KernelGen – система компиляции для программной модели «центральный графический процессор – периферийная хост-система»*

Н.Н. Лихогруд[1], Д.Н. Микушин[2], А.В. Адинец[3]

Факультет Вычислительной математики и кибернетики МГУ им. М.В. Ломоносова[1], НОЦ «Параллельные вычисления»[2], Научно-исследовательский вычислительный центр МГУ им М.В. Ломоносова[3]

В работе рассматривается метод автоматической генерации CUDA-кода из программ на языках Си и Фортран, отличающийся полным переносом исполнения программы на GPU, включая последовательные части. Данный экспериментальный метод позволяет сократить количество передаваемых по обменной шине данных за счёт исполнения бóльшего объёма кода на GPU. Преобразование и компиляция программ проводится с помощью DragonEgg, LLVM, Polly/LLVM и nvopencc. В настоящее время система позволяет компилировать корректный и работоспособный PTX-ассемблер для нескольких тестовых приложений.

1. Введение

Массовое использование GPU в кластерных вычислительных системах требует массовой адаптации множества сложных приложений. Программная модель CUDA достаточно хорошо подходит для небольших программ с ярко выраженным вычислительным ядром. Однако для сложных приложений, состоящих из множества отдельных блоков, таких как математические модели, сложность настройки взаимодействия оригинального кода и кода для GPU многократно возрастает. По этой причине для упрощения программирования GPU активно развиваются гибридные программные модели, позволяющие с помощью директив компиляции разметить участки кода исходной программы для работы на GPU, по аналогии с OpenMP. Для стандартизации набора директив основными разработчиками подобных коммерческих технологий создан консорциум OpenACC [1]. Существуют аналогичные открытые системы F2C-ACC [2] и KernelGen 0.1 [3] для преобразования исходного кода на языке Fortran в аналогичную гибридную форму, но они не могут автоматически оценивать параллельность переносимых на GPU вычислительных циклов. Открытые трансляторы внутреннего представления вычислительных циклов компилятора GCC в OpenCL-код [5] и компилятора Clang в CUDA-код [6] производят преобразования с учётом зависимостей данных на основе модели CLooG [7].

Программные модели с выделением фрагментов кода для акселерации предполагают расстановку необходимых аннотаций вручную. По этой причине редко удаётся полностью портировать достаточно большое приложение на GPU, что приводит к наличию обменов данными, связывающих CPU- и GPU-части. Например, при портировании только одного блока WSM5 модели WRF с помощью директив PGI Accelerator, время обменов данными составляет 40-60% общего времени [12]. Такая же ситуация возможна, если для выполнения на CPU оставлять все непараллельные фрагменты кода.

В существующем стандартном процессе компиляции CUDA-кода *nvcc* вспомогательный процессор $cudafe$ работает только с C++-кодом, разделяя каждый входной исходный файл на CPU-часть и GPU-часть, дополнительно преобразуя некоторые конструкции языка. Затем CPU-часть обрабатывается CPU-компилятором (gcc, icc, cl.exe и др.), а GPU-часть – компилятором *nvopencc* [8] (вариант $open64$ [9] с бэкендом для генерации ptx-ассемблера).

*Работа поддержана контрактами НОЦ «Параллельные вычисления» 12-2011 и 13-2011, тестирование велось на оборудовании, установленном на факультете ВМК МГУ, НИВЦ МГУ и компании NVIDIA.

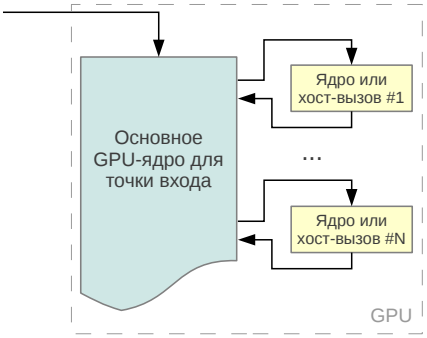

Рис. 1. Схема исполнения «центральный графический процессор – периферийная хост-система»

Теоретически, обе части можно было бы компилировать с помощью *open*64, но, вероятно, разработчики желали сохранить привязку приложений к оригинальному CPU-компилятору для лучшей совместимости. Таким образом, несмотря на то, что компиляторы обоих ветвей помимо C++ поддерживают и другие языки, $cudafe$ и сам способ организации расщепления кода делают их использование невозможным.

При разработке гибридных систем компиляции также необходимо предусмотреть возможность применения к кластерным приложениям с множеством GPU, которые уже распараллелены по вычислительным элементам. Если используется многопроцессное распределение (например, MPI), то необходимо переключение GPU, если многопоточное (OpenMP, POSIX Thread), то также необходима потоковая безопасность (thread-safety) управляющей системы (runtime-библиотеки).

Настоящая работа является развитием существующей системы компиляции KernelGen. На данном этапе были поставлены и решены следующие основные задачи:

Минимизировать обмен данными между памятью системы и GPU за счёт переноса всего кода на GPU. Во всех существующих в настоящее время системах компиляции разметка исходного кода определяет отдельные участки для выполнения на GPU, что делает неизбежным явный обмен данными. Более того, и в интегрированных системах типа AMD Fusion (если их поддержка будет включена в OpenACC), где память CPU и GPU используют один и тот же физический носитель, адресные пространства логически разделены и требуют копирования данных. Данная разработка впервые использует противоположную технику: по умолчанию на GPU переносится *весь* код приложения, предполагая, что даже последовательные части кода может быть более выгодно исполнять на GPU, чем обмениваться данными и выполнять их на CPU. Тем не менее, полностью перенести весь код на GPU невозможно, например, из-за наличия системных вызовов или операций ввода-вывода. Для этого случая необходимо предусмотреть режим обратного вызова, при котором исполнение GPU-ядра останавливается до завершения хост-вызова. Таким образом, элементы привычной модели «основная хост-система и периферийный графический процессор» меняются местами: «центральный графический процессор – периферийная хост-система» (рис. 1).

Обеспечить поддержку широкого множества языков программирования. Как показал опыт разработки KernelGen версии 0.1, создать препроцессор разделения кода для другого языка, аналогичный $cudafe$, возможно, но не слишком рационально по необходимым ресурсам. Подходящей альтернативой разделению исходного кода могут быть преобразования на уровне внутреннего представления компилятора. KernelGen использует LLVM [10] (инфраструктура компилятора с простым внутренним представлением LLVM IR

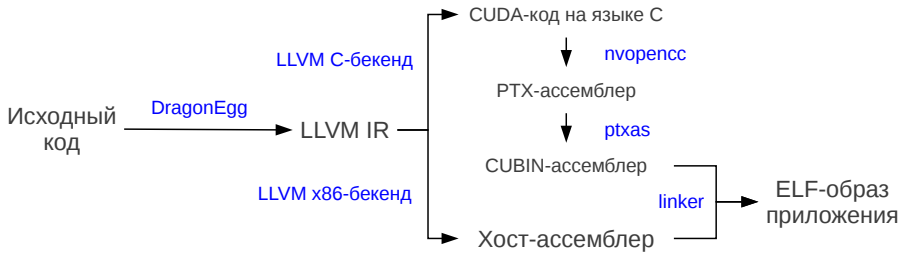

Рис. 2. Схема компиляции кода KernelGen

SSA) и DragonEgg [11] (плагин gcc, генерирующий LLVM IR для любого поддерживаемого входного языка), как показано на рис. 2

Производить автоматическую оценку параллельности и преобразование вычислительных циклов. Для определения параллельных участков кода задействован CLooG, но не напрямую, а посредством LLVM Polly [4], который встраивает выпуклый анализ циклов в структуру оптимизирующих преобразований (passes) LLVM. В настоящее время LLVM Polly поддерживает генерацию эффективного кода только для CPU и OpenMP. В данной работе он был изменён таким образом, чтобы распараллеливать код с учётом специфики GPU: отображение многомерных параллельных циклов на многомерную вычислительную решётку и запуск ядер средствами CUDA API.

В следующих разделах настоящей статьи дана харатеристика этапов преобразования исходного кода и генератора параллельных циклов.

2. Этапы преобразования исходного кода

KernelGen работает напрямую с оригинальным приложением, не требуя каких-либо изменений ни в исходном коде, ни в системе сборки. Для наилучшей поддержки больших приложений это свойство чрезвычайно важно и часто недооценивается. Чтобы обеспечить обычный процесс сборки, код для GPU сначала добавляется в отдельную секцию объектных файлов, затем объединяется и снова разделяется на отдельные ядра на этапе линковки. Окончательная компиляция GPU-ядер в ассемблер происходит при необходимости, уже во время работы приложения (JIT, just-in-time compilation).

В результате работы компилятора, исходное приложение преобразуется во множество GPU-ядер: одно или несколько *основных* ядер и множество *вычислительных* ядер. Основные ядра исполняются на GPU в одном потоке. Их задача – хранить данные, исполнять последовательные участки кода и производить вызовы вычислительных ядер и отдельных CPU-функций, которые невозможно перенести на GPU (рис. 1). Вычислительные ядра исполняются на GPU множеством параллельных нитей с полной загрузкой мультипроцессоров. Таким образом, максимальная доля кода выполняется на GPU, а CPU лишь координирует исполнение. В частности, при работе MPI-приложения каждый рабочий процесс в данном случае будет представлять собой GPU-ядро с небольшим числом CPU-вызовов MPI. Использование MPI дополнительно облегчается за счёт поддержки GPU-адресов в командах обмена данными [13].

Компиляция. При компиляции отдельных объектов, генерируется как x86-ассемблер (таким образом, приложение по-прежнему работоспособно при отсутствии GPU), так и представление LLVM IR. Для разбора исходного кода используется компилятор GCC, чьё внутренее представление *gimple* преобразуется в LLVM IR с помощью плагина DragonEgg. Затем в IR-коде производится выделение тел циклов в отдельные функции, вызываемые

через универсальный интерфейс вида

```
__device__ int kernelgen_launch(
    unsigned char* name, unsigned long long szarg, unsigned int* arg);
```

где name – имя или адрес функции (вместо имён в начале работы программы подставляются адреса), arg – структура, агрегирующая аргументы вызова, szarg – её размер.

Стандартный механизм выделения циклов в функции LLVM *CodeExtractor* расширен, так чтобы цикл не заменялся, а дополнялся вызовом функции по условию:

```
if (kernelgen_launch(name, szarg, arg) == -1)
{
    // Lauch original loop.
}
```

Таким образом, *kernelgen_launch* запускает параллельное ядро только при выполнении определённых условий, в противном случае (например, если в цикле слишком мало итераций) оригинальный цикл выполняется последовательно в основном GPU-ядре.

Далее, существовавшие в оригинальном коде функции подставляются друг в друга (inline). Подстановка необходима, т.к. ни CUDA, ни OpenCL не реализуют для GPU систем линковки (и, вообще говоря, неясно как её реализовывать при статическом распределении регистров и разделяемой памяти), а значит полученные ядра должны быть как можно более самодостаточными. Если ядро имеет неразрешимую внешнюю зависимость, то такой вызов преобразуется в вызов вида

```
__device__ void kernelgen_hostcall(
    unsigned char* name, unsigned long long szarg, unsigned int* arg);
```

при котором GPU-приложение останавливает свою работу и передаёт данные и адрес функции для выполнения на CPU. Функции *kernelgen_launch* и *kernelgen_hostcall* работают в GPU-ядре и вызывают остановку его выполнения. После завершения работы другого ядра или CPU-функции, основное ядро продолжает работу.

Линковка. При линковке отдельных объектов в результирующее приложение или библиотеку, в IR-коде производится окончательная подстановка и оптимизация. В результате предыдущих преобразований, единый IR-модуль содержит код main-функции (или код множества точек входа в случае динамической библиотеки) и код функций с отдельными вычислительными циклы.

Отдельной обработки требуют глобальные переменные. С одной стороны, глобальные переменные размещаются в глобальной памяти (а значит могут совместно использоваться различными ядрами), с другой – *nvopencc* компилирует их в неинициализированные символы вместо внешних из-за отсутствия линковки. Таким образом, синхронизацией глобальных переменных между несколькими ядрами необходимо управлять вручную. Вместо этого на данном этапе была реализована локализация: определения включены в тело основного ядра и в аргументы вычислительных ядер. Однако, такое решение некорректно в случае нескольких основных ядер.

Модель исполнения. Основное ядро запускается в самом начале выполнения приложения и работает на GPU постоянно. Во время работы вычислительного ядра или CPU-функции основное ядро переходит в состояние активного ожидания и продолжает работу после завершения внешнего вызова. Для реализации данной схемы GPU должно поддерживать одновременное исполнение нескольких ядер (concurrent kernel execution) или временную выгрузку активного ядра (kernel preemption). Одновременное исполнение ядер доступно в GPU NVIDIA, начиная с Compute Capability 2.0, в GPU AMD такой возможности нет, но есть вероятность появления kernel preemption в одной из следующих версий OpenCL. По этой причине в данный момент KernelGen работает только с CUDA.

Вызовы *kernelgen_launch* и *kernelgen_hostcall* состоят из двух частей – device-функции

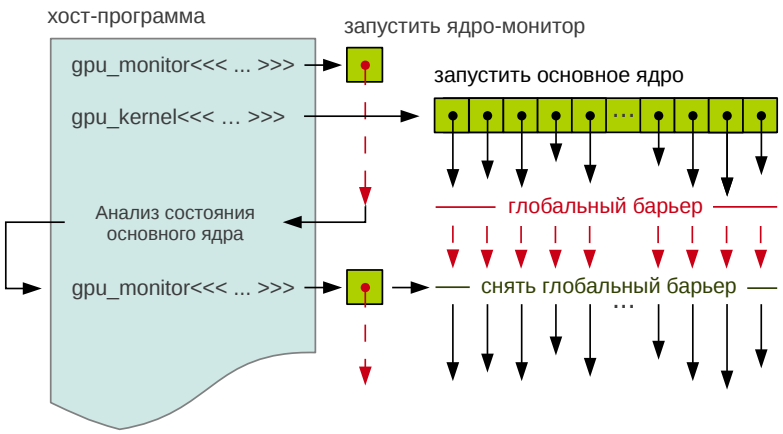

Рис. 3. Схема системы активной синхронизации

на GPU и одноимённого вызова в CPU-коде, который выполняет, соответственно, окончательную генерацию кода и запуск вычислительного ядра или загрузку данных с GPU и запуск CPU-функции средствами Foreign Function Interface (FFI). Взаимодействие между часятми может быть организовано посредством глобальной памяти GPU или pinned-памяти хоста. Однако для гарантированной передачи корректного значения необходимо обеспечить *атомарный* режим операций чтения и записи, доступность которого является определяющим фактором. По этой причине был реализован метод, использующий глобальную память.

В глобальную память GPU помещается целый маркер с состояниями 0 и 1, начальным состоянием 1. В момент начала работы приложения запускается ядро-монитор, атомарно меняет значение маркера на 0 и ожидает его изменения на 0. Следом запускается основное ядро, которое при наступлении события заполняет данными управляющую структуру, атомарно меняет значение маркера на 1 и переходит в состояние ожидания значения 0, позволяя при этом завершиться ядру-монитору. Синхронизация при завешении ядра-монитора на стороне CPU и является сигналом начала взаимодействия: производится чтение управляющей структуры и требуемый вызов. После его окончания, снова запускается ядро-монитор, которое разблокирует основное ядро (рис. 3).

Дополнительное препятствие взаимодействию GPU-ядра с другим ядром или CPU состоит в том, что данные нити (CUDA thread) хранятся в регистрах или локальной памяти. Это означает, что аргументы, переданные из основного GPU-ядра не могут быть использованы где-либо, кроме как в нём самом. Для преодоления этого ограничения, код компилятора *nvopencc* был дополнен реализацией опции $-CG : auto\_as\_static = 0$. В таком режиме компиляции локальные переменные помещаются не в .local-секцию, а в .global, делая их доступными всем GPU-ядрам и хосту.

3. Генерация CUDA-ядер для параллельных циклов с помощью LLVM/Polly

Polly [4] (от polyherdal analysis – выпуклый анализ) – это оптимизирующее преобразование циклов, основанное на CLooG и инфраструктуре LLVM. Оно способно распознавать параллельные циклы, оптимизировать кеширование за счёт добавления блочности, оптимизировать доступ к памяти за счёт перестановки циклов и генерировать код, использующий OpenMP. Для заданного кода CLooG строит *абстрактное синтаксическое дерево* (AST), проводя расщепление циклов по некоторым измерениям. Благодаря возможности расщепления частично-параллельных измерений, для исходного цикла может быть найдено экви-

валентное представление из одного или нескольких циклов, часть которых параллельна. Подобный подход используется довольно редко, большинство современных компиляторов ограничиваются проверкой параллельности измерений существующих циклов без глубокого анализа.

Элементами AST являются *статические части потока управления* (static control parts – SCoPs) – части программы, в которых поток управления и шаблоны доступа к памяти могут быть вычислены во время компиляции. Часть программы представляет собой SCoP при выполнении следующих условий:

1. Единственными операторами управления являются циклы-счётчики (for) и условные операторы.

2. Какждый цикл-счётчик имеет только одну целочисленнную индексную переменную, которая изменяется в теле цикла с константным шагом. Верхняя и нижняя границы цикла заданы афинными выражениями, зависящими от параметров и индексных переменных внешних циклов, где под параметрами понимаются любые целочисленные переменные, не изменяющиеся внутри SCoP.

3. Условные операторы сравнивают только значения двух афинных выражений.

4. Помимо управлющих конструкций присутствуют только операторы присваивания, выражения и индексный доступ к массивам. Выражениями могут быть операторы или вызовы функций без побочных эффектов, аргументами которых являются параметры, индексные переменные или элементы массивов.

5. Обращения к массивам проводятся по индексам, являющимися афинными выражениями от параметров и переменных.

При генерации LLVM IR высокоуровневые конструкции исходных языков преобразуются в последовательности низкоуровневых инструкций. Так, циклы описываются условными переходами, формирующими эквивалентный поток управления, обращения к массивам выражаются адресной арифметикой, афинные выражения расщепляются на последовательности трёхадресных инструкций. Для восстановления высокоуровневой информации из промежуточного представления Polly использует средства LLVM. Преимуществом такого подхода является возможность извлечения высокоуровневой информации из низкоуровневой, даже если она присутствует в высокоуровневом коде программы лишь неявно. Другими словами, Polly может оптимизировавать не только циклы, присутствующие в явном виде, но и любой код семантически эквивалетный циклам, например программы на языке Fortran, использующие goto.

В KernelGen из выделенных на этапе компиляции функций с тесно-вложенными циклами средствами LLVM/Polly выбираются параллельные по одному и более измерениям. С помощью класса $polly :: clastExpGen$ для каждого цикла генерируется код расчёта числа итераций, используемый при задании вычислительной сетки ядра. К исходным циклам также применяются стандартные упрощающие преобразования LLVM и пространственно-временные оптимизации Polly. Данные этапы могут быть применены как на этапе компиляции, так и во время исполнения приложения. В первом варианте можно сразу же провести оптимизацию и отбросить циклы, определённые как непараллельные. В то же время, второй вариант позволяет выполнить дополнительную подстановку значений скалярных переменных из контекста исполнения, что в некоторых случаях улучшает качество анализа за счёт того, что неафинные выражения становятся афинными.

В Polly имеется генератор кода для OpenMP. Если внешний цикл является параллельным, то его содержимое перемещается в отдельную функцию, с добавлением вызовов функций библиотеки libgomp – GNU релизацией OpenMP. При этом распределение итераций по

ядрам производит среда выполнения и распараллеливается только внешний цикл. Адаптация этого метода для генерации GPU-ядер потребовала следующих изменений:

1. Отображение пространства итераций на нити GPU, с учётом необходимости объединения запросов в память нитей варпа (coalescing transaction).

2. Рекурсивная обработка вложенных циклов с целью использования возможностей GPU по созданию многомерных сеток нитей.

Пусть в заданной группе циклов можно распараллелить N тесно-вложенных циклов. Тогда ядро может быть запущено на решётке с числом измерений N (для CUDA $N \leq 3$). Для каждого измерения, распределяемого между нитями GPU, генерируется код, рассчитывающий положение нити в блоке и блока в сетке. Каждому параллельному циклу ставится во взаимно однозначное соответствие измерение решетки, причём в обратном порядке – внутреннему циклу соответствует измерение X (это позволяет объединять запросы в память). Для каждого параллельного цикла генерируется код, определяющий нижнюю и верхнюю границы части пространства итераций, которая должна быть выполнена нитью. Затем генерируется последовательный код цикла с изменёнными границами и шагом.

4. Заключение

В настоящее время компилятор KernelGen способен производить описанные этапы преобразования и компилировать тестовые программы в корректный PTX-ассемблер. Прежде чем приступить к тестированию более сложных приложений и исследованию производительности, необходимо реализовать общий метод синхронизации данных CPU и GPU, оценочную функцию для переключения между параллельными и последовательными версиями циклов, а также систему сравнения результатов с контрольными версиями циклов. Исходный код системы доступен на сайте проекта: `http://hpcforge.org/projects/kernelgen/`.

Литература

1. The OpenACC^{TM}Application Programming Interface. Version 1.0, November, 2011, `http://www.openacc-standard.org`.

2. Govett M. Development and Use of a Fortran -> CUDA translator to run a NOAA Global Weather Model on a GPU cluster. // Path to Petascale: Adapting GEO/CHEM/ASTRO Applications for Accelerators and Accelerator Clusters. April 2-3, 2009, National Center for Supercomputing Applications. University of Illinois at Urbana-Champaign.

3. Mikushin D. KernelGen – naïve GPU kernels generation from Fortran source code.. COSMO General Meeting, September, 2011, Rome.

4. Grosser T., Zheng H., Aloor R., Simbürger A., Größlinger A., Pouchet L.-N. Polly – Polyhedral Optimization in LLVM. IMPACT 2011 (at CGO 2011), Charmonix France, April 2011.

5. Kravets A., Monakov A., Belevantsev A. GRAPHITE-OpenCL: Automatic parallelization of some loops in polyhedra representation. // GCC Developers' Summit, GCC Developers' Summit. October 25-27, 2010, Ottawa, Canada.

6. Verdoolaege S., et al. PPCG – C to CUDA processor

7. Bastoul C. Code Generation in the Polyhedral Model Is Easier Than You Think. // PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques. September, 2004, Juan-les-Pins, France.

8. Murphy M. NVIDIA's Experience with Open64. // Open64 Workshop at CGO 2008, April 6, 2008, Boston, Massachusetts.

9. Open64 compiler.

10. Chris Lattner LLVM: An Infrastructure for Multi-Stage Optimization. Masters Thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

11. Sands D. Reimplementing llvm-gcc as a gcc plugin. // Third Annual LLVM Developers' Meeting. October 2, 2009, Apple Inc. Campus, Cupertino, California.

12. Wolfe M., Toepfer C. The PGI Accelerator Programming Model on NVIDIA GPUs Part 3: Porting WRF.

13. Jeff Squyres J., Bosilca G., Sumimoto S., vandeVaart R. Open MPI State of the Union. // Open MPI Community Meeting, Supercomputing 2011.