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

Препринты ИПМ • Препринт № 73 за 2016 г.

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

Андрианов А.Н., Баранова Т. П.,
Бугеря А. Б., Ефимкин К.Н.

Трансляция непроцедурного языка НОРМА для графических процессоров

Рекомендуемая форма библиографической ссылки: Трансляция непроцедурного языка НОРМА для графических процессоров / А.Н.Андрианов [и др.] // Препринты ИПМ им. М.В.Келдыша. 2016. № 73. 24 с. doi:10.20948/prepr-2016-73
URL: http://library.keldysh.ru/preprint.asp?id=2016-73

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

А.Н. Андрианов, Т.П. Баранова,
А.Б. Бугеря, К.Н. Ефимкин

Трансляция непроцедурного языка НОРМА для графических процессоров

Москва — 2016

Андрианов А.Н., Баранова Т.П., Бугеря А.Б., Ефимкин К.Н.

Трансляция непроцедурного языка НОРМА для графических процессоров

В работе рассмотрены методы автоматической трансляции непроцедурных спецификаций в исполняемые программы для графических процессоров. На примере непроцедурного языка НОРМА приведены алгоритмы и другие конструктивные решения, с использованием которых был создан компилятор программ на языке НОРМА для графических процессоров с использованием технологии NVIDIA CUDA. Оценивается эффективность исполняемых программ для графических процессоров, получаемых автоматически с помощью компилятора, созданного на основе рассмотренных методов. Приводятся результаты применения компилятора для решения трёх различных задач.

Ключевые слова: суперкомпьютеры, параллельное программирование, графические процессоры, язык НОРМА, CUDA

Alexander Nikolaevich Andrianov, Tatiana Petrovna Baranova, Alexander Borisovich Bugerya, Kirill Nikolaevich Efimkin

Nonprocedural NORMA language translation for GPUs

Methods for automatic translation of nonprocedural specifications to executable program for Graphics Processing Units (GPU) are considered. The algorithms and design solutions realized in the development of the compiler from nonprocedural NORMA language into the program using NVIDIA CUDA technology are described. The performance of automatically generated programs is estimated. The compiler was applied for solving three different tasks, the corresponding results are presented.

Key words: HPC, parallel programming, GPU, NORMA language, CUDA

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 15-01-03039-а.

Введение

В современном научном сообществе задача разработки эффективных параллельных программ имеет важное стратегическое значение. Наверное, уже не осталось ни одной области науки или отрасли промышленности, так или иначе не связанной со сферой высокопроизводительных вычислений. Несмотря на то что параллельное программирование появилось уже достаточно давно, успешно развивается и активно исследуется, вопрос создания эффективных параллельных программ для решения расчетных задач до сих пор крайне актуален для программистов и прикладных специалистов.

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, GPU), а затем – ускорители Xeon Phi. И для каждого такого типа ускорителей опять возникает вопрос о методах и средствах их эффективного программирования. Так, применение технологии CUDA 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].

Конечно, нельзя не отметить, что в составе средств разработки программ для графических процессоров появляются различные новые инструменты, библиотеки, такие как cuBLAS, cuSPARSE [2] и прочие, предоставляющие прикладному программисту эффективные элементарные «кирпичики» для конструирования своей программы, однако при этом всё равно оставляющие нерешёнными ряд задач, таких как управление памятью, распределение вычислений между несколькими вычислителями и т.п. Наличие таких портированных пакетов и специальных библиотек, несомненно, существенно облегчает задачу прикладному специалисту, но только в том случае, если все его потребности в высокопроизводительных вычислениях покрываются имеющимися распараллеленными пакетами и/или библиотеками. Если же с помощью таких готовых средств построить решение для своей задачи не удаётся, то иного пути, кроме как изучить специальные инструменты программирования для целевой архитектуры и начать реализовывать свой алгоритм этими средствами на столь низком уровне, у прикладного специалиста нет.

Надежды на автоматическое распараллеливание уже написанных последовательных программ на различные виды параллельных архитектур пока совершенно не оправдываются, несмотря на то, что фирмы-производители

таких аппаратных решений давно 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 [3].

Работы по созданию и продвижению новых средств и языков программирования для графических процессоров, гибридных решений и различных нетрадиционных вычислительных архитектур (например, FPGA), также ведутся весьма активно [4, 5], однако проблема простой разработки параллельных программ и утилизации новых возможностей вычислительной техники так и остается в настоящее время нерешенной.

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 прочие детали целевой системы при этом не используются), а затем компилятор автоматически строит параллельную программу (уже учитывая архитектуру целевого параллельного компьютера, модели параллелизма и прочие заданные параметры компиляции). 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 [6]. Непроцедурный язык НОРМА и система программирования НОРМА [7-9] 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 систему программирования НОРМА [9] входит компилятор программ на языке НОРМА, позволяющий получать исполняемую программу на заданном процедурном языке программирования для определённой модели параллелизма. До недавнего времени компилятор умел создавать программы на языках Фортран или Си для следующих вычислительных архитектур:

- последовательные программы;
- для многоядерных систем с общей памятью с использованием технологии OpenMP;
- для распределённых систем с использованием технологии MPI.

В результате работ по дальнейшему развитию системы НОРМА, проводимых в ИПМ им. М.В. Келдыша, был разработан компилятор программ на языке НОРМА, который на выходе создаёт исполняемую программу для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA. В данном препринте излагаются принципы отображения непроцедурных спецификаций в вычисления на графических процессорах, рассматриваются возникающие сложности, обсуждаются сопутствующие задачи и способы их решения. Также приводятся результаты испытаний полученного компилятора на реальной задаче из области газодинамики, на тесте CG из пакета NPB (NAS Parallel Benchmarks [10]) и на фрагменте задачи с большой вычислительной плотностью.

Методы 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 исходной программы на языке НОРМА, в случае разрешимости этих зависимостей, строится так называемая «параллельная ярусная схема» выполнения программы. На каждом ярусе этой ярусной схемы располагаются операторы программы, которые не имеют зависимостей друг от друга и могут выполняться независимо и, соответственно, параллельно. Каждый из операторов текущего яруса имеет зависимости 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 исходной программы на языке НОРМА.

Компоновка вычислительных ядер

В результате ряда исследований по трансформации параллельной ярусной схемы программы на языке НОРМА в программу с использованием технологии NVIDIA CUDA для графических процессоров предлагается использовать следующий подход для автоматического построения исполняемой программы, см. рис.1.

Исполняемая программа стартует и завершается на центральном процессоре, на нём же выполняется ввод-вывод данных и итерационные циклы. Вся логика по управлению вычислениями, выделению памяти на графическом процессоре, организации обменов данными между памятью графического и центрального процессора тоже создаётся в программном коде, выполняющемся на центральном процессоре.

Вызовы процедур и функций выполняются из программы на центральном процессоре, если это «обычные» пользовательские или библиотечные функции, имеющие реализации для центрального процессора, или же из вычислительного ядра, выполняющегося на графическом процессоре, если это пользовательские или библиотечные функции, имеющие реализации для графического процессора. Сами вычислительные операторы выполняются, по возможности, 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 над 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.

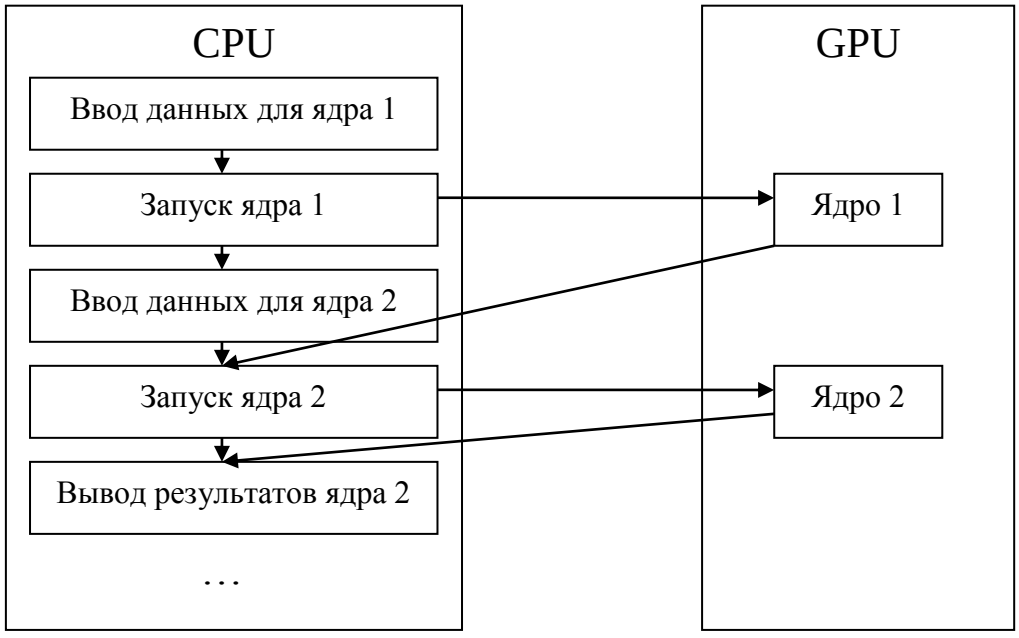

Рис.1. Пример общей схемы выполнения исполняемой программы.

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, то есть чем крупнее будут исполняемые ядра в программе, тем эффективнее она будет работать, так как будет меньше запусков ядер, меньше передачи параметров, в том числе и через память центрального процессора, и эффективнее будет ряд других действий. 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 программы на языке НОРМА (другими словами, в одной точке индексного пространства вычислений).
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 полностью (не обязательно по всем точкам, но обязательно по всем направлениям) 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 (здесь и ниже будем использовать синтаксис запуска ядра в программе с использованием технологии CUDA):

```
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 меньше 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 >>>()
```

4. 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.7) становится невозможным.
5. 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.7) становится невозможным.
6. Если оператор содержит вызов другого раздела или функции, которые выполняются на центральном процессоре, этот оператор не может быть помещён в ядро и дальнейшее добавление в формируемое ядро операторов со следующего уровня (п.7) становится невозможным.
7. 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 выполненные 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. 4-7. Добавление таких операторов возможно при соблюдении следующих условий:

- если на добавляемом слое есть операторы ввода-вывода или операторы, содержащие вызов другого раздела или функции, которые выполняются на центральном процессоре, то такие операторы 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, вычисляемых в этом же ядре, 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.

8. Скалярный оператор без функций редукции может быть выполнен как на центральном процессоре, так и на графическом процессоре (какой-либо одной нитью). Решение о том, помещать ли такой скалярный оператор в формируемое вычислительное ядро или лучше выполнить его на центральном процессоре, принимается в зависимости от того, как выполняются другие операторы – те, от которых зависит рассматриваемый скалярный оператор, и те, которые зависят от него. Если среди таких операторов есть хотя бы один, выполняемый на центральном процессоре, то и рассматриваемый скалярный оператор должен быть выполнен на центральном процессоре. В противном случае он может быть помещён в формируемое вычислительное ядро.

В результате обработки ярусной схемы программы на языке НОРМА по приведённому алгоритму получается линейный набор выполняемых элементов для центрального процессора и связанный с ним набор вычислительных ядер, включающих в себя один или более операторов исходной программы на языке НОРМА. Для каждого такого вычислительного ядра определяются также следующие характеристики:

- Какие переменные из операторов этого ядра первый раз используются на графическом процессоре. Для них перед вызовом данного ядра необходимо будет сгенерировать операторы выделения памяти на графическом процессоре.
- Какие переменные из операторов этого ядра последний раз используются на графическом процессоре. Для них после вызова

данного ядра необходимо будет сгенерировать операторы освобождения памяти на графическом процессоре.

- Какие переменные, используемые операторами этого ядра, имеют на момент вызова ядра своё актуальное значение в памяти центрального процессора. Для них перед вызовом данного ядра необходимо будет сгенерировать операторы передачи данных в соответствующую память графического процессора.
- Какие переменные, вычисляемые операторами этого ядра, потребуются затем в операторах, выполняемых центральным процессором. Для них после вызова данного ядра необходимо будет сгенерировать операторы передачи данных в соответствующую память центрального процессора.

Спецификатор PLAIN

Вызовы разделов и функций должны производиться кодом программы, выполняемым на центральном процессоре, если эти вызываемые разделы или функции не имеют реализаций для графического процессора. И, очевидно, что все стандартные и пользовательские функции, имеющие реализации для графического процессора, могут вызываться из программного кода, выполняемого на графическом процессоре. Компилятор программ на языке НОРМА имеет встроенный список стандартных функций и при их вызове автоматически понимает, что они могут быть вызваны как на центральном процессоре, так и на графическом. Но как быть с пользовательскими функциями, имеющими реализацию на графическом процессоре? Для решения этой проблемы в язык НОРМА было введено новое понятие – спецификатор раздела или функции.

Спецификатор раздела или функции может быть указан в программе на языке НОРМА как в реализации раздела или функции, так и в декларации EXTERNAL в вызывающих разделах и функциях. При этом раздел должен иметь один и тот же спецификатор во всех случаях. На данный момент поддерживаются 2 спецификатора: PLAIN и CUDA.

Пример использования спецификатора CUDA в реализации раздела conj_grad_N:

```
CUDA PART conj_grad_N.
 colidx, rowstr, x, a RESULT z, rnorm
BEGIN …
```

Пример использования спецификатора PLAIN в декларации EXTERNAL функции conj_grad_HF1:

```
PLAIN EXTERNAL FUNCTION conj_grad_HF1 DOUBLE.
```

Спецификатор CUDA будет описан ниже. Спецификатор PLAIN предназначен для того, чтобы сообщить компилятору, что данная вызываемая функция или раздел является обычной «плоской» функцией или разделом: не

содержит внутри себя какие-либо распараллеливающие конструкции и имеет реализацию для выбранного выходного языка – в данном случае реализацию для графического процессора с использованием технологии NVIDIA CUDA.

При вызове функций и разделов, объявленных внешними со спецификатором PLAIN, компилятор будет действовать так же, как и при вызове стандартных функций: генерировать код вызова там же, где происходит генерация всего оператора, содержащего этот вызов. Если же производится вызов функции или раздела, объявленного внешним без спецификатора PLAIN, то оператор, содержащий этот вызов, будет выполняться на центральном процессоре.

Эффективное управление памятью

При создании программ для графических процессоров, в силу того что объём памяти графического процессора обычно гораздо меньше, чем объём памяти центрального процессора, всегда встаёт вопрос об эффективном использовании памяти графического процессора. Это становится особенно актуально при решении задачи автоматической генерации программ для графических процессоров, так как при автоматической генерации программ, как правило, возникают дополнительные вспомогательные переменные, необходимые для осуществления заданных операций, которые требуют выделения дополнительной памяти. Но несомненное достоинство автоматической генерации программ – это обладание полной информацией о том, где и как в программе используются переменные, а следовательно, это возможность полностью контролировать процесс выделения и освобождения памяти и полностью исключить возможность «утечки» памяти в результате её не освобождения после использования.

Очевидно, что наиболее эффективным с точки зрения минимизации объёма требуемой памяти графического процессора был бы подход, при котором память запрашивалась бы непосредственно перед вызовом ядра, где она бы использовалась, и освобождалась бы сразу после такого использования. Но в целом ряде случаев буквальная реализация такого подхода приводит к катастрофическому провалу по производительности получающихся программ. Операции выделения, освобождения памяти и передача данных между памятью центрального и графического процессоров – очень дорогие по времени операции. Поэтому, например, если временная переменная требуется на каждом шаге итерации, то заводить её в памяти на каждом шаге, а потом сразу после использования уничтожать – это приведёт к тому, что основное время работы программы будет тратиться на операции с памятью, а не собственно на вычисления. Другой пример – если переменная требуется для вычислений в одном из ядер, а затем, ещё через несколько шагов – в другом ядре, и компилятор обладает информацией, что значение этой переменной между вызовами этих ядер изменено не было. Очевидно, что с точки зрения производительности гораздо эффективнее будет сохранить в памяти

графического процессора значение такой переменной после вызова первого ядра, чем освободить занимаемую переменной память, выделять её заново и заново загружать значения из памяти центрального процессора. Хотя с точки зрения требуемого объёма памяти графического процессора это приводит к излишним накладным расходам (в памяти хранится переменная, не требующаяся для текущих расчётов).

При реализации механизма управления памятью в компиляторе программ на языке НОРМА для графических процессоров было решено использовать подходы, позволяющие достичь максимальной производительности получающихся программ, пусть даже за счёт повышенных требований к объёму памяти графического процессора. Были сформулированы и реализованы в компиляторе следующие правила управления памятью:

1. Память на графическом процессоре выделяется перед первым использованием переменной каким-либо вычислительным ядром и освобождается после последнего использования.
2. Если операции выделения или освобождения памяти на графическом процессоре попадают внутрь итерации, то они выносятся за пределы итерационного цикла (выделение – до начала цикла, освобождение – после окончания цикла).
3. Компилятор отслеживает то, где находится актуальное значение переменной – в памяти центрального процессора, или в памяти графического процессора, или и там и там. Это позволяет не осуществлять копирование значений между памятью центрального и графического процессора, если целевая память и так уже содержит актуальное значение. Актуальность значения отслеживается для переменной в целом, для всей вычисленной подобласти. Если происходит частичное присваивание значения (по какой-то подобласти) на одном из процессоров, то для памяти другого процессора полагается, что переменная теряет свою актуальность.

Соглашение о вызовах «CUDA», спецификатор CUDA

Программа на языке Норма может состоять из нескольких разделов и функций. Разделы и функции вызывают друг друга, передают параметры. Если вызывается не «плоский» раздел (описанный со спецификатором PLAIN), то вызов осуществляется на центральном процессоре и параметры ему передаются как обычно – по значению или по указателю в памяти центрального процессора. Вызываемый раздел рассматривает свои формальные параметры как обычные переменные и дальше уже сам решает, нужно ли ему для вычислений выделять соответствующую память в графическом процессоре и копировать туда значения своих формальных параметров, применяя механизм управления памятью, описанный выше, или нет. Такой подход прекрасно инкапсулирует происходящее в вызываемом разделе от вызывающего, но имеет

существенный недостаток: если вызов происходит много раз в каком-то цикле, то при каждом вызове раздела для каждого передаваемого параметра будут производиться операции выделения/освобождения памяти графического процессора и копирования данных. Эта ситуация аналогична описанной выше, когда начало и конец использования переменной находится внутри итерации. Но если в пределах трансляции одного раздела эту проблему можно решить, просто вынеся операции с памятью за пределы итерационного цикла, то в случае вызова раздела в цикле из другого раздела операции с памятью, которые должен производить вызываемый раздел, должны выноситься за пределы цикла в вызывающем разделе. Реализовать такие операции при условии раздельной трансляции разделов не представляется возможным.

Для решения этой проблемы была разработана и реализована в компиляторе следующая схема. Для каждого параметра, передаваемого через память центрального процессора, создаются дополнительно ещё два связанных с ним параметра: указатель на соответствующую память графического процессора и переменная-статус, определяющая, в какой памяти находится актуальное значение передаваемого параметра. Причём эти два параметра передаются по указателю, что даёт возможность вызываемому разделу их изменять, а вызывающему – учитывать сделанные изменения. Таким образом, если вызываемому разделу нужно произвести вычисления на графическом процессоре с использованием формального параметра, то при первом вызове он выделит память на графическом процессоре и запишет её адрес в первом дополнительном параметре. Затем он скопирует значение переменной в выделенную память и отразит это в статусе (втором дополнительном параметре). И потом, при всех последующих вызовах, при сохранении значений в дополнительных параметрах, вызываемый раздел будет понимать, что данные уже в памяти графического процессора, и никаких лишних манипуляций производить не будет. А освобождать выделенную память будет уже вызывающий раздел, после цикла вызовов, проанализировав первый дополнительный параметр – если он имеет отличное от NULL значение, то его надо освободить.

Такой алгоритм передачи параметров был назван соглашением о вызовах «CUDA». Для его успешного использования необходимо, чтобы он был применён как вызывающим, так и вызываемым разделом. Для этого в язык НОРМА был введён новый спецификатор раздела или функции CUDA. При его наличии и при генерации выходной программы для графических процессоров с использованием технологии NVIDIA CUDA компилятор будет генерировать актуальные и формальные параметры и их использование согласно данному соглашению.

Следует отметить, что использование соглашения о вызовах «CUDA» решает проблему эффективного использования памяти графического процессора для передаваемых параметров, но, к сожалению, аналогичную проблему для локальных и временных переменных вызываемого раздела

решить не в состоянии. Для этого необходимо будет разработать и реализовать другие механизмы межпроцедурного взаимодействия.

Реализация функций редукции

К функциям редукции в языке НОРМА относятся функции сумма, произведение, максимум и минимум. Вычисление функций редукции выполняется на графическом процессоре, по возможности в одном ядре с другими операторами.

Перед началом вычисления данные из области редукции копируются в разделяемую память блока нитей. Далее каждая нить блока выполняет операцию над двумя значениями в разделяемой памяти, с сохранением промежуточного результата в той же области памяти. После этого нити блока синхронизируются, и программа переходит к следующему шагу, на котором число работающих нитей блока уменьшается в 2 раза (число нитей в блоке всегда генерируется равным какой-то степени 2). На этом шаге каждая работающая нить блока выполняет операцию над двумя значениями, вычисленными на предыдущем шаге. Так повторяется до тех пор, пока последняя единственная работающая нить не получит результат для данных в одном блоке.

Если область применения функции редукции можно полностью посчитать на одном блоке нитей графического процессора, такая функция может быть вычислена одним ядром вместе с другими операторами программы. Иначе, если область применения функции редукции больше количества нитей в блоке, для вычисления функции необходимо продолжить обработку промежуточных результатов вычислений, полученных в каждом блоке. Так как технология CUDA не предоставляет средств для межблочной синхронизации, продолжение вычислений с использованием данных из разных блоков необходимо организовывать завершением текущего ядра и запуском нового, в котором каждому полученному промежуточному результату будет соответствовать одна нить нового распределения на блоки и нити.

Таким образом, для вычисления одной функции редукции может потребоваться запуск нескольких ядер последовательно. Количество ядер зависит от размеров области применения функции редукции и от количества нитей в блоке графического процессора. В первое ядро функции редукции включаются другие операторы текущего уровня, но добавление в это ядро операторов со следующего уровня становится невозможным.

Для передачи промежуточных результатов между ядрами перед запуском первого ядра функции редукции в глобальной памяти графического процессора выделяется область памяти. Каждый блок считывает исходные данные из своего участка этой области и в него же записывает результат, поэтому синхронизации и выделения новой памяти на последующих шагах не требуется.

В первом ядре функции редукции распределение по нитям происходит так же, как и в случае, когда редукцию можно посчитать целиком на одном ядре.

Результатом вычисления первого ядра будет Nd/Nt значений, где Nd – количество точек в области, а Nt – количество нитей в блоке. Результат будет записан в глобальной памяти графического процессора и передан в следующее ядро. Во втором и следующих ядрах распределение по индексам происходит так же, как в первом, но количество точек в исходной области будет в Nt раз меньше. Результатом вычисления второго ядра будет Nd/Nt/Nt значений, и он будет записан в глобальную память вместо результата первого ядра. Будет вызвано следующее ядро, и так до тех пор, пока не будет получен результат по всей области.

Оператор в языке НОРМА вместе с функциями редукции может содержать в себе арифметические выражения как в параметрах функции редукции, так и с результатом функции редукции. Арифметические выражения в параметрах функции редукции вычисляются в первом ядре перед началом вычисления функции редукции. В последнем ядре вычисляются арифметические выражения с результатом функции редукции.

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.

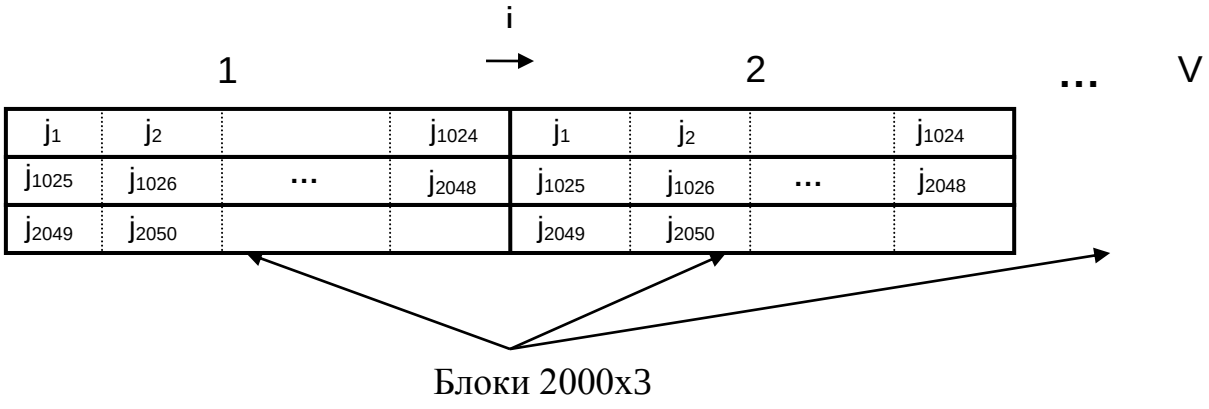

Рис. 2. Отображение области Oij на блоки и нити.

Ниже приведена получившаяся программа для первого ядра вместе с общими определениями констант. В ней производится инициализация переменной Vij (в ядре представлена переменной Vij_dev) и начинается ее суммирование по области Oj. Для этого каждый блок вычисляет сумму для всех входящих в него нитей. Но, т.к. точки области Oj распределены на 3 блока, то итоговый результат, который должен быть вычислен как сумма частичных результатов, полученных в каждом из этих 3-х блоков, в рамках данного ядра вычислить невозможно.

```
#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];
    // Получаем индекс j
    int j = threadIdx.x + blockIdx.y*wthreadK1;
    if(j < w) { // Проверяем что не вышли за границу области
        int i = blockIdx.x;
        int gidx = i*w + j;
        // Присваиваем начальные значения
        Vij_dev[gidx] = j + 1 + i * w;
        // 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];
    }
}
```

Поэтому для завершения суммирования организуется второе ядро.

```
__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 >>>();
```

Этот фрагмент, как можно заметить, заключается в последовательном вызове вышеприведённых ядер на нужной конфигурации блоков и нитей.

Результаты применения компилятора

Приведённые выше методы и решения построения программы для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA по декларативным описаниям были реализованы в компиляторе программ на языке НОРМА. Полученный компилятор был протестирован на реальной задаче из области газодинамики, на тесте CG из пакета NPB и на фрагменте задачи с большой вычислительной плотностью. Во всех трёх примерах применения компилятора удалось добиться получения полностью корректного результата и была проведена оценка эффективности получающихся исполняемых программ. Производительность программ для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA сравнивалась с последовательной и OpenMP версиями той же самой программы. Для теста CG из пакета NPB также произведено сравнение с оригинальными версиями теста, написанными вручную. Все запуски производились на вычислительном кластере K-100 [11]. Компиляция последовательных и OpenMP версий программ производилась компилятором Intel версии 15.0.0, компиляция CUDA программ – компилятором nvcc версии 6.5.

Прикладная задача из области газодинамики

В основном рабочем итерационном цикле данной программы производится расчёт различных величин для всех точек одномерной области. Для каждой точки вызывается «плоский» внешний раздел, который на основе предыдущих значений точки и её соседей высчитывает данные для новых значений. Все вычисления производятся с одинарной точностью. Времена выполнения версий программы приведены в таблице 1.

Таблица 1

Время выполнения программы решения задачи из области газодинамики

Последовательная программа, 1 ядро Intel Xeon X5670	OpenMP программа, 12 ядер Intel Xeon X5670	CUDA программа, 1 nVidia Fermi C2050
461 сек	45.6 сек	16.9 сек

Полученный результат – ускорение в 27 раз по сравнению с последовательной программой – показывает, что для такого класса задач язык НОРМА подходит отлично. Построенная полностью автоматически по непроцедурным спецификациям на языке НОРМА программа для графических процессоров демонстрирует высокую скорость выполнения и эффективность применённых в компиляторе программ на языке НОРМА методов и решений.

Реализация теста CG из пакета NPB

Пакеты измерения представляют собой специальные наборы тестов. Тесты классифицируются на ядра (kernels) и приложения (applications benchmarks). Ядра – это фрагменты кода, взятые из реальных приложений, позволяющие измерять скорость исполнения реальной программы на разных платформах. Приложения – это программы, созданные на основе реальных приложений и адаптированные для задач тестирования. Спецификой ядер и приложений является то, что они отражают часть программы, которая занимает наибольшее время выполнения (алгоритмы численных методов, перемножение матриц, векторов). Наиболее широко тесты для измерения производительности кластерных систем представлены в пакете NAS Parallel Benchmarks (NPB) [10], обладающем помимо объективных достоинств еще одним необъективным – авторитетностью программ и их авторов. Это послужило причиной выбора данного пакета для исследования возможности реализации тестов на языке НОРМА и проверки эффективности кода, генерируемого компилятором с языка НОРМА.

После анализа исходных текстов программ тестового пакета NPB для эксперимента по портированию данного тестового пакета на язык НОРМА в первую очередь было выбрано ядро CG, как наиболее подходящее под концепцию языка НОРМА. Тестовое ядро CG (Conjugate Gradient) осуществляет приближение к наименьшему собственному значению большой разреженной симметричной положительно определенной матрицы с использованием метода обратной итерации вместе с методом сопряженных градиентов в качестве подпрограммы для решения СЛАУ. Тест применяется для оценки скорости передачи данных при отсутствии какой-либо регулярности. Вычисления производятся в определенных классах задач. Под классом понимается размерность основных массивов данных, используемых в тесте: «Sample code», «Class A» (маленькие), «Class B» (большие), «Class C»

(очень большие), «Class D» (огромные). Все вычисления производятся с двойной точностью.

После анализа исходных текстов ядра CG было принято решение переписать на языке НОРМА основную рабочую функцию теста, а также этапы инициализации теста и основной тестовый цикл. В то же время стало очевидно, что такие действия, как начальная инициализация разреженной матрицы, плохо подходят для реализации средствами языка НОРМА. Завершающий шаг – анализ правильности полученных результатов и вывод статистических данных о прохождении теста – также было решено оставить в оригинальном виде на Си для неоспоримости факта правильности анализа полученных результатов оригинальным кодом и сохранения формата вывода. Фактически получается что тест начинает свою работу и заканчивает её кодом на Си, а та часть, в которой производятся все вычисления и при выполнении которой производится засечка времени и делается вывод о производительности системы – реализуется на языке НОРМА. Поэтому в оставшемся коде на Си не нужно делать никаких предположений об архитектуре целевой системы и об используемых инструментах параллельного программирования. Этот код так и остаётся обычным последовательным кодом на Си и на быстродействие теста никак не влияет.

Среди конструкций, которые необходимо было портировать на язык НОРМА оказались и такие, что не могут быть выражены средствами языка НОРМА. В первую очередь речь идёт о «сердце» теста – умножении разреженной матрицы на вектор. Но язык НОРМА поддерживает вызовы так называемых «внешних» функций, которые могут быть написаны на целевом языке программирования. Задача по реализации умножения разреженной матрицы на вектор (это делается с помощью косвенной адресации и цикла с переменными границами) была возложена на такую вспомогательную внешнюю функцию, а в программе на языке НОРМА в каждой точке области, соответствующей результирующему вектору, результат выполнения этой внешней функции в данной точке присваивается результирующей переменной.

Но подобная реализация умножения разреженной матрицы на вектор (фактически оператор оставлен в том же виде, что был в оригинальном тесте), несмотря на очевидное достоинство – простоту реализации (вся внешняя функция получилась из 8 строк) – и хорошие показатели распараллеливания в версии OpenMP, плохо подошла для выполнения на графических процессорах. Поэтому были разработаны и реализованы ещё 2 варианта этой внешней функции, специально для версии для графических процессоров, которые учитывают особенности этой архитектуры. В первом из них каждая точка области рассчитывается одним блоком графического процессора. А цикл с переменными границами, в котором производится суммирование вычисленных значений, выполнен в виде редукции по нитям, где каждая нить предварительно высчитывает своё значение, соответствующее значению на определённом витке цикла. Такая реализация внешней функции получилась приблизительно в 120

строк кода. Второй вариант – использовать функцию из библиотеки cuSPARSE [2]. Такая реализация внешней функции получилась приблизительно в 60 строк кода. Результаты выполнения различных версий теста, как оригинальных, так и полученных в результате компиляции программ с использованием языка НОРМА, приведены в таблице 2. Измерения производились для классов теста A, B и C. Во всех случаях проверка результата прошла успешно и приведённые цифры – количество миллионов операций в секунду (Mop/s), основной показатель скорости выполнения теста.

Таблица 2

Результаты запуска теста CG (Mop/s)

	Class A		Class B		Class C	
	Си	Си + НОРМА	Си	Си + НОРМА	Си	Си + НОРМА
Последовательная программа, 1 ядро Intel Xeon X5670	1200	1175	718	716	513	615
OpenMP программа, 12 ядер Intel Xeon X5670	9737	8035	3595	3004	3240	2806
CUDA программа, простая реализация, 1 nVidia Fermi C2050	—	927	—	722	—	645
CUDA программа, реализация с редукцией, 1 nVidia Fermi C2050	—	2134	—	2178	—	1876
CUDA программа, использование cuSPARSE, 1 nVidia Fermi C2050	—	6521	—	4504	—	2704

Как можно заметить, результаты выполнения теста CG с использованием языка НОРМА практически идентичны (видимо, с точностью до погрешности измерений) соответствующим результатам оригинального теста CG, написанного вручную. Это даёт нам право утверждать, что тест CG прекрасно подошёл для записи его на языке НОРМА. И что задачи, имитирующиеся этим тестом, также должны хорошо подходить для программирования их (или по крайней мере их основных вычислительных ядер) на языке НОРМА. Ничуть не проиграв по скорости выполнения получившихся программ, компилятор программ с языка НОРМА полностью взял на себя задачу по автоматическому распараллеливанию выходной программы и успешно её выполнил.

В то же время видно, что для эффективного выполнения на графических процессорах таких задач необходимо прибегать к более тонкому ручному программированию или, когда это возможно, – использовать специальные библиотеки.

Фрагмент задачи с большой вычислительной плотностью

Данный фрагмент было решено привести здесь как пример вычислений, которые отлично подходят как для записи на языке НОРМА, так и для последующих вычислений на графических процессорах. Такие вычисления (с небольшим изменением) были выявлены в реальной прикладной задаче, когда один из авторов данной статьи занимался ручным распараллеливанием этой прикладной задачи. После выделения данного фрагмента было решено попробовать вынести его в отдельную процедуру и записать его на языке НОРМА, чтобы посмотреть, каким будет результат автоматического распараллеливания его на различные архитектуры. Приведём этот фрагмент полностью:

```
DOMAIN PARAMETERS Nx=300,Ny=300.
ok1:(k1=1..Nx). ok2:(k2=1..Ny). ok:(ok1;ok2).
oi1:(i1=1..Nx). oi2:(i2=1..Ny). oi:(oi1;oi2).
VARIABLE X,Y DEFINED ON ok DOUBLE.
    FOR ok ASSUME   X   = 1.0. // Initial data
CONSTANT d1 = 10.0D. CONSTANT d2 = 20.0D.
CONSTANT pi2 = 3.14157D/2.0.
    FOR ok ASSUME Y =
  SUM((oi)X[k1=i1,k2=i2]*sin(fmod(k1*i1*d1*k2*i2*d2,pi2))/i2).
```

В данном фрагменте в каждой точке двумерной области происходит суммирование по такой же двумерной области выражения, где синус вычисляется по данным, зависящим как от точки области результата, так и от точки области суммирования. Все вычисления производятся с двойной точностью. В итоге получаются массовые однородные независимые друг от друга вычисления. Времена выполнения этого фрагмента на различных архитектурах приведены в таблице 3.

Таблица 3

Время выполнения фрагмента задачи с большой вычислительной плотностью

Последовательная программа, 1 ядро Intel Xeon X5670	OpenMP программа, 12 ядер Intel Xeon X5670	CUDA программа, 1 nVidia Fermi C2050
894 сек	80.8 сек	15.3 сек

Продемонстрированная высокая производительность вычислений на графическом процессоре показывает, что для такого класса задач применение языка НОРМА более чем оправдано. По достаточно короткой непроцедурной записи компилятор программ на языке НОРМА способен автоматически сгенерировать эффективное решение для параллельных архитектур.

Заключение

В настоящее время ведется разработка версии компилятора с языка НОРМА для высокопроизводительных вычислительных систем с гибридной архитектурой. В генерируемой выходной программе могут быть одновременно использованы технологии MPI, OpenMP и NVIDIA CUDA. Компилятор будет автоматически решать такие задачи, как распределение вычислительной нагрузки между графическими процессорами и центральным процессором, а также между узлами гибридной вычислительной системы, организация обмена данными между узлами и внутри одного узла и ряд других.

Литература

1. М.А. Кривов, М.Н. Притула, С.Г. Елизаров. Опыт портирования среды для HDR-обработки изображений на GPU и APU. URL: http://pavt.susu.ru/2012/short/175.pdf
2. CUDA Toolkit Documentation. URL: http://docs.nvidia.com/cuda
3. OpenACC, URL: http://openacc.org
4. 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.
5. Описание языка программирования COLAMO. URL: http://colamo.parallel.ru
6. И.Б. Задыхайло. Организация циклического процесса счета по параметрической записи специального вида. Журн. выч. мат. и мат. физ., т.3, N 2, 1963, с.337-357.
7. А.Н. Андрианов, А.Б. Бугеря, К.Н. Ефимкин, И.Б. Задыхайло. НОРМА. Описание языка. Рабочий стандарт. — М.: Препринты ИПМ им.М.В.Келдыша. — 1995. — № 120. — 52 с.
8. А.Н. Андрианов, А.Б. Бугеря, Е.Н. Гладкова, К.Н. Ефимкин, П.И. Колударов. Простые вещи. — Суперкомпьютеры, 2014, № 2(18) — Москва: Изд-во СКР-Медиа , 2014. — с. 58-61.
9. Система НОРМА. URL: http://www.keldysh.ru/pages/norma
10. NAS Parallel Benchmarks. URL: http://www.nas.nasa.gov/publications/npb.html
11. Гибридный вычислительный кластер К-100. URL: http://www.kiam.ru/MVS/resourses/k100.html

Оглавление

Введение ... 3
Декларативный подход. Язык НОРМА ... 4
Методы построения CUDA программы по декларативным описаниям ... 5
Компоновка вычислительных ядер ... 6
Спецификатор PLAIN ... 11
Эффективное управление памятью ... 12
Соглашение о вызовах «CUDA», спецификатор CUDA ... 13
Реализация функций редукции ... 15
Пример построения программы с использованием CUDA ... 16
Результаты применения компилятора ... 18
Прикладная задача из области газодинамики ... 18
Реализация теста CG из пакета NPB. ... 19
Фрагмент задачи с большой вычислительной плотностью ... 22
Заключение ... 23
Литература ... 23