых архитектур. Гипотетическое добавление в процесс разработки прикладной программы фигуры «кодировщика, знающего железо», о чем говорится, например, в [1], проблемы в общем случае не решает: ведь кодировщик не знает алгоритмов и методов, а без этого знания правильная структурная переработка программы невозможна.

3. Если не универсальный процессор, то что?

Итак, нам нужна архитектура, предельно эффективная именно в коммуникационном отношении. Из самых общих соображений очевидно, что экономнее всего можно организовать коммуникации не в общем виде, а для конкретного алгоритма. Иными словами, наиболее эффективным в коммуникационном отношении мог бы быть «процессор одной задачи», реализующий вычислительный алгоритм непосредственно в «железе» (точнее, в кремнии) в виде цифровой электронной схемы, способной выполнять этот, и только этот, алгоритм. Чтобы такая реализация алгоритмов непосредственно в кремнии была технически возможна за обозримую цену, кремний должен быть реконфигурируемым, то есть материалом для схемной реализации алгоритмов должны быть FPGA [4,5].

Конечно, прямая схемная реализация алгоритма в кремнии – далеко не единственный возможный подход к созданию эффективных вычислительных архитектур, но мы этого и не утверждаем. Утверждаем мы только то, что эффективность вычислительных архитектур – это достижимая в их рамках эффективность организации коммуникаций, особенно – на микроуровне, и что прямая схемная реализация алгоритма открывает для достижения этой коммуникационной эффективности весьма широкий круг возможностей. Другие подходы, возможно, столь же или даже более эффективные, существуют, но в настоящей работе не рассматриваются.

В каждой альтернативной архитектуре применяются свои способы оптимизации коммуникаций. В вычислителях на базе FPGA самым мощным, неоднократно доказавшим свою эффективность приемом оптимизации коммуникаций является синхронный конвейер.

Само название этого приема построения схем отсылает нас к (авто)сборочному конвейеру. Аналогия с ним, действительно, очень полная и точная.

Вычислительный конвейер, реализующий, например, цикл многократного расчета по некоторым формулам, содержит в себе много комплектов рассчитываемых данных в разной степени готовности. Продвигается он дискретными шагами. Время между двумя последовательными шагами (оно же – время интервала подачи исходных данных, оно же – время интервала получения окончательных результатов) называется **интервалом инициализации**, или **скважностью**, конвейера. В случае вычислительного (не автосборочного) конвейера время дискретно, измеряется в тактах рабочей частоты, и скважность – тоже. Максимальное быстродействие схемы получится при скважности равной 1 (однотактный конвейер). Время прохождения конкретным набором исходных данных всего конвейера целиком по самому длинному пути, от начала до конца, называется **глубиной**, или **латентностью**, конвейера. В случае вычислительного конвейера латентность, как и скважность, измеряется в тактах рабочей частоты. Наконец, конвейер может содержать не одну, а несколько параллельных ниток. Их число принято называть **шириной** конвейера.

Очевидно, что более длинные конвейеры можно строить из более коротких, если у последних одинаковая скважность. Элементарными конвейерами при построении схем в FPGA являются, в основном, арифметические устройства (сложители, вычитатели, умножители) и блоки адресуемой памяти. Все они являются однотактными конвейерами с различной латентностью.

Конвейер хорош тем, что сам для себя является памятью промежуточных результатов, исключает как ненужные перемещения данных, так и простои функциональных устройств. Кроме того, он позволяет использовать высоколатентные функциональные устройства, а они гораздо экономнее по объему оборудования, чем низколатентные, применяемые в стандартных процессорах – значит, при том же объеме исходного оборудования их можно реализовать больше. Современные кристаллы FPGA позволяют строить конвейеры из сотен арифметических операций, что и обеспечивает беспрецедентный объем полезной работы в среднем за такт. Платой за все это великолепие является отсутствие гибкости, свойственной программируемым вычислителям. Конкретный длинный конвейер можно построить только для реализации одной конкретной, известной заранее вычислительной процедуры. Поэтому технология длинных синхронных конвейеров и является основной именно в вычислителях на базе FPGA.

4. Чем плох вычислитель на базе FPGA

На первый взгляд может показаться, что проблема создания супер-эффективной вычислительной архитектуры решена полностью. К сожалению, это не так. Кратко упомянутая выше архитектура обладает в реализации внушительным количеством ограничений и узких мест, что делает перспективы ее применения вовсе не такими безоблачными, как могло бы показаться. Перечислим основные из них.

4.1. Низкая (в 20-30 раз ниже, чем у процессора общего назначения) рабочая частота реализуемых схем

Причин такой низкой частоты – две. Во-первых, сама реконфигурируемость FPGA значительно снижает допустимую частоту схем, реализуемых в них. Во-вторых, «одноразовый» характер схем, изготовление их разработчиками, не имеющими или почти не имеющими специальной подготовки в области электроники, исключают длительную (и вообще всякую ручную) работу над повышением качества физической реализации схемы, что только и могло бы повысить рабочую частоту. Обе причины, как легко видеть, имеют не случайный, а системный характер, то есть, скорее всего, будут действовать долговременно. По крайней мере, за последние 20 лет рабочая частота схем вычислительного характера, реализуемых в FPGA, практически не изменилась, хотя объем оборудования в составе FPGA вырос за это время на порядки [7].

Для разработчика вычислительного приложения это означает, что схемы, которые обгоняют процессор по объему работы, выполняемой за один такт, всего в 20-50 раз, интереса не представляют. Обгонять процессор по объему работы в расчете на такт надо хотя бы в несколько сотен раз. То есть на логическом уровне эффективность внутренних коммуникаций должна быть **очень** высокой (конвейер – очень длинным, очень широким и, в идеале, единственным).

4.2. Малый объем памяти

Чтобы «прокормить» длинный конвейер, необходима одновременная работа многих независимых запоминающих устройств,

причем синхронных (с постоянным временем выборки и записи). Внутри одной FPGA есть возможность построить сотни таких устройств, но их суммарная емкость составляет, в лучшем случае, около 16 мегабайт, чаще – 1–2 мегабайта [7]. При построении процессора одной задачи сотни и тысячи функциональных устройств, в том числе – устройств памяти, соединяются именно так, как нужно для реализации данного алгоритма. Возможность реализовать более или менее произвольный граф соединений достигается только внутри одной FPGA. Если бы мы попытались «вытащить» существующую внутри FPGA систему реализации необходимых соединений за пределы микросхемы, нам бы элементарно не хватило ее (микросхемы) ножек. Значит, все чудеса оптимизации коммуникаций, о которых мы говорили выше, достигаются в вычислителях на базе FPGA для объемов обрабатываемых данных, ограниченных первыми мегабайтами, что на три порядка меньше объема оперативной памяти современных вычислительных серверов на базе процессоров общего назначения. Но ведь процессор общего назначения, как и FPGA, не содержит память терабайтного объема внутри себя. Эта память подключается к нему извне. Почему бы не подключить внешнюю память к вычислителю на базе FPGA, подобно тому, как мы это делаем для универсального процессора?

Внешнюю оперативную память к FPGA подключают довольно часто, используют – значительно реже. Причин тому несколько.

Во-первых, в машинах на базе процессоров общего назначения именно канал связи процессора с памятью и является главным узким местом. Если мы хотим обогнать такую машину, то воспроизводить ее самое слабое место довольно странно.

Во-вторых (и это главное), воспроизводя свойственный процессору общего назначения канал связи с внешней оперативной памятью, мы дополнительно сужаем его примерно во столько же раз, во сколько раз ниже у нас тактовая частота (см. выше, в предыдущем подразделе). При попытке компенсировать низкую частоту организацией очень широкой шины данных мы рискуем опять упереться в элементарную нехватку ножек микросхемы. Похоже, мы опять столкнулись с проблемой, которая на первый взгляд кажется технической, в действительности же имеет системный характер.

В последние 3–4 года начали появляться технологии подключения к FPGA внешней оперативной памяти гигабайтных объ-

емов, например, при помощи высокоскоростных (с частотой 10 гигагерц и выше) трансиверов [9]. Пока эти технологии еще не стали распространенными повсеместно, и в настоящей работе мы их не рассматриваем, прекрасно понимая при этом, что их дальнейшее развитие может изменить очень и очень многое в вычислительной схемотехнике.

4.3. Облик вычислительной системы

Теперь мы вполне можем представить себе общую архитектуру вычислительной системы общего назначения, имеющей в своем составе FPGA. Это – гибридный кластер, похожий на широко распространившиеся в последнее десятилетие гибридные кластеры с GPGPU в составе вычислительных узлов. Вместо GPGPU роль ускорителя вычислений в составе такого вычислительного узла выполняет модуль на базе одной или нескольких FPGA 10–100-миллионного класса, связанных с универсальными процессорами своего узла широким каналом PCI Express. Внешней оперативной памяти для FPGA в составе такого ускорительного модуля нет.

Именно такой облик вычислительной системы на базе FPGA-ускорителей сложился исторически в тех немногих, очень узких и специальных, областях, где такие вычислительные системы уже давно используются на практике [8].

4.4. Основные ограничения на класс приложений

Из сформулированного только что облика вычислительной системы почти автоматически выводится облик приложения, которое имеет шанс быть ускоренным на такой системе. Прежде всего, используемый в приложении алгоритм должен допускать запись в мелкоблочной форме с характерным размером блока обрабатываемых данных от 1 до 10 мегабайт. Алгоритм обработки данных в пределах блока должен допускать конвейеризацию. Теперь уточним и конкретизируем эти ограничения.

4.4.1. Удельная вычислительная нагрузка при обработке мелких блоков

Записывая алгоритм в мелкоблочной форме, мы предполагаем, что каждый мелкий блок данных будет копироваться из па-

мяти универсального процессора в FPGA-ускоритель и обрабатываться в нем очень быстро, после чего результаты обработки будут копироваться в обратном направлении. Время копирования при этом есть накладной расход, который должен оправдываться ускорением обработки данных. Оправдается он в том случае, если обработка данных будет достаточно длительной. Осталось придать конкретный смысл слову «достаточно».

Пусть обработка блока данных в ускорителе требует передачи из памяти процессора в память ускорителя N1 байта исходных данных, и передачи N2 байтов результатов расчета в обратном направлении. Пусть при этом в процессе обработки необходимо выполнить F полезных флопов (операций над вещественными числами). Будем называть величину

$$L=F/(N1+N2)$$

удельной вычислительной нагрузкой на блок. Для простоты допустим, что скорости передачи данных в обоих направлениях одинаковы, и обработка не совмещается во времени с передачей данных. Тогда время накладного расхода на обработку блока данных в ускорителе – это в точности время передачи данных, и оно равно:

$$TT=(N1+N2)/VT,$$

где VT – это скорость передачи данных между процессором и ускорителем.

Чтобы обработка данных в ускорителе имела смысл, время их обработки в процессоре (TP) должно быть больше, чем время их обработки в ускорителе (TF) в сумме со временем передачи данных:

$$TP > TF+(N1+N2)/VT.$$

Выразив TP и TF через число полезных флопов, из которых состоит обработка, и производительность процессора (VP) и ускорителя (VF) в полезных флопсах, получим:

$$F/VP > F/VF+(N1+N2)/VT.$$

Поделив обе части на положительное число (N1+N2), получим:

$$L/VP > L/VF+1/VT$$

или

$$L/VP – L/VF > 1/VT.$$

Поскольку мы стремимся все-таки ускорить вычисления, VF > VP, и членом L/VF можно пренебречь по сравнению с L/VP.

В итоге получаем:

$$L/VP > 1/VT$$

или

$$L > VP/VT.$$

Назовем это неравенство необходимым условием применимости алгоритма к реализации на ускорителе. Оно означает, что удельная вычислительная нагрузка должна быть больше (на практике – много больше) частного от деления скорости обработки данных в процессоре (в полезных флопсах) на скорость их передачи между процессором и ускорителем.

Сопоставим это с характерными показателями сегодняшних процессоров и каналов связи. Поскольку речь идет о методах, представимых в строго мелкоблочной форме, обращение к данным в программе их обработки на процессоре, скорее всего, имеет регулярный характер, и мы можем грубо принять к.п.д. процессорного ядра равным 10%, а величину VP, соответственно, равной 500 000 000. Принимая VT равным 1 000 000 000, что грубо соответствует PCI Express Gen2 x4, получаем:

$$L >> 0.5.$$

Значение L у нас измеряется во флопах на байт, а в каждом вещественном числе двойной точности – 8 байт. Значит, хоть какой-то осмысленный разговор об ускорении по сравнению хотя бы с одним процессорным ядром начинается с алгоритмов, выполняющих над каждым переданным в ускоритель вещественным числом хотя бы несколько десятков арифметических операций, прежде чем результат обработки покинет ускоритель. На практике, с учетом всевозможных накладных расходов, требуется значение L от десятка и выше (около сотни флоп на переданное вещественное число).

Если алгоритм приложения невозможно представить в мелкоблочной форме с размером блока, умещающегося во внутреннюю память используемой FPGA, и/или обработка получившихся мелких блоков не удовлетворяет указанным требованиям к удельной вычислительной нагрузке, перенос такого приложения на гибридный компьютер с FPGA-ускорителями является заведомо бессмысленным. Если же этот тест пройден, сформулируем еще ряд важных ограничений.

4.4.2. Принципиальная возможность построения конвейера

Как мы уже упоминали выше, единственный на сегодня прием ускорения вычислений в FPGA, подтвержденный на практике – это построение длинного синхронного конвейера, лучше всего – однотактного (со скважностью 1). Часто бывает, что приходится оформлять приложение не как один конвейер, а как несколько конвейеров, работающих последовательно, или – как конвейер, который запускается, отрабатывает и опустошается в некотором внешнем цикле. В любом случае, вычисления, занимающие основное время в критическом цикле программной реализации алгоритма, должны быть конвейеризованы.

Однако, конвейеризовать можно не всякий алгоритм. Например, не поддается конвейеризации алгоритм, в основном цикле которого есть вложенный цикл с не известным заранее числом оборотов (такой алгоритм соответствует конвейеру с не известным заранее числом «постов сборки»). Эта ситуация – не самая плохая, поскольку вложенный цикл часто оказывается возможным выполнить фиксированное число раз. Гораздо хуже, если имеется рекуррентная зависимость входа одной из ступеней конвейера от результата работы последующих ступеней. Такой алгоритм соответствует ситуации, когда очередную операцию сборки «автомобиля» невозможно выполнить, пока предыдущий по ходу конвейера автомобиль не будет собран полностью. Или, выражаясь более строго, граф информационной зависимости алгоритма содержит обратную дугу. Эту неприятность можно побороть одним единственным способом: сократить длину обратной дуги до величины скважности конвейера. Например, в вычислениях, которые нуждаются в высококачественных генераторах псевдослучайных величин, вычисление очередного псевдослучайного числа, исходя из предыдущего числа, может потребовать на универсальном процессоре многих сотен тактов. Однако, существуют специальные варианты таких алгоритмов, в которых схемная реализация получения очередного числа псевдослучайной последовательности возможна за один такт. В любом случае:

Если алгоритм, подлежащий схемной реализации, не конвейеризуется, схемная реализация, с точки зрения ускорения, почти наверняка бессмысленна.

Наконец, еще одно важное ограничение: оборудования в составе используемой FPGA должно быть достаточно для построения конвейера. В случае системы из нескольких конвейеров, работающих последовательно, оборудование для их арифметических устройств можно, хотя бы теоретически, переиспользовать, задействовав одни и те же арифметические устройства в разных конвейерах в разные моменты времени. Оборудование же в составе одного конвейера переиспользовать нельзя: грубо говоря, в схеме должно быть ровно столько отдельных устройств сложения вещественных чисел, сколько знаков «+» содержится в формулах, реализуемых этим конвейером. Это ограничение особенно обременительно: получается, что в «хорошем» алгоритме арифметики должно быть «много, но не слишком много». Объем оборудования в современных FPGA позволяет реализовать от нескольких сотен до тысяч одновременно работающих арифметических устройств.

4.5. Выводы

Сформулированные выше соображения об исключительно высокой эффективности длинных синхронных конвейеров, построение которых неотделимо от схемной реализации алгоритма, по-прежнему справедливы. Однако, при ближайшем рассмотрении они «обросли» солидным количеством довольно жестких ограничений на алгоритмы, реализуемые таким эффективным образом. Обработка данных должна сводиться к обработке довольно мелких блоков, внутри которых обработка должна допускать конвейеризацию, и притом характеризоваться довольно высокой удельной вычислительной нагрузкой, но при этом достаточно простой логикой.

Также важно понимать, что схемная реализация алгоритма, будучи возможной в принципе, на практике требует весьма солидных затрат сил и рабочего времени (см. ниже). Далеко не всякий алгоритм, который можно в принципе «запихнуть» в FPGA-ускоритель, пусть даже с некоторым выигрышем по формальным показателям эффективности, стоит этих затрат с точки зрения реальных программистов, разрабатывающих реальный код. Чтобы предлагаемая архитектура имела шанс быть реально принятой программистским сообществом, на первых порах нужны задачи, вписывающиеся в названные выше ограничения с большим запасом.

5. Есть ли задачи для таких машин

В предыдущем разделе мы показали, что набор необходимых условий успешного использования вычислительной системы с FPGA-ускорителями весьма нетривиален. Отдельные приложения, удовлетворяющие всем указанным ограничениям, существуют, но для оценки перспективности предложенной вычислительной архитектуры необходимо понять, насколько таких приложений много, измеряется ли их количество отдельными штуками, или речь идет о целых классах задач. В этом разделе попробуем сопоставить с названными выше ограничениями некоторые классы приложений, традиционных для высокопроизводительных вычислений.

Выбор именно тех классов приложений, о которых идет речь ниже, не является случайным. Зная, какую большую роль в суперкомпьютерной отрасли играют задачи механики сплошной среды, мы начали свои исследования именно с них. Когда выяснилось, что для этих задач машины с FPGA-ускорителями не очень подходят, мы начали целенаправленно искать такие классы приложений, которые особенно хорошо удовлетворяют ограничениям Раздела 4. Рассмотренные ниже классы приложений – это то, что мы на сегодняшний день нашли.

5.1. Механика сплошной среды

Сеточные аналоги задач механики сплошной среды сводятся к СЛАУ с очень большими и очень разреженными матрицами специального (для индексных сеток) или общего (для неструктурированных сеток) вида. Решаются эти СЛАУ стационарными или нестационарными итерационными методами [10]. Нестационарные методы сходятся очень быстро, но не всегда, стационарные – гораздо медленнее, но надежнее. Исследование деталей возможной реализации этих методов на вычислительных системах с FPGA-ускорителями (и вообще с ускорителями нетрадиционной архитектуры) проводилось [11], но подробное его обсуждение выходит за рамки настоящей работы. Приведем здесь основные результаты этого исследования.

Нестационарные методы (например, крыловские) практически невозможно записать в мелкоблочной форме: потенциал локализации вычислений на нескольких мегабайтах данных в этих

методах чрезвычайно мал, если не равен нулю. В краткосрочной перспективе использование вычислительных систем с FPGA-ускорителями для реализации этих методов следует признать бессмысленным.

Стационарные методы, особенно – на индексных сетках, обладают несколько большим потенциалом локализации вычислений, причем для задач исследования течений жидкости или газа он выше, чем для задач моделирования свойств твердого тела (теплопроводность, упругость). Неплохими показателями удельной вычислительной нагрузки характеризуется, например, модельная задача моделирования течения вязкого сжимаемого газа в канале, исходный текст программы для которой был любезно предоставлен нам А.Е. Луцким (ИПМ им. М.В. Келдыша РАН). Однако эти задачи характеризуются достаточно сложной логикой, для реализации необходимых конвейеров годятся только очень большие FPGA.

В целом, сеточные аналоги задач механики сплошной среды, решаемые стационарными итерационными методами на индексных сетках, следует признать, с точки зрения целесообразности их реализации на машинах с FPGA-ускорителями, «пограничными». При наличии очень больших (примерно 100 000 000 эквивалентных вентилей) FPGA, соединенных с процессором общего назначения очень быстрыми (хотя бы PCI Express Gen3 x8) каналами передачи данных, перенос этих задач на этот класс машин можно считать оправданным в принципе. Учитывая высокую (на сегодня) трудоемкость разработки, инструментальные средства радикального снижения которой еще только предстоит создать, работы по реализации этого класса задач на машинах с FPGA-ускорителями следует отнести на среднесрочную перспективу. К первоочередным, призванным уже сегодня убедительно продемонстрировать потенциал предлагаемой архитектуры, эти работы не относятся.

5.2. Молекулярная динамика, квантовая химия и метод частиц

При всем различии указанных в заголовке подходов с физико-химической точки зрения, применяемые в них алгоритмы обладают общими свойствами, очень существенными (и благоприятными) для реализации на FPGA-ускорителях. Суть всех этих

алгоритмов в том, что рассматривается взаимодействие (обычно – попарное) дискретных частиц, из которых состоит изучаемая система (микрообъем металла или газа, большая молекула и т.п.). Системы эти бывают статическими (когда набор взаимодействующих частиц задан заранее и не меняется) и динамическими (когда частицы перемещаются в большом объеме, и необходимо следить за тем, какие частицы в данный момент времени находятся достаточно близко друг к другу для того, чтобы их взаимодействие имело смысл учитывать) [12].

Понятно, что мелкоблочная реализация алгоритмов моделирования динамических систем, по меньшей мере, весьма нетривиальна. Однако, даже ограничившись рассмотрением статических систем, мы получаем практически неисчерпаемый класс очень актуальных задач. Достаточно сказать, что к этому классу помимо задач моделирования тонких свойств металлов относятся очень и очень многочисленные (и крайне актуальные сегодня) задачи молекулярной биологии. Обрабатываемые данные в таких задачах настолько малы по объему, что зачастую целиком умещаются даже в средних, по сегодняшним меркам, FPGA, удельная вычислительная нагрузка огромна, а логика обработки достаточно компактна, чтобы оборудования для соответствующих конвейеров заведомо хватило. С точки зрения рассмотренных в подразделе 4.4 ограничений на свойства алгоритма, задачи эти просто созданы для вычислительных систем с FPGA-ускорителями (конечно, правильнее было бы сказать, что такие вычислительные системы должны быть созданы, в первую очередь, именно для этих задач).

Задачи моделирования статических систем также выигрышны с точки зрения преодоления «барьера вхождения» и практического принятия новой архитектуры сообществом разработчиков. Дело в том, что управляющая (высокоуровневая) логика алгоритмов, как и используемые в них структуры данных, очень просты, что делает возможным оформление программного (и, в нашем случае, схемного) кода в пакеты стандартных приложений [13]. Это, в свою очередь, отодвигает на задний план сложность и трудоемкость разработки схем для FPGA-ускорителей: будучи единожды разработанными, такие схемы будут использоваться снова и снова.

В нашем обзоре мы ничего не сказали о возможности параллельной реализации расчета на нескольких FPGA, точнее, на нескольких «связках» вида «процессорное ядро+FPGA-ускоритель». Для задач названного класса такая возможность реализу-

ется легко, но, по мнению всех специалистов по молекулярной биологии, с которыми нам удалось поговорить, она не нужна. Для этих задач характерен массовый счет большого количества независимых вариантов с разными исходными данными, так что несколько связок «процессорное ядро+FPGA-ускоритель» разумнее использовать как набор независимых вычислителей.

Высказанные здесь соображения подтверждаются на практике экспериментами с текстами модельных приложений из области молекулярной динамики, любезно предоставленными нам сотрудниками ИМПБ РАН А.Н. Коршуновой, В.Д. Лахно, И.В. Лихачевым и Н.С. Фиалко.

5.3. Обработка изображений

Многочисленность и актуальность приложений в этой области сегодня не вызывают никаких сомнений. Алгоритмы обработки изображений почти всегда допускают мелкоблочную реализацию. Достаточно высокая удельная вычислительная нагрузка характерна для алгоритмов обработки изображений высокого качества, в которых цвета пикселов представлены вещественными числами, а особенно – для алгоритмов, в которых используются свертки по скользящему окну. К последним относятся, например, алгоритмы вычисления оптического потока [14], а также алгоритмы, реализующие исключительно популярные сегодня сверточные нейросети. Успешность и сравнительная простота реализации таких алгоритмов проверялась экспериментально при заинтересованном участии С.М. Соколова (ИПМ им. М.В. Келдыша РАН).

5.4. Выравнивание белковых последовательностей

Задачи такого рода крайне многочисленны и актуальны в современной молекулярной биологии. Алгоритмически они сводятся к поиску максимально длинных и притом максимально похожих (но не идентичных) фрагментов в различных текстах большой длины. Все сказанное выше о задачах моделирования статических молекулярных систем в отношении пригодности этих задач к реализации на вычислительных системах с FPGA-ускорителями, относится в той же (если не в большей) степени и к задачам выравнивания белковых последовательностей. Уже при парном выравнивании (когда похожие фрагменты ищутся в двух текстах) эти задачи ха-

рактеризуются очень высокой удельной вычислительной нагрузкой, которая возрастает многократно при множественном (более двух текстов) выравнивании [15]. Реализация этих алгоритмов нами пока не выполнялась, но планируется.

5.5. Выводы

По мере роста нетрадиционности вычислительных архитектур (от единичного процессора общего назначения – через многопроцессорные вычислительные системы – к вычислительным системам с ускорителями, в особенности – с ускорителями на базе FPGA) универсальность их (архитектур) снижается. Переход к FPGA-ускорителям снижает универсальность вычислительной системы, по сравнению с системами более старых архитектур, довольно резко, но область применения таких систем остается достаточно широкой. К ней относятся, в первую очередь, актуальные задачи молекулярной биологии, некоторые не биологические задачи молекулярной динамики и сходные с ними, а также задачи обработки изображений, в особенности – на основе сверток. Строительство опытных образцов вычислительных систем с FPGA-ускорителями, а также разработку технологий и методик их прикладного программирования, следует ориентировать, в первую очередь, именно на такие приложения.

6. Технологии реализации приложений

Оценивать возможность и перспективность реализации приложений мы теперь умеем, осталось выполнить саму реализацию. Она состоит из трех частей:

– структурное преобразование исходного текста программного варианта приложения;

– написание программной части приложения (управляющей программы для процессора общего назначения);

– написание аппаратной части приложения (цифровой схемы ускорителя, реализуемой в FPGA) [16].

Замечание. В отношении разработки аппаратной части вполне уместно слово «написание», поскольку все сколько-нибудь разумные современные технологии разработки схем такой сложности подразумевают описание логики ее работы на языке, внешне напоминающем язык программирования, и последующую

трансляцию этого текста в схему без дополнительного ручного вмешательства.

6.1. Общая методика разработки

Наш опыт показывает, что заметное, если не основное, время реализации занимает поиск и устранение «глупых» ошибок в интерфейсах сопряжения программной и аппаратной частей. На втором месте – ошибки в смысловом соответствии схемы ее программному прототипу (в схеме нет «глупых» ошибок и формальных некорректностей, но она делает по смыслу не совсем то, что делала заменяемая этой схемой часть программы).

Присутствующие в современных схемотехнических САПР средства совместной разработки (co-design) программной и аппаратной частей гибридного приложения [17] предназначены для борьбы с ошибками именно такого рода, но для наших целей эти системы не очень подходят. Они не решают проблемы встраивания программной части приложения в среду разработки и исполнения гибридно-параллельных программ, используемую на гибридно-параллельных вычислительных кластерах. В результате, при их использовании появляются две версии программной части: «тестовая» и «настоящая», и все «глупые» ошибки сопряжений и смысловые несоответствия воспроизводятся на новом уровне.

Рис. 1. Структура гибридного приложения.

Этих недостатков лишена разработанная нами инженерная методика переноса приложений на вычислительную систему с FPGA-ускорителями. Подробно она описана в [16]. Суть методики заключается в следующем.

Приложение, как уже говорилось выше, состоит из **программной** и **аппаратной** частей. Одновременно с аппаратной частью разрабатывается и сопровождается ее (аппаратной части) **программная модель**. Эту программную модель можно в любое время подставить на место настоящей аппаратной части при сборке приложения, получив его (приложения) тестовый программный вариант. Структура гибридного приложения представлена на Рис.1.

Для предотвращения «глупых» ошибок важно, чтобы исходный текст программной части приложения не менялся (и не «в принципе», а совсем) в зависимости от того, какой вариант аппаратной части используется (настоящий или программная модель). Следовательно, программная часть должна общаться с обоими вариантами одинаково. Само по себе это довольно противоестественно. В то время, как для общения с настоящей аппаратной частью необходима специальная библиотека передачи данных между процессором и схемой, программную модель аппаратной части можно было бы просто вызвать, как обыкновенную функцию с аргументами. Тексты программной части в этих двух случаях окажутся разными, чего мы всеми силами стремимся избежать. Для этого связь между программной частью и программной моделью аппаратной части искусственно организуется через специальную «мини-библиотеку» – программную модель канала связи. Внешний интерфейс этой мини-библиотеки копирует внешний интерфейс настоящей библиотеки передачи данных, что и позволяет программной части приложения не зависеть от варианта аппаратной части.

Тестовый программный вариант приложения служит источником эталонных результатов счета (настоящий вариант должен давать те же результаты, что и тестовый). Исходный текст программной модели аппаратной части приложения, в свою очередь, служит формальным заданием на разработку настоящей аппаратной части. Особым является случай, когда разработка настоящей аппаратной части приложения ведется на аннотированном C++ с помощью компилятора Vivado HLS. В этом случае один и тот же неизменный текст на C++ с прагмами является одновременно исходным текстом и настоящей аппаратной части, и ее программной модели, в

зависимости от того, каким компилятором его компилировать. Это еще более упрощает разработку и комплексную отладку.

Методика подразумевает использование целого ряда программных и схемных инструментов собственной разработки, обеспечивающих программные и аппаратные интерфейсы между упомянутыми выше компонентами.

Исключительная важность того, чтобы тексты тестового программного и окончательного аппаратного вариантов приложения были не просто похожи, а в точности совпадали, ни в коей мере не является именно нашим «открытием». Существуют даже более радикальные варианты обеспечения единства текста, основанные на аккуратном низкоуровневом моделировании канала связи процессора с ускорителем, а также на непосредственном программном моделировании описания схемы на VHDL [21].

6.2. Разработка аппаратной части приложения

Итак, в нашем распоряжении – работоспособная программная модель аппаратной части приложения, являющаяся одновременно ее формальным описанием. Осталось эту аппаратную часть (в «настоящем» варианте) разработать.

Разработка, в любом случае, ведется на некотором языке схемотехнического проектирования (HDL), то есть ее результатом является текст описания схемы, по внешнему виду напоминающий программу на традиционном языке программирования.

Как мы отмечали выше, в подразделе 2.3, традиционные языки программирования процессоров общего назначения плохо приспособлены для программирования новых архитектур, в частности (и в особенности) – для описания логики цифровых электронных схем. Прикладные программисты, которым в нашем сценарии теперь отводится роль разработчиков – схемотехников, плохо приспособлены для восприятия других языков с их картиной мира, принципиально отличающейся от картины мира традиционных алгоритмических языков. Из этого противоречия есть всего два выхода:

– либо разработчику придется приспособиться к новому, непривычному языку;

– либо привычный, знакомый разработчику язык придется приспособить к описанию эффективных схем вычислительного характера. Обсудим кратко реализацию обоих подходов.

6.2.1. Приспосабливаем разработчика

Чтобы оценить, насколько трудно может быть разработчику, имеющему только программистскую подготовку, воспользоваться языками схемотехнического проектирования, присмотримся повнимательнее к этим языкам.

Профессиональные схемотехники используют для проектирования цифровых электронных схем, в том числе – вычислительного характера, языки VHDL [18] и Verilog. Разница между этими двумя языками примерно такая же, как между C и Фортраном. При разном синтаксисе и мелких различиях в некоторых конструкциях, в целом оба языка оперируют примерно одним и тем же набором базовых понятий, или, как говорят специалисты по компьютерным языкам, используют одну и ту же программистскую модель. Модель эта далека от традиционной: в ней, например, нет понятия текущей точки исполнения алгоритма, время дискретно и т.п. Далее будем называть традиционную программистскую модель алгоритмической, а модель языков VHDL и Verilog – схемотехнической [19]. Описание схемотехнической модели выходит за рамки настоящей работы.

Бесспорное достоинство схемотехнической программистской модели состоит в том, что ее адекватность целевому железу (точнее, кремнию) доказана многолетней практикой. На языках VHDL и Verilog удается описывать очень эффективные цифровые электронные схемы, все необходимые для этого изобразительные средства в них имеются. Причины, по которым оба этих языка категорически непригодны для использования разработчиками, не имеющими профессиональной подготовки в области схемотехники, можно разделить на две больших группы:

– непривычная программистская модель как таковая;

– плохое качество языков, их низкий уровень в отношении изобразительных средств для проектирования схем именно вычислительного характера.

Важным отрицательным фактором является то, что эти две группы причин диалектически усиливают друг друга. По ряду исторических причин, языки VHDL и Verilog, действительно, довольно плохо и невнятно спроектированы [2,19]. Та программистская модель, которой они, в конечном итоге, следуют на практике, плохо видна в конструкциях языка, зачастую выражена косвенно и неявно. Профессиональные схемотехники успешно

пользуются этими языками потому, что та картина мира, которой соответствует эта программистская модель, им хорошо знакома, причем знакомятся они с ней совершенно отдельно и независимо от освоения языка, а VHDL и/или Verilog изучают только потом. Нашему гипотетическому разработчику схем, который предварительно схемотехнику не изучал, желательно было бы понять ее азы именно через язык, но плохое качество языков, в совокупности с многолетней традицией крайне невнятно и путано их описывать в учебниках, препятствует этому. О том же, чтобы, освоив через язык основы схемотехники, попытаться самостоятельно создать необходимые «надстройки», облегчающие построение именно вычислительных схем, не приходится даже мечтать.

С другой стороны, плохое качество и низкий уровень проблемной ориентации языка трудно признать необходимыми условиями создания эффективных схем. Необходимым условием является использование схемотехнической программистской модели, а ее плохое и низкоуровневое оформление в языке – это недостатки субъективного характера, которые можно и нужно исправить.

Для этого нами была разработана система проектирования схем вычислительного характера Автокод Stream [20]. В ее основе – двухуровневый язык схемотехнического проектирования собственной разработки.

Базовый уровень – язык Автокод – реализует наше представление о схемотехнической программистской модели в несколько упрощенном, но главное – в явном виде. Мы старались, чтобы изучение этого языка помогало формировать схемотехническую картину мира в той части, которая необходима для построения схем вычислительного характера. Автокод транслируется на VHDL.

Проблемно-ориентированный уровень надстроен над базовым, и представлен языком Автокод Stream. На этом уровне базовый язык расширен конструкциями, явно ориентированными на построение длинных синхронных конвейеров со скважностью 1 такт. Конвейеры со всей необходимой логикой синхронизации строятся по записи формул, которые ими реализуются. Автокод Stream транслируется на Автокод.

Базовый уровень языка, все возможности которого доступны также и на проблемно-ориентированном уровне, дает разработчику полный контроль над временной диаграммой создаваемой схемы. Потери эффективности создаваемых схем при переходе

от ручного кодирования на VHDL к Автокоду практически отсутствуют.

К сожалению, примеров использования этой системы проектирования сторонними разработчиками у нас на сегодня нет.

6.2.2. Приспосабливаем язык

Мечта использовать для новых вычислительных архитектур традиционные языки алгоритмической программистской модели так же стара, как сами новые архитектуры, и ни разу пока не была реализована действительно успешно. Проблема – в недостаточной адекватности традиционной программистской модели нетрадиционному железу (кремнию). Для расширения языка в сторону большей адекватности, он снабжается набором специальных прагм. Текст на языке без прагм компилируется в формально работоспособную, но очень неэффективную схему, добавление прагм постепенно улучшает результат компиляции. При этом очень быстро выясняется, что надо не просто добавлять прагмы, а предварительно переписать исходный текст вполне определенным образом, чтобы их (прагмы) было, куда и как добавлять. По существу, разработчику предлагается постепенно освоить схемотехническую картину мира, выражая ее понятия на совершенно новом языке, внешне похожем на старый. С точки зрения схемотехнической картины мира, новый «синтетический» язык ни стройным, ни внятным не назовешь, и этим ситуация очень похожа на описанную в предыдущем подразделе. Разница, однако, в том, что процесс обучения новому языку происходит постепенно, а начинается с языка привычного и хорошо известного. Дополнительное (и очень важное) преимущество состоит в том что, как угодно переделанный и испещренный прагмами исходный текст все же остается правильным текстом на традиционном языке, и его можно отлаживать, как программу. Отладка необходима, поскольку в ходе переписывания исходного текста с целью оптимизации результирующей схемы очень легко внести ошибку.

Такому подходу не откажешь в привлекательности – при условии, что он работает на практике. Мы тщательно проверили его экспериментально на целом ряде модельных задач, используя компилятор из C++ с прагмами в VHDL, разработанный фирмой Xilinx. Компилятор называется Vivado HLS [17]. В последние несколько лет этот компилятор стремительно набирает популяр-

ность среди профессиональных схемотехников как альтернатива ручному кодированию на VHDL. Наши испытания показали, что подход работоспособен, но с несколькими весьма существенными оговорками.

Прежде всего, надежда на самостоятельное построение компилятором эффективной схемы без прагм вообще, или с прагмами, обозначающими указания общего характера, совершено безосновательны. Как неоднократно утверждалось выше, схемы вычислительного характера получаются эффективными тогда и только тогда, когда они реализуют длинный синхронный конвейер. Самостоятельно компилятор таких конвейеров не строит. Разработчику следует выбрать в тексте те циклы, которые хотелось бы конвейеризовать, пометить их соответствующими прагмами, и, внимательно читая диагностические сообщения компилятора, добиваться того, чтобы требуемые конвейеры были построены, причем с разумной скважностью. Наиболее распространенная причина, по которой компилятор не может построить требуемый конвейер – нехватка портов доступа к памяти. Чтобы отреагировать на эту ситуацию адекватно, разработчик должен, как минимум, знать, что это такое, как максимум – уметь ставить себя на место компилятора, чтобы понять, почему портов памяти может не хватать, когда кажется, что их хватает. Часто память требуется расслоить или продублировать, для чего, опять же, очень желательно знать, что такое расслоенная память, и как она используется в схеме. Словом – разработчику необходимо быть схемотехником, а еще – немножко специалистом по компиляторам.

Существует еще целый ряд «маленьких хитростей», которые все объединяет одно общее свойство: чтобы их успешно использовать, надо неплохо разбираться в схемотехнике.

Таким образом, фокус не удался. Мы снова пришли к тому, с чего начинали в прошлом подразделе – к не очень явному и не очень внятному языку, который требует знания схемотехники, но не может помочь это знание приобрести, и к тому же – не очень соответствует целевой программистской модели.

Качество создаваемой компилятором схемы по сравнению с ручным кодированием на VHDL страдает, но не очень сильно (в полтора – два раза или менее) [16], в основном – за счет большой латентности создаваемых конвейеров, а также не всегда обоснованно завышенной их скважности.

Справедливости ради отметим, что есть у этой технологии и бесспорные преимущества. Во-первых, высокоуровневость языка: выписывать сложную логику на C++, действительно, удобно. Особенно в тех случаях, когда речь идет о логике за пределами критического цикла, не нуждающейся в очень уж глубокой оптимизации. Во-вторых, способность компилятора самостоятельно выбрать ту скважность создаваемого конвейера, которую получается обеспечить для данного цикла (или для данной формы записи этого цикла). Иногда бывает, что однотактный конвейер построить не удается, но для приемлемого уровня ускорения достаточно, например, трехтактного. Автокод Stream умеет строить только однотактные конвейеры.

6.2.3. Выводы

В технологиях разработки схем вычислительного характера в последние годы наблюдается несомненный прогресс. Тем не менее, инструментов такой разработки, которыми можно было бы пользоваться без специальной схемотехнической подготовки, или с помощью которых эту подготовку можно было бы приобрести по ходу дела, пока не создано.

Несмотря на то, что проектирование схемы ускорителя, вопреки широко распространенному мнению, не является главной трудностью в практическом применении вычислительных систем на базе FPGA, трудность эта остается весьма и весьма существенной.

Лет 30–35 назад, когда системы команд процессоров общего назначения были простыми, а техника генерации компиляторами объектного кода – далеко не такой совершенной, как сейчас, практиковалась разработка чувствительных к быстродействию программ на двух языках. Сложную, но не критичную по быстродействию, логику писали на языке высокого уровня, те же фрагменты кода, от которых быстродействие зависело в первую очередь, оптимизировали путем переписывания на языке ассемблера.

Похоже, что сегодня эта технология выборочной оптимизации критических по быстродействию фрагментов вручную может найти применение в вычислительной схемотехнике. Обеспечить вставку в схему, описанную на C++ с прагмами и скомпилированную при помощи Vivado HLS, компонента схемы на Автокоде Stream или VHDL, не так легко, как вызвать ассемблерную подпрограмму из программы на Фортране. Схема, в отличие от про-

граммы, работает в каждый момент времени не только в текущей точке выполнения алгоритма, которой в схеме просто нет, а вся целиком. Тем не менее, для высокоуровневой, «крупноблочной» управляющей логики в схемах, генерируемых компилятором Vivado HLS, удается построить аналог «вызова подпрограммы» для компонента схемы, написанного на другом языке. Мы сейчас работаем над включением этой возможности в технологию построения схем.

7. Общие выводы

Технологии вычислительной схемотехники пригодны для очень эффективной реализации довольно ограниченного круга приложений. Проанализировав эти ограничения, можно прийти к выводу, что указанный круг приложений, несмотря на свою ограниченность, все же довольно широк, а среди входящих в него приложений имеются весьма актуальные. Наработанные на сегодняшний день технологии и методики реализации приложений на машинах с FPGA-ускорителями уже пригодны на практике, при условии, что одним из дополнительных критериев отбора приложений будет возможность максимального переиспользования созданного схемного кода. С учетом достигнутого понимания проблематики в целом, частная задача разработки и отладки схем FPGA-ускорителей, действительно, выходит на первый план во всей технологической цепочке. Главной трудностью при этом является то, что необходимый для успешной разработки уровень понимания одновременно алгоритмов и «железа» (кремния) на сегодня не достигается в рамках какой-либо традиционной инженерной специальности. Пока непонятно, можно ли в достаточной степени смягчить последствия этой трудности только путем предоставления разработчику схем более совершенных, чем сегодня, языковых инструментов.

Благодарности

Авторы искренне благодарны А.Е. Луцкому и Н.С. Фиалко за тексты модельных программ и готовность отвечать на глупые вопросы по этим текстам, А.Н. Коршуновой и В.Д. Лахно – за исключительно полезные разъяснения, касающиеся свойств алгоритмов некоторых приложений молекулярной биологии,

И.В. Лихачеву – за активное, заинтересованное сотрудничество и первую в истории нашего коллектива схему FPGA-ускорителя, созданную прикладным программистом. Мы также благодарны С.М. Соколову и А.А. Богуславскому за сотрудничество по модельной реализации вычисления оптического потока, а В.О. Подрыге и С.В. Полякову – за разъяснение некоторых основных понятий методов частиц и молекулярной динамики. Также мы крайне признательны Ю.А. Климову и С.А. Романенко за ценные критические замечания, высказанные при подготовке этого текста. Наша особая благодарность – Б.Я. и Р.Б. Штейнбергам, своими ценнейшими советами вдохновившим нас на поиск подходящих классов приложений, оказавшийся в итоге успешным.

Авторы очень благодарны своим коллегам Г.П. Савельеву и В.Л. Орлову, и в особенности – Ю.П. Смольянову, за постоянную поддержку работы всеми возможными средствами.

Список литературы

1. *А.О. Лацис*, *В.К. Левин*. Перспективы развития суперкомпьютерной техники (по материалам лекции 27.08.2012, Дубна, MPAMCS-2012), Матем. моделирование, 2013, том 25, номер 11, 128–136

2. *А.О. Лацис*. Параллельная обработка данных. – М., «Академия», 2010 г., 336 с. ISBN 978-5-7695-5951-8

3. NVIDIA Accelerated Computing. CUDA Zone. https://developer.nvidia.com/cuda-zone Дата обращения 09.10.2017 г.

4. *С.С. Андреев*, *С.А. Дбар*, *А.О. Лацис*, *Е.А. Плоткина*. О новых архитектурах и новых тестах производительности. Тезисы докладов 20-й всероссийской конференции «Теоретические основы и конструирование численных алгоритмов решения задач математической физики», посвященной памяти К.И. Бабенко. Дюрсо, 2014, с. 17–18.

5. *С.С. Андреев*, *С.А. Дбар*, *А.О. Лацис*, *Е.А. Плоткина*. Почему новые вычислительные архитектуры такие неудобные. Тезисы докладов 21-й всероссийской конференции «Теоретические основы и конструирование численных алгоритмов решения задач математической физики», посвященной памяти К.И. Бабенко. Дюрсо, 2016, с. 17–18.

6. *P. Kogge*. Next-Generation Supercomputers. https://spectrum.ieee.org/computing/hardware/nextgeneration-supercomputers. Дата обращения 09.10.2017 г.

7. All Programmable FPGAs and 3D ICs. https://www.xilinx.com/products/silicon-devices/fpga.html. Дата обращения 09.10.2017 г.

8. ROSTA. Продукты. http://rosta.ru/# Дата обращения 09.10.2017 г.

9. Hybrid Memory Cube Consortium. http://www.hybridmemorycube.org Дата обращения 10.10.2017 г.

10. *R. Barrett*, *M. Berry*, *T. F. Chan*, *J. Demmel*, *J. M. Donato*, *J. Dongarra*, *V. Eijkhout*, *R. Pozo*, *C. Romine*, *H. Van der Vorst*. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. http://www.netlib.org/templates/templates.pdf. Дата обращения 09.10.2017 г.

11. *С.С. Андреев*, *С.А. Дбар*, *А.О. Лацис*, *Е.А. Плоткина*. Какая нам польза от теста HPCG. Тезисы докладов Третьего Национального суперкомпьютерного форума. Переславль-Залесский, 2014. http://2014.nscf.ru/TesisAll/4_Systemnoe_i_promezhytochnoe_PO/06_072_LatsisAO.pdf Дата обращения 10.10.2017 г.

12. *В.О. Подрыга*, *С.В. Поляков*. Молекулярно-динамическое моделирование процесса установления термодинамического равновесия нагретого никеля. Препринт ИПМ им. М.В. Келдыша РАН № 41 за 2014 г.

13. LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov Дата обращения 10.10.2017 г.

14. OpenCV Library. http://opencv.org Дата обращения 10.10.2017 г.

15. Clustal: Multiple Sequence Alignment. http://www.clustal.org Дата обращения 10.10.2017 г.

16. *С.С. Андреев*, *С.А. Дбар*, *А.О. Лацис*, *Е.А. Плоткина*. Инженерная методика адаптации приложений к гибридному кластеру с ускорителями на ПЛИС. http://www.kiam.ru/MVS/research/fpga/ingmet/ Дата обращения 10.10.2017 г.

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

18. *В.П. Бабак*, *А.Г. Корченко*, *Н.П. Тимошенко*, *С.Ф. Филоненко*. VHDL: справочное пособие по основам языка. М., Издательский дом «Додэка-XXI», 2008г. – 224с. ISBN 978-5-94120-169-3.

19. *С.А. Дбар*, *А.О. Лацис*. О модели программирования вычислительной схемотехники. Языки программирования и компиляторы – 2017: труды конференции. Южный федеральный университет; под ред. Д.В. Дуброва. – Ростов-на-Дону: издательство Южного федерального университета, 2017. ISBN 978-5-9275-2349-8. с. 98–100.

20. *С.С. Андреев*, *С.А. Дбар*, *А.О. Лацис*, *Е.А. Плоткина*. Гибридный реконфигурируемый вычислитель. http://www.kiam.ru/MVS/research/fpga/index.html Дата обращения 10.10.2017 г.

21. *А.Б. Шворин*. Эмулятор PCI Express для HDL-моделирования. Вестник Южно-Уральского государственного университета. Том 3, № 4 (2014), с.51–60. http://vestnik.susu.ru/cmi/article/download/2737/2634 Дата обращения 11.10.2017 г.

Содержание

Отделение математических наук

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

Как и почему могут быть использованы на практике суперкомпьютеры на базе FPGA

Формат 60 x 84/16
Гарнитура Таймс
Усл. печ. л. 2,3. Усл. изд. л. 1,8
Тираж 20 экз.

Издатель – Российская академия наук

Подготовлено к печати
Управлением научно-издательской деятельности РАН

Отпечатано на оборудовании Управления делами РАН

Издано в авторской редакции

Издается в соответствии с распоряжением
президиума Российской академии наук
от 24 октября 2017 г. №10106-765,
распространяется бесплатно.