АВТОМАТИЧЕСКАЯ ГЕНЕРАЦИЯ ПРОГРАММ ДЛЯ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ ПО НЕПРОЦЕДУРНЫМ СПЕЦИФИКАЦИЯМ

А.Н. Андрианов, А.Б. Бугеря, Е.Н. Гладкова, К.Н. Ефимкин, П.И. Колударов

Введение. В современном научном сообществе задача разработки эффективных параллельных программ имеет важное стратегическое значение. Наверное, уже не осталось ни одной области науки или отросли промышленности, так или иначе не связанной со сферой высокопроизводительных вычислений. Несмотря на то, что параллельное программирование появилось уже достаточно давно, успешно развивается и проводится масса исследований на эту тему, вопрос «как создать эффективную параллельную программу для решения такой-то задачи» до сих пор крайне актуален для программистов и математиков.

Достаточно быстрое развитие новых аппаратных возможностей для поддержки параллельных вычислений, наблюдаемое в последнее время, еще больше усложняет проблему. Например, появление массово доступных многоядерных процессоров поставило вопрос об эффективном программировании для них. Практически одновременно появились массово доступные графические ускорители (графические процессоры), и опять возник вопрос об эффективном программировании для них. Агрессивное продвижение своих решений фирмами-производителями вычислительных систем, обладающих этими возможностями, часто дезориентирует прикладных специалистов, разрабатывающих параллельные вычислительные программы, толкает их на изменение средств разработки программ, хотя ясные и достаточно убедительные аргументы в пользу таких изменений отсутствуют. Так, применение технологии CUDA для эффективного программирования для графических ускорителей в первых своих версиях являлось, фактически, программированием на уровне ассемблера, с учетом тонких особенностей аппаратуры.

С помощью такого ручного низкоуровневого программирования за последние годы некоторые расчётные прикладные пакеты и математические библиотеки были портированы для использования на вычислительных системах с графическими процессорами [1]. Это, несомненно, существенно облегчает задачу прикладному специалисту, но только в том случае, если всего его потребности в высокопроизводительных вычислениях покрываются имеющимися распараллеленным пакетом и/или библиотеками. Если же с помощью таких готовых средств построить решение для своей задачи не удаётся, то иного пути, кроме как изучать CUDA или ATI Stream (или OpenCL – это уже более высокий уровень абстракции, но и эффективность его реализации пока далека от желаемого уровня) и начать реализовывать свой алгоритм на столь низком уровне, у прикладного специалиста нет.

Надежды на автоматическое распараллеливание уже написанных последовательных программ на графические процессоры пока совершенно не оправдываются, несмотря на то, что фирмы-производители графических процессоров, как NVIDIA, так и AMD, активно поддерживают данное направление исследований. Из уже реализованных подходов можно отметить те, которые базируются на вполне разумном симбиозе распараллеливающего компилятора и подсказок со стороны программиста, выполненных в виде специальных программных директив, например OpenACC [2]. Но, к сожалению, производительность получаемого таким путём программного кода далека от производительности программ, написанных ручным программированием для целевой платформы, и поэтому пока не может быть признанной удовлетворительной.

Работы по созданию и продвижению новых средств и языков программирования для графических процессоров, гибридных решений и различных нетрадиционных вычислительных архитектур (например, FPGA), ведутся весьма активно [3, 4], однако проблема простой разработки параллельных программ и утилизации новых возможностей вычислительной техники так и остается в настоящее время не решенной.

Декларативный подход. Язык Норма. Один из возможных подходов к решению задачи автоматизации параллельного программирования вычислительных задач, и, в частности, задачи автоматического построения эффективной программы для графических процессоров, является подход с использованием декларативных (непроцедурных) языков. При использовании этого подхода прикладной специалист программирует решение вычислительной задачи на непроцедурном языке (понятия, связанные с архитектурой параллельного компьютера, моделями параллелизма и т.п. при этом не используются), а затем компилятор автоматически строит параллельную программу (учитывая архитектуру целевого параллельного компьютера, модели параллелизма и т.п.). С учетом отмеченных выше проблем привлекательность этого подхода в настоящее время только усиливается и интерес к идеям непроцедурного декларативного программирования и реализации этих идей в языках программирования неуклонно растет.

Идеи декларативного программирования были сформулированы еще в прошлом веке, теоретические исследования этого подхода для класса вычислительных задач проведены в пионерских работах И.Б.Задыхайло еще в 1963 году [5]. Непроцедурный язык Норма и система программирования Норма [6-8] разработаны в ИПМ им. М.В. Келдыша РАН также достаточно давно и предназначены для автоматизации решения вычислительных сеточных задач на параллельных компьютерах. Расчетные формулы записываются на языке Норма в математическом, привычном для прикладного специалиста виде. Язык Норма позволяет описывать решение широкого класса задач математической физики. Программа на языке Норма имеет очень высокий уровень абстракции и отражает метод решения, а не его реализацию при конкретных условиях. Такое описание не ориентировано на конкретную архитектуру компьютера, поэтому оно предоставляет большие возможности для выявления естественного параллелизма и организации вычислений.

В настоящее время в ИПМ им. М.В. Келдыша ведутся работы по созданию версии компилятора программ на языке Норма+, который на выходе создаёт исполняемую программу для графических процессоров фирмы NVIDIA с использованием технологии CUDA.

Принципы построения CUDA программы по декларативным описаниям. При трансляции с языка Норма+ решается задача синтеза выходной параллельной программы, то есть выходная параллельная программа строится автоматически. В результате анализа зависимостей по данным между операторами программы на языке Норма+, в случае разрешимости этих зависимостей, представляется возможным построить так называемую «параллельную ярусную схему» выполнения программы. На каждом ярусе данной ярусной схемы располагаются операторы программы, которые не имеют зависимостей друг от друга и могут выполняться независимо и, соответственно, параллельно. В то же время каждый из этих операторов имеет зависимость от одного или более операторов, располагающихся на предыдущем ярусе ярусной схемы. Таким образом, группу операторов, располагающихся на одном уровне ярусной схемы, можно выполнять параллельно в результирующей программе, но только после того, как будут полностью выполнены все операторы предыдущего уровня ярусной схемы. Параллельная ярусная схема программы является, фактически, представлением идеального (естественного) параллелизма, определяемого соотношениями и зависимостями между расчетными переменными программы.

В результате ряда исследований по трансформации описанной параллельной ярусной схемы программы на языке Норма+ в программу с использованием технологии NVIDIA CUDA для графических процессоров авторами предлагается использовать следующий подход для автоматического построения исполняемой программы для графических процессоров с использованием технологии CUDA.

Исполняемая программа стартует и завершается на центральном процессоре, на нём же выполняется ввод-вывод данных и итерационные циклы. Очевидно также, что вся логика по управлению вычислениями, выделением памяти на графическом процессоре, организация обменов данными между памятью графического и центрального процессора, тоже создаётся в программном коде, выполняющемся на центральном процессоре. Сами вычислительные операторы выполняются на графическом процессоре. Для этого они группируются в определённые наборы, каждый из которых может выполняться в пределах одного ядра программы с использованием технологии NVIDIA CUDA. Программа, выполняющаяся на центральном процессоре, осуществляет контроль за общим выполнением программы, запускает полученные ядра в определённом порядке, ведёт итерационные циклы и проверяет условие выхода из итерации. На рис.1 приведён пример общей схемы выполнения исполняемой программы.

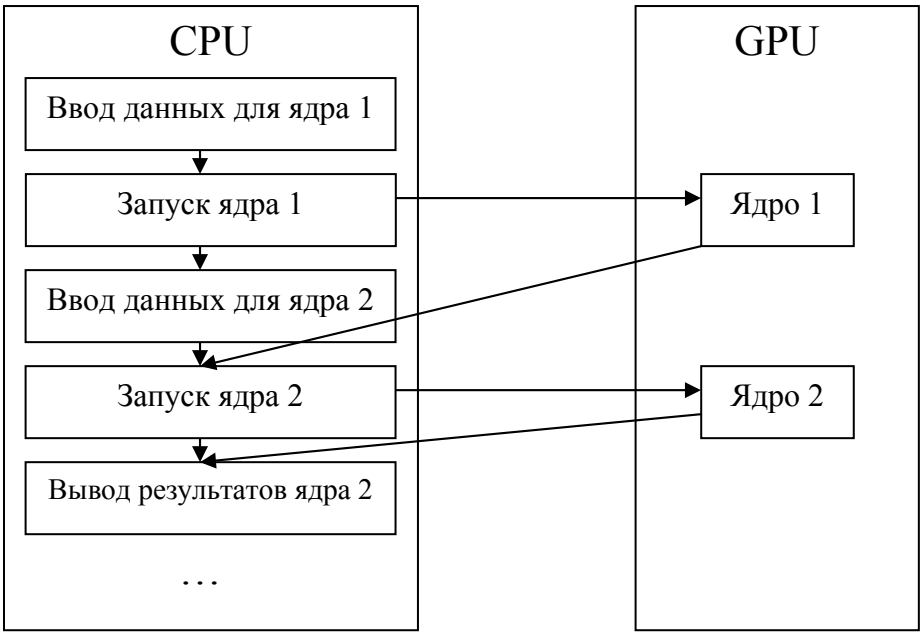

Рис.1 Пример общей схемы выполнения исполняемой программы.

Один из ключевых моментов в этой схеме организации исполняемой программы, и он же алгоритмически самый тяжёлый – это задача группировки вычислительных операторов в ядра, которые будут выполняться на графическом процессоре. С одной стороны, чем больше операторов будет содержать такое ядро, т.е. чем крупнее будут исполняемые ядра в программе, тем эффективнее она будет работать, т.к. будет меньше запусков ядер, меньше передачи параметров, в том числе и через память центрального процессора, и т.п. Но, с другой стороны, все операторы, сгруппированные в одно ядро, должны работать с одним и тем же распределением рабочих областей на конфигурацию блоков и нитей, определяемую при запуске ядра. И может оказаться, что поиск такого распределения для большой группы операторов, объединённых в одно ядро, может дать худший результат, чем разбиение этой группы операторов на более мелкие отдельные ядра и поиск распределения для каждой группы в отдельности. Кроме того, при поиске таких группировок необходимо

соблюдать ряд обязательных условий, накладываемых ярусной схемой и сущностью группируемых операторов. Например, мы можем менять порядок выполнения операторов, но только в пределах одного яруса. А операторы ввода-вывода должны выполняться на центральном процессоре.

В итоге, после ряда исследований характеристик получающихся программ, предлагается использовать следующий набор эвристик и обязательных условий для решения задачи группировки операторов программы на языке Норма+ в один набор, который может выполняться в пределах одного ядра программы с использованием технологии NVIDIA CUDA:

1. Каждая нить графического процессора осуществляет вычисления в одной точке расчетной области Норма-программы (другими словами, одной точке индексного пространства вычисления).
2. Поскольку операторы, находящиеся на одном ярусе параллельной ярусной схемы, можно выполнять параллельно, они могут выполняться одним ядром CUDA программы, причём в любом порядке. Поэтому в очередном ярусе выбираются операторы, которые выполняются на одной и той же одномерной, двумерной или трёхмерной подобласти. Вообще говоря, оператор может выполняться и на области большей размерности, но выбранная для распараллеливания подобласть должна присутствовать в области выполнения оператора. Эти выбранные операторы могут выполняться одним ядром, при условии соблюдения описанных ниже условий.
3. Выбранная для распараллеливания подобласть разбивается на блоки и нити для ядра графического процессора следующим образом:

- если подобласть одномерная, то за число нитей берётся предопределённое число, кратное степени двойки (1024 по умолчанию) и равное максимальному количеству нитей в блоке на целевой архитектуре. А для блоков выбирается одномерный массив с учётом ограничений целевой архитектуры на размер массива блоков по одному направлению или двумерный массив таким образом, чтобы произведение количества блоков на количество нитей полностью покрывало выбранную подобласть;
- Пусть D(N) – выбранная для распараллеливания подобласть из N точек. Тогда распределение выглядит так:

```
D(N), (((N + 1023)/1024) <= 65535) =>
      kernel<<< ((N + 1023)/1024), 1024 >>>()
D(N), (((N + 1023)/1024) > 65535) =>
      kernel<<< dim3((((N + 1023)/1024 + 65534)/65535), 65535), 1024 >>>()
```

- если подобласть двумерная, и одно из направлений меньше или равно максимальному количеству нитей в блоке на целевой архитектуре, то этому направлению будут соответствовать нити, количество которых определяется как ближайшая сверху степень двойки. А для второго направления подобласти выбирается одномерный массив блоков, равный размеру этого направления, с учётом ограничений целевой архитектуры на размер массивов блоков по одному направлению, или подбирается двумерный массив блоков таким образом, чтобы общее количество блоков полностью покрывало второе направления подобласти.

```
D(N1, N2), (N2 <= 1024), (N1 <= 65535) =>
      kernel<<< N1, N2 >>>()
D(N1, N2), (N2 <= 1024), (N1 > 65535) =>
      kernel<<< dim3((N1 + 65534)/65535, 65535), N2 >>>()
```

- если подобласть двумерная, и ни одно из направлений не может быть представлено нитями в блоке на целевой архитектуре, то одному из направлений назначается двумерный массив, одно измерение которого – нити, количество которых определяется как предопределённое число, кратное степени двойки (1024 по умолчанию) и равное максимальному количеству нитей в блоке на целевой архитектуре. А второе измерение этого двумерного массива выбирается из одного из направлений в двумерном массиве блоков, таким образом, чтобы произведение количества блоков по данному направлению на количество нитей полностью покрывало рассматриваемое направление подобласти. Второму направлению подобласти ставится в соответствие второе направление в двумерном массиве блоков.

```
D(N1, N2), (N2 > 1024), (N1 <= 65535) =>
      kernel<<< dim3(N1, (N1 + 1023)/1024), 1024 >>>()
```

- если подобласть трёхмерная, и одно из направлений меньше или равно максимальному количеству нитей в блоке на целевой архитектуре (а остальные направления меньши или равны размеру блока), то этому направлению будут соответствовать нити количеством ближайшая сверху степень двойки. А для остальных направлений подобласти ставится в соответствие двумерный массив блоков соответствующей размерности.

```
D(N1, N2, N3), (N3 <= 1024), (N1, N2 <= 65535) =>
      kernel<<< dim3(N1, N2), N3 >>>()
```

- Если оператор является оператором ввода-вывода данных, то он должен выполняться на центральном процессоре, и он может выполняться параллельно с запуском ядра. При этом дальнейшее добавление в формируемое ядро операторов со следующего уровня (п.6) становится невозможным.
- Если в операторе присутствует функция редукции, и область применения данной функции редукции полностью представлена нитями по какому-то индексу, а все остальные индексы из области применения функции редукции (если они есть) не входят в область, распараллеленную в текущем ядре, то этот оператор может быть выполнен ядром без исключений. В противном же случае, если область

применения функции редукции имеет индексы, которым соответствует какое-либо направление массива блоков, данное ядро может выполнить функцию редукции только частично, и затем выполнение ядра должно быть прекращено и запущено следующее ядро (или даже последовательно несколько ядер) с другим распределением – так, чтобы в итоге все частичные результаты выполнения функции редукции были представлены исключительно нитями и было возможно получить окончательный результат. Такой оператор должен быть поставлен последним в ядре, и дальнейшее добавление в формируемое ядро операторов со следующего уровня (п.6) становится невозможным.

- После завершения формирования последнего вычислительного ядра с операторами данного слоя ярусной схемы можно попробовать добавить в это ядро операторы со следующего слоя ярусной схемы при соблюдении двух описанных ниже условий. Это возможно, т.к. ядро выполняет операторы, содержащиеся в нём, последовательно в отдельно взятой точке распараллеленной области. Если зависимости между операторами разных уровней параллельной ярусной схемы простые, без смещения по какому-либо индексу из области распараллеливания (например X = …; R = func(X)), то последовательно ваполненные действия в каждой точке области дадут правильный результат. И не важно, скажем, что при вычислении R в точке k значение X в точке k+1 ещё не было вычислено другой нитью – важно только, что значение X в точке k уже вычислено. Затем, после завершения очередного уровня параллельной ярусной схемы, можно начинать включать в текущее ядро операторы со следующего уровня и т.п., пока процесс добавления не закончен по одной из причин, описанной в п. 4-6. Добавление таких операторов возможно при соблюдении следующих условий:
- если на добавляемом слое есть операторы ввода-вывода, они должны быть выполнены на центральном процессоре после завершения выполнения создаваемого ядра, формирование которого должно быть закончено текущим слоем;
- в добавляемых операторах не должно быть использования переменных с индексами из подобласти распределения со смещением (например R = func(X[i+1])). Если такие операторы есть, то они не должны быть добавлены в создаваемое ядро, формирование которого должно быть закончено текущим слоем. С такими оставшимися операторами можно начать формировать следующее ядро.

Пример построения программы с использованием CUDA. В качестве примера рассмотрим применение описанной выше схемы для простого фрагмента программы на языке Норма+:

```
Oij:(Oj:(j=1..w);Oi:(i=1..v)).
VARIABLE Vij DEFINED ON Oij DOUBLE.
VARIABLE Vsum DEFINED ON Oi DOUBLE.
DOMAIN PARAMETERS v=2000,w=3000.
FOR Oij ASSUME Vij = j+(i-1)*w.
FOR Oi ASSUME Vsum = SUM((Oj)Vij).
```

В приведённом фрагменте величине Vij, определённой на двумерной области Oij с индексами i и j, присваиваются начальные значения j+(i-1)*w, а затем производится суммирование по индексу j (по области Oj). В данном случае распараллеливается вся область Oij. По индексу j область распределяется на нити и на одну из размерностей блоков, а по индексу i – на вторую размерность блоков (3-й подпункт пункта 3 приведённой выше схемы). Начало оператора суммирования выполняется в первом же ядре, но, т.к. всё направление индекса j не покрывается полностью нитями, необходимо создание второго ядра, завершающего суммирование.

Схема распределения области Oij на блоки и нити приведена на рис.2. Жирными линиями выделены блоки, их конфигурация 2000 (размер по индексу i) на 3 (дополнительное направление по индексу j, которое в объединении с 1024 нитями в каждом блоке обеспечивает представление 3000 точек по индексу j). Нити в каждом блоке показаны пунктиром.

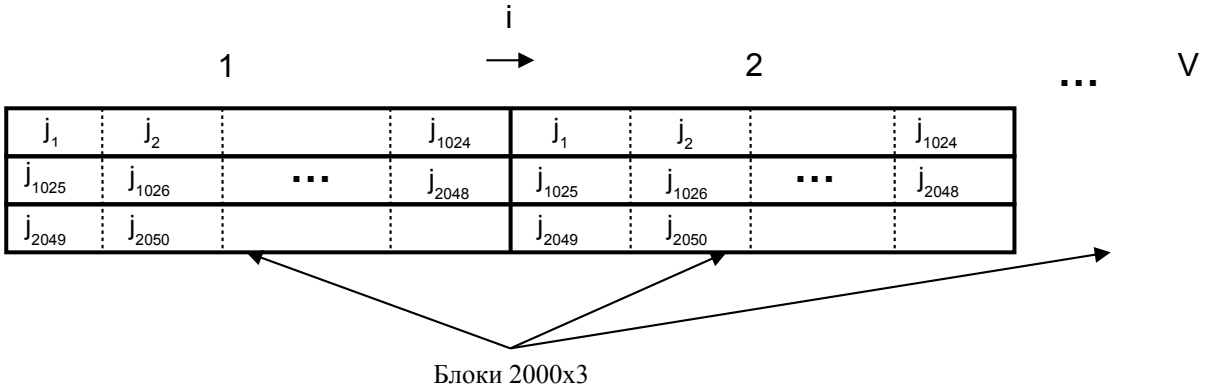

Рис. 2. Отображение области Oij на блоки и нити.

```
#define v 2000
#define w 3000
#define wthreadK1 1024 // nearest pow2 from w
#define wBlockYK1 (w+wthreadK1-1)/wthreadK1
#define wthreadK1Red 4 // nearest pow2 from wBlockYK1

__global__ void SummaK1() {
    __shared__ double shared[wthreadK1];
    int j = threadIdx.x + blockIdx.y*wthreadK1; // Получаем индекс j
    if(j < w) { // Проверяем что не вышли за границу области
        int i = blockIdx.x;
        int gidx = i*w + j;
        Vij_dev[gidx] = j + 1 + i * w; // Присваиваем начальные значения
        // Начинаем суммирование
        shared[threadIdx.x] = Vij_dev[gidx];
        for (int d = wthreadK1/2; d > 0; d /= 2) {
            int from = threadIdx.x + d;
            __syncthreads();
            if ((threadIdx.x < d) && (from + blockIdx.y*wthreadK1 < w))
                shared[threadIdx.x] += shared[from];
        }
        // Сохраняем частичный результат.
        if (threadIdx.x == 0)
            Vsum_block[blockIdx.x*wBlockYK1+blockIdx.y] = shared[0];
    }
}

__global__ void SummaK1Red() {
// Осуществляет суммирование Vsum_block
        …….
}
```

И фрагмент программы, выполняющейся на центральном процессоре, выглядит следующим образом:

```
SummaK1<<< dim3(v, wBlockYK1), wthreadK1 >>>();
SummaK1Red<<< v, wthreadK1Red >>>();
```

Предварительные результаты. Приведённая выше схеме построения CUDA программы по декларативным описаниям была применена к программе на языке Норма+ из области газодинамики. Была взята программа на языке Си, полученная компиляцией указанной программы на языке Норма+ в последовательную программу, и ручным способом трансформирована в программу с использованием CUDA так, как это мог бы сделать компилятор по описанной выше схеме. Полученная параллельная программа была запущена на вычислительном кластере К-100 [9]. Производительность параллельной программы сравнивалась с последовательной программой и OpenMP версией той же самой программы. Времена выполнения версий программы приведены в таблице 1.

Таблица 1. Время выполнения различных версий программы.

Последовательная программа, 1 ядро Intel Xeon X5670	OpenMP программа, 11 ядер Intel Xeon X5670	CUDA программа, 1 nVidia Fermi C2050
44.5 сек	4.62 сек	1.35 сек

Полученный результат – ускорение в 33 раза по сравнению с последовательной программой – позволяет надеяться на успешное применение схемы для автоматического построения эффективных программ для графических процессоров по декларативным описанием.

Заключение. В настоящее время ведется разработка версии компилятора с языка Норма+ для графических процессоров с использованием технологии CUDA с применением описанной в данной работе схемы.

Работа выполнена при финансовой поддержке гранта РФФИ № 12-01-00527-а.

ЛИТЕРАТУРА:

1. М.А. Кривов, М.Н. Притула, С.Г. Елизаров. Опыт портирования среды для HDR-обработки изображений на GPU и APU. http://pavt.susu.ru/2012/short/175.pdf
2. http://openacc.org
3. В.А. Бахтин, И.Г. Бородич, Н.А. Катаев, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Ю.Л. Сазанов. Распараллеливание с помощью DVM-системы некоторых приложений гидродинамики для кластеров с графическими процессорами. Научный сервис в сети Интернет: поиск новых решений: Труды

Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск). — М.: Изд-во МГУ, 2012. с. 444 – 450.
4. http://colamo.parallel.ru/
5. И.Б. Задыхайло. Организация циклического процесса счета по параметрической записи специального вида. Журн. выч. мат. и мат. физ., т.3, N2, 1963, с.337-357.
6. А.Н. Андрианов, А.Б. Бугеря, К.Н. Ефимкин, И.Б. Задыхайло. Норма. Описание языка. Рабочий стандарт. — М.: Препринт ИПМ им.М.В.Келдыша РАН, 1995, № 120, 52с.
7. А.Н. Андрианов, А.Б. Бугеря, К.Н. Ефимкин, П.И. Колударов. Декларативный язык Норма и программирование для новых архитектур: многоядерные системы. Труды Международной суперкомпьютерной конференции “Научный сервис в сети ИНТЕРНЕТ: Суперкомпьютерные центры и задачи, г. Новороссийск, 2010. — М.: Изд-во МГУ, 2010.
8. Система Норма. http://www.keldysh.ru/pages/norma
9. Гибридный вычислительный кластер К-100. http://www.kiam.ru/MVS/resourses/k100.html