ПОЧЕМУ ДО СИХ ПОР НЕТ ХОРОШЕЙ СИСТЕМЫ ПРОГРАММИРОВАНИЯ ДЛЯ ГИБРИДНОГО СУПЕРКОМПЬЮТЕРА С FPGA-УСКОРИТЕЛЯМИ.

С.С. Андреев[1], С.А. Дбар[1], **А.О. Лацис[1]**, Е.А. Плоткина[1]

[1]Институт прикладной математики им. М. В. Келдыша РАН, lacis@kiam.ru

С появлением GPGPU и гибридных суперкомпьютеров на их основе, задача создания приемлемой системы программирования для таких машин была решена в очень короткий срок. Для гибридных машин с FPGA-ускорителями ничего подобного так и не создано, хотя первые образцы таких машин появились раньше, чем суперкомпьютеры на базе GPGPU. Это может означать одно из двух: либо машины с FPGA-ускорителями никому не нужны, либо на пути создания систем программирования для них встретились серьезные трудности системного характера. Обе возможности представляются интересными для исследования.

1. Нужны ли вообще суперкомпьютеры с FPGA-ускорителями?

То, что универсальному процессору в качестве единственного вычислительного устройства суперкомпьютера необходима альтернатива, сегодня практически не оспаривается. Последние несколько лет показали также, что использование в качестве такой альтернативы GPGPU недостаточно, нужны еще гораздо более эффективные (в любой разумной метрике) вычислительные устройства. Можно ли рассчитывать на появление в обозримой перспективе неких «GPGPU-2», которые решат, наконец, все проблемы, и закроют тему? Для ответа на этот вопрос вспомним, в чем именно заключается недостаточность эффективности универсального процессора.

Главный недостаток универсального процессора (он же – главный источник будущих усовершенствований) заключается в крайне неэффективности «логистики», то есть системы передачи данных между функциональными устройствами, как в системе процессор-память, так и в самом процессоре. Других серьезных источников повышения эффективности вычислителя, кроме совершенствования его «логистики», сегодня не существует. Появление GPGPU было важным шагом именно в этом направлении. Шаг этот, как мы сейчас понимаем, оказался недостаточным. Гораздо более радикальное улучшение «внутренней логистики», чем в GPGPU, возможно только путем адаптации этой «логистики» к конкретному алгоритму, то есть путем использования реконфигурируемых массивов оборудования для построения процессоров одной задачи. Другого материала, кроме FPGA, для создания таких процессоров одной задачи сегодня не существует. Возможно, когда-нибудь появятся некие специальные «FPGA для вычислений», но это, если и произойдет, то не раньше, чем будут достигнуты серьезные продвижения в вычислительной схемотехнике на базе стандартных FPGA.

Созревшие на сегодня потребности в совершенствовании суперкомпьютерного вычислителя настолько же серьезны, насколько далеки от удовлетворения. В этих условиях мы просто не можем игнорировать такой мощнейший источник улучшения системы передачи данных внутри вычислителя, как переход от процессоров с программным управлением к процессорам одной задачи, для чего технически необходимы FPGA. Тем более, что альтернативных подходов сравнимой мощности пока, похоже, не просматривается. Иными словами, гибридные суперкомпьютеры с FPGA-ускорителями объективно нужны, даже если на первый взгляд и кажется, что это не так.

2. Анатомия трудностей.

С упрощенной, наивной точки зрения, проблема создания системы программирования сводится к реализации языка программирования для ускорителя. По крайней мере, такой вывод напрашивается, если посмотреть на систему программирования CUDA, например. В нашем случае, однако, проблемы начинаются гораздо раньше. Строго говоря, первая из проблем, которые нам предстоит рассмотреть, относится даже не к созданию системы программирования, а к применению суперкомпьютеров с FPGA-ускорителями вообще.

2.1. Проблема мелко-блочных методов.

Серьезные продвижения в улучшении «внутренней логистики» возможны внутри одной микросхемы FPGA, поскольку связность элементов оборудования в пределах микросхемы беспрецедентна, и не сравнима ни с какими системами передачи данных вроде канала «процессор-память». Ценой перехода в мир высокой связности, то есть сотен устройств памяти, непосредственно соединенных с десятками вычислителей для их безостановочного «питания» данными, является накладной расход на копирование данных из универсального процессора в сопроцессор и обратно. Расход этот должен окупаться длительной внутренней обработкой. Это – азбука, приобретающая в случае FPGA совершенно экстремальный характер. Ведь суммарный объем внутренней памяти одной FPGA – в лучшем случае 8-16 мегабайт, а зачастую всего 1-2 мегабайта. Чтобы лучше осознать, насколько это мало, представим себе GPGPU, в составе которого вообще нет глобальной памяти – только разделяемая память и регистры. Понятно, что подобрать численный метод для ускорения на такой машине будет нелегко.

Тем самым, формальное создание системы программирования, позволяющей разбить программу на "процессорную" и "ускорительную" части, само по себе почти бессмысленно. Использование такой системы программирования молчаливо предполагает, что программы, ранее успешно работавшие на суперкомпьютере из универсальных процессоров, могут быть перенесены в новую среду почти механически, с минимальными структурными изменениями. Это неявное (и неверное) допущение уже не раз сыграло с программистами злую шутку при переносе программ на суперкомпьютеры с GPGPU. В нашем же случае структурные изменения в программе еще более обязательны, должны быть еще более глубокими, должны существенно затронуть "процессорную" часть, и не факт, что они вообще приведут к успеху. Получается, что, еще до постановки задачи создания гибридной системы программирования, необходим глубокий и кропотливый анализ численных методов на предмет их пригодности для переноса в новую среду, разработка способов и инженерных методик такого переноса. Работа эта, хотя и нацелена на будущий перенос программ на машины с FPGA-ускорителями, сама по себе не предполагает никакой работы с FPGA, выполняется с методом, в терминах метода, на традиционных языках программирования.

Интересно отметить, что важность локализации вычислений в численных методах - идея далеко не новая, впервые была выдвинута всерьез еще в начале 90-х, с появлением в составе универсальных процессоров кэш-памяти заметного объема. Однако, адаптация алгоритмов к эффективному использованию кэш-памяти, в отличие от адаптации для будущего переноса в FPGA, имеет пошаговый и аддитивный характер. Программу можно частично приспособить к эффективному использованию кэш, и она частично ускорится. Адаптация к переносу в FPGA такими свойствами не обладает: программу приходится трансформировать либо полностью, либо никак. Здесь просматривается очевидная аналогия с "распараллеливанием" программ для MPI-машин: программу невозможно "распараллелить" приблизительно и частично.

Пока мы провели самые предварительные исследования пригодности к мелко-блочной реализации для основных классов методов вычислительной линейной алгебры.

2.1. Проблемы языка описания схемы.

Предположим, что проблема мелко-блочности численного метода как-то решена, и сосредоточимся на выборе языка описания схемы создаваемого процессора одной задачи (он же - сопроцессор-ускоритель). Самая привлекательная возможность - реализовать трансляцию в описание схемы (например, на язык VHDL, применяемый для описания схем профессиональными схемотехниками) привычного программистам алгоритмического языка. Такие трансляторы существуют. Вопрос в том, какого качества схема в принципе может быть получена из программы на языке Си, Фортран или им подобном. Здесь опять очевидна аналогия с так желанным многими программистами, но так и не осуществленным за много лет "автоматическим распараллеливанием" изначально последовательных программ. В самой трансляции Си в VHDL нет не только ничего невозможного, но и ничего сложного. Проблема только в том, что Си - алгоритмический язык, подходящий для универсального процессора именно потому, что универсальный процессор - алгоритмическая по своей природе машина. Из программы на языке Си легко получить схему, исполняющую алгоритм, то есть работающую примерно так же, как работал бы универсальный процессор, выполняя эту программу. Но такая схема, выполняя арифметические операции одну за другой по мере поступления, записывая каждый промежуточный результат в память, и т.п., заведомо обладает всеми недостатками процессора в смысле "логистики", от которых мы как раз хотим уйти. Эффективные в смысле "логистики" схемы, способные обогнать процессор в некоторой разумной метрике сравнения, работают не по алгоритмическому, а по конвейерному принципу. Традиционные алгоритмические языки не предназначены для построения длинных синхронных конвейеров, и ни из чего не следует, что построение их в автоматическом режиме, исходя из текста на Си, более реалистично, чем пресловутое "автоматическое распараллеливание", которое на заре эпохи параллельных суперкомпьютеров казалось таким близким, но так толком и не получилось до сих пор.

Используемые профессиональными схемотехниками языки описания схем VHDL и Verilog лишены указанных недостатков, в высокой степени адекватны целевому "железу", но крайне скверно спроектированы, и притом не содержат средств высокоуровневого описания длинных вычислительных конвейеров. Это делает прямое использование их прикладными программистами совершенно не реалистичным.

Сегодня мы опробовали или собираемся опробовать три подхода к описанию схем сопроцессора-ускорителя, хоть в каком-то приближении потенциально пригодных если не для самостоятельного использования, то хотя бы для понимания прикладными программистами.

1). "VHDL с человеческим лицом". Взяв от VHDL и Verilog главное, что делает их пригодными для описания хороших схем - их своеобразную модель программирования - можно попытаться построить более простой, и более ориентированный на вычислительные приложения, язык, и написать транслятор с этого языка на VHDL. Мы такой язык - Автокод Stream - придумали и реализовали. Язык автоматизирует построение длинных арифметических конвейеров, качество получаемых на выходе схем не отличается от качества ручного кодирования на VHDL.

2). Алгоритмический язык, обогащенный прагмами для учета особенностей целевого "железа", в частности, для автоматизированного превращения циклов в длинные конвейеры. Сравнительно недавно фирма Xilinx выпустила компилятор Vivado HLS, реализующий именно этот подход на очень высоком уровне качества. Схемы на выходе получаются не настолько хорошими, насколько можно было бы ожидать при кодировании на VHDL, но уровень автоматизации при построении схем очень высок. Вполне удовлетворительное быстродействие часто удается получить за счет несколько большего, чем при использовании ручного кодирования на VHDL, расхода площади кристалла.

3). Использование языка некоторой третьей модели программирования, в надежде, что его автоматическая трансляция позволит получать на выходе хорошие схемы. Типичный пример реализации этого подхода - система Mitrion C фирмы Mitrionics. Язык Mitrion C имеет Си-подобный синтаксис, но является не алгоритмическим, а функциональным языком. Мы этот язык пока не пробовали, но, по отзывам, система вполне работоспособна.

2.2. Проблемы реализации схемы в кремнии.

Схема, полученная из текста на VHDL, изготовленного либо вручную, либо с применением одного из трех упомянутых выше подходов, представляет собой граф, вершины которого - очень мелкие, элементарные устройства обработки информации, а дуги - связи между ними. При физической реализации этот граф необходимо отобразить на реальную геометрию микросхемы FPGA, как правило - на плоскую решетку. При этом действуют очень жесткие ограничения на длины связей, получающихся из дуг графа. Эти ограничения заданы рабочей частотой схемы, при их нарушении схема не сможет работать правильно. Этот процесс отображения графа схемы на реальную геометрию FPGA называется трассировкой. В современных САПР он выполняется автоматически, что, собственно, и превращает работу схемотехника в разновидность работы программиста. Подобно программисту, схемотехник, описывающий схему на VHDL, не заботится о физике ее реализации - за физическую корректность "отвечает" трассировщик, программа в составе САПР. Поскольку трассировщик - это автомат, выдающий результат "под ключ", о проблемах, связанных с трассировкой, при построении системы программирования для гибридных FPGA-суперкомпьютеров, часто забывают. К сожалению, проблемы эти не только очень серьезны, но имеют системный, а не технический, характер. Их решение может потребовать изменений в высокоуровневом языке описания схемы, и даже в модели программирования, реализуемой этим языком.

Микросхемы FPGA в последние годы, подобно универсальным процессорам, стабилизировались по рабочей частоте, но быстро растут в размерах, то есть увеличивается количество элементарных устройств обработки информации, которые можно разместить на кристалле (далее будем понимать слова "размер микросхемы" именно в этом смысле, а не в смысле ее размера в миллиметрах). По мере роста размеров микросхем и, соответственно, роста сложности схем, размещаемых в них, проблемы с трассировкой обостряются. Трассировка сложной схемы, занимающей 70-80% современной FPGA большой емкости, может занимать часы, а то и десятки часов. Уже одно это - серьезный негативный фактор, который нельзя не учитывать в практической работе. Но гораздо хуже то, что с ростом сложности схемы и размера кристалла, в который ее предполагается поместить, резко растет вероятность того, что трассировка вообще закончится неудачей, то есть отобразить схему на реальную геометрию кристалла не получится. При этом, очень часто проблема не в суммарном числе вершин графа, а в его высокой связности, препятствующей отображению его на плоскость с учетом ограничений на длины связей. При попытке реализации в FPGA сопроцессоров для реальных вычислительных задач с использованием технологии №2, мы встречали ситуации, когда даже при заполнении кристалла менее чем на 20% требуемый граф связей категорически не удавалось отобразить на реальную геометрию.

На первый взгляд, проблема имеет чисто технический характер, и вообще не должна обсуждаться в одном ряду с такими "философскими" вопросами, как свойства модели программирования и/или численных методов. В самом деле, почему бы не сделать оптимистичное допущение, что в следующих версиях схемотехнических САПР программы трассировки будут усовершенствованы, и то, что сегодня не получилось за сутки, завтра начнет получаться за пару часов? Рискнем утверждать, что отнесение данной проблемы к чисто "техническим", по крайней мере, не вполне верно.

Легко видеть, что проблема слишком плотного для трассировки в реальную геометрию графа связей - это проблема той самой "логистики", с необходимости совершенствования которой мы начали, только повернутая под немного другим углом зрения. В самом деле, слишком плотный граф означает всего лишь, что мы потребовали создать систему доставки данных, не осуществимую технически, поскольку в ней, грубо говоря, почти все устройства обработки связаны почти со всеми непосредственно. Произошло это по той простой причине, что мы уже при описании схемы проигнорировали возможные ограничения на связность, поскольку в нашем языке, в его модели программирования, просто нет понятий, позволяющих эти ограничения как-то учесть. Очевидно, эти понятия надо найти и ввести в язык, но это как раз и означает модификацию, или даже замену, самой модели программирования.

Интересно сравнить в этом отношении упомянутые выше подходы к языкам описания схемы сопроцессора-ускорителя. Данных для сравнения по третьему подходу (функциональный язык Mitrion C) у нас нет, так что ограничимся сравнением подхода №1 ("VHDL с человеческим лицом") и подхода №2 (Си с прагмами).

В рамках обоих подходов создаваемая схема молчаливо предполагается однородной и синхронной, то есть на все ее цепи при реализации в кремнии накладываются одинаковые частотные ограничения. Это несколько парадоксально само по себе, поскольку длинные синхронные конвейеры, из которых преимущественно состоят действительно эффективные схемы, допускают в принципе наличие с своем составе длинных цепей, не успевающих срабатывать за один такт.

Когда профессиональный схемотехник строит схему (не обязательно вычислительного характера) вручную (на VHDL), и трассировка этой схемы не удается, ее крайне трудоемкая и кропотливая ручная доработка часто как раз и заключается в искусственном разбиении цепей на более короткие звенья, путем установки между звеньями буферных регистров, не предусмотренных первоначальной логикой. Программа трассировки, как правило, не может выполнить эту работу автоматически, поскольку для этого необходимо понимать (и иногда немного менять) сам смысл работы схемы.

РАЗДЕЛ БУДЕТ ДОПИСАН В БЛИЖАЙШЕЕ ВРЕМЯ