KernelGen – прототип распараллеливающего компилятора C/Fortran для GPU NVIDIA на основе технологий LLVM*

Н.Н. Лихогруд[1], Д.Н. Микушин[2]

[1]Факультет Вычислительной математики и кибернетики
Московского Государственного Университета им. М.В. Ломоносова,
[2]Institute of Computational Science, Università della Svizzera italiana

Проект KernelGen (`http://kernelgen.org/`) имеет цель создать на основе современных открытых технологий компилятор Fortran и C для автоматического портирования приложений на GPU без модификации их исходного кода. Анализ параллелизма в KernelGen основан на инфраструктуре LLVM/Polly и CLooG, модифицированной для генерации GPU-ядер и alias-анализе времени исполнения. PTX-ассемблер для GPU NVIDIA генерируется с помощью бекенда NVPTX. Благодаря интеграции LLVM-части с GCC с помощью плагина DragonEgg и модифицированного линковщика, KernelGen способен, при полной совместимости с компилятором GCC, генерировать исполняемые модули, содержащие одновременно CPU- и GPU-варианты машинного кода. В сравнительных тестах с OpenACC-компилятором PGI KernelGen демонстрирует бо̀льшую гибкость по ряду возможностей, обеспечивая при этом сравнимый или незначительно более высокий уровень производительности.

1. Введение

Широкое использование GPU в кластерных вычислительных системах требует массовой адаптации множества сложных приложений. Программные модели CUDA и OpenCL достаточно хорошо подходят 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. Многие компании и научные группы по-прежнему откладывают портирование своих приложений, так как это приводит к возрастанию издержек на сопровождение и развитие нескольких различных версий одной и той же функциональности для CPU и GPU. Увеличить эффективность портирования призваны следующие типы технологий разработки:

- **Директивные расширения существующих языков высокого уровня с ручным управлением параллелизмом, по аналогии с OpenMP.** Данный тип технологий позволяет автоматически преобразовывать исходный код программы в гибридный CPU-GPU код, основываясь на заданных пользователем управляющих директивах. Для стандартизации набора директив в языках C/C++/Fortran коммерческими разработчиками подобных решений созданы консорциумы OpenACC [8] и OpenHMPP [9]. Аналогичный набор директив, но уже для ускорителей архитектуры Many Integrated Core (MIC) развивается компанией Intel [10]. В рамках систем F2C-ACC [11] и САПФОР [12] предложены наборы директив для преобразования исходного кода на языке Fortran в гибридную форму, причём САПФОР проводит распараллеливание как на уровне GPU, так и на уровне нескольких GPU-узлов, объединённых в MPI-кластер.

*Работа поддержана грантом HP2C (`hp2c.ch`), тестирование велось на оборудовании компании NVIDIA, кластере «Ломоносов» МГУ им М.В. Ломоносова и кластере «Tödi» Швейцарского национального суперкомпьютерного центра (CSCS).

В целом, несмотря на бо̀льшую гибкость, директивные расширения все же требуют значительного участия программиста в организации корректных и эффективных вычислений. Компиляторы некоторых из приведённых директивных расширений реализуют проверку параллельности циклов и непротиворечивости директив, в других такая проверка отсутствует. Часто возникают ситуации, в которых компилятор слишком «осторожен» при принятии решений на основе внутреннего анализе циклов и производит распараллеливание только при наличии дополнительных указаний от пользователя. Большинство директивных расширений не поддерживают генерацию GPU-ядер для циклов, в которых присутствуют вызовы функций из других модулей компиляции или библиотек, что существенно ограничивает применимость подобных технологий в больших проектах.

- **Специализированные языки (domain-specific languages, DSL), со встроенными средствами параллелизма, ориентированные на определённый класс задач.** В последние годы было предложено множество различных DSL- и Embedded DSL-языков со встроенными средствами параллелизма. Их основная идея состоит в том, чтобы приблизить синтаксис к характерным объектам и действиям задачи, в то же время исключив из языка конструкции, привязывающие реализацию к конкретной архитектуре. Обработка возникающего нового уровня абстракции производится компилятором или source-to-source процессором, генерирующим код для всех целевых архитектур. Так, в работе [1] предложен Си-подобный DSL-язык для выражения вычислений на сетках с учетом начальных и граничных условий, а также соответствующий компилятор с бекендами для различных CPU (SSE, AVX). В работе [5] аналогичная задача решается при помощи eDSL, основанного на шаблонах C++. Поддерживается генерация кода для CPU и GPU NVIDIA. Другой eDSL на основе C++ – Halide [6], с поддержкой x86-64/SSE, ARM v7/NEON и GPU NVIDIA, нацелен на эффективную реализацию методов обработки изображений. К классу DSL/eDSL можно отнести систему Nemerle Unified Device Architecture (NUDA) [7], позволяющую создавать новые расширения языка Nemerle и соответствующие плагины для компилятора.

  Тестирование DSL/eDSL как правило проводится в сравнении с программами, написанными вручную, что не позволяет судить о том, насколько существенен может быть выигрыш в эффективности DSL-языков по сравнению с директивными расширениями. Кроме того специализация ограничивает конкурентную среду, так как у каждого языка как правило существует только один разработчик. Глубокое сравнительное тестирование затруднено необходимостью реализации бенчмарков на каждом используемом языке.

- **Автоматический анализ параллельности кода с помощью эвристик или методов многогранного анализа.** Технологии данного типа предназначены для вычисления зависимостей данных и пространств итераций с помощью точных методов или эвристик. Эвристики в настоящее время являются частью большинства коммерческих компиляторов, когда в составе открытых и экспериментальных решений можно найти более сложные методы, такие как многогранный анализ (polyhedral analysis). В работе [13] для компилятора GCC реализовано расширение для автоматической идентификации параллельных циклов и генерации для них кода на OpenCL. Аналогичное расширение PPCG для компилятора Clang (LLVM) [14] способно преобразовывать код на C/C++ в CUDA-ядра. Обе технологии преобразуют вычислительные циклы из внутреннего представления компилятора в код на OpenCL или CUDA при помощи системы многогранного анализа Chunky Loop Generator (CLooG) [15]. Source-to-source компилятор Par4all [16] преобразует код на языке C или Fortran в код CUDA, OpenCL или OpenMP с помощью системы многогранного анализа PIPS.

Явное программирование на CUDA, директивные расширения и DSL-языки в любом случае предполагают модификацию или переработку исходного кода программы. По этой причине портирование больших приложений на GPU с помощью этих технологий сильно затруднено. Если же приложение портировано лишь частично, то синхронизация данных между хостом и GPU может значительно влиять на общую производительность. Так, при портировании только одного блока WSM5 модели WRF с помощью директив PGI Accelerator, время обменов данными составляет 40-60% общего времени [21].

На основе сопоставления свойств существующих технологий с требованиями, возникающими при портировании на GPU типичного вычислительного приложения, можно выделить ряд возможностей, имеющих потенциально наиболее важную роль при планировании и разработке программных систем следующего поколения:

- Поддержка широкого множества *существующих* языков программирования;
- Автоматическая оценка параллельности вычислительных циклов, не требующая внесения изменений в исходный код или каких-либо дополнительных действий со стороны пользователя;
- Генерация кода, полностью совместимая со стандартной хост-компиляцией;
- Минимизация обмена данными между памятью системы и GPU;
- Встраивание в существующие схемы распараллеливания, в первую очередь – MPI.

Целью проекта KernelGen является создание компилятора, удовлетворяющего всем перечисленным условиям и проработка стратегии развития необходимых для этого технологий. Очевидно, что подобная система не может быть построена ни на основе директивных расширений, ни на основе DSL, в то же время в ней вполне могли бы быть использованы наработки исследовательских решений по автоматическому анализу циклов. KernelGen находится на стыке математических методов, теории компиляторов и практической реализации новой технологии. Работа такого рода требует именно смешанного исследовательского формата, поскольку развитие существующих компиляторов слишком инертно для значительных нововведений, а недостаточная ориентация на практику ведёт к преждевременной стандартизации методов, не отвечающим реальным потребностям.

Данная статья организована следующим образом. В разделе 2 предлагаются решения по организации процессов компиляции, линковки и генерации кода, а также нестандартная модель исполнения, позволяющая естественным образом обеспечить более эффективное взаимодействие параллельных частей кода на GPU. В части 3 излагается способ модификации существующей технологии анализа параллельности циклов для генерации GPU-кода. Разделы 4 и 5 посвящены, соответственно, необходимым дополнительным подсистемам исполнения приложений и сравнительному анализу работы тестовых задач.

2. Этапы преобразования кода

При разработке системы компиляции на основе существующих наработок значительную роль играет выбор наиболее подходящей базовой инфраструктуры по большому числу критериев: наличие фронтендов для различных языков, полнота и гибкость внутреннего представления, существование базового набора оптимизирующих преобразований и эффективных бекендов для целевых архитектур, динамика развития и поддержка со стороны сообщества разработчиков. Наиболее развиты по этим критериям компиляторы GCC, LLVM и Open64. Компилятор GCC поддерживает наибольшее число языков программирования, но не имеет бекендов для GPU, когда как LLVM и Open64 имеют бекенды для NVIDIA PTX ISA. Компилятор Open64 имеет фронтенды для C, C++ и Fortran, генерирует качественный

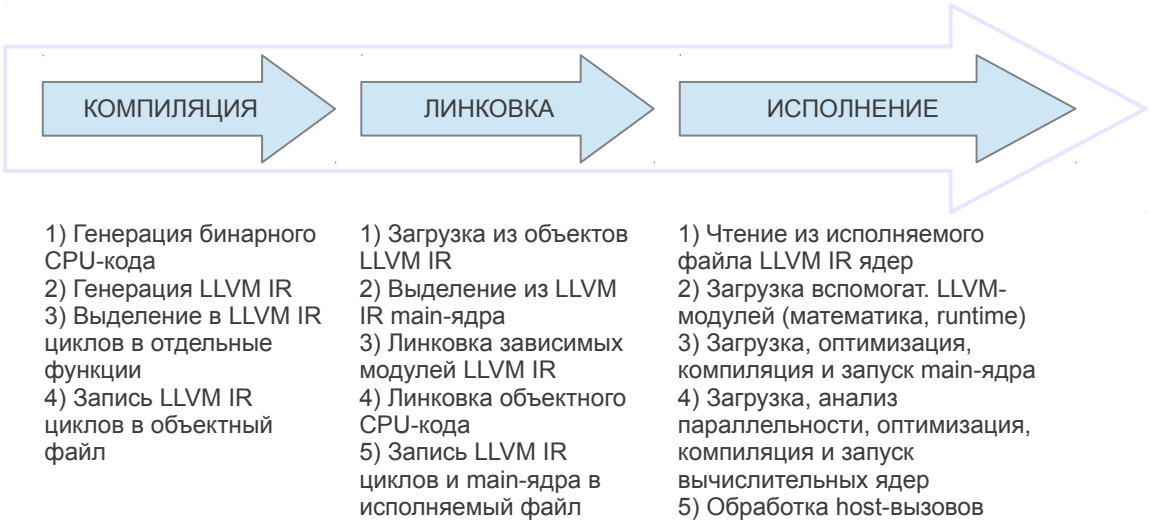

Рис. 1. Этапы преобразования кода компилятором KernelGen

код, но при этом, к сожалению, имеет сильно сегментированное сообщество разработчиков, развивающих множество отдельных веток кода в интересах коммерческих компаний и исследовательских организаций. Компилятор LLVM не имеет собственного фронтенда для языка Fortran, но способен при помощи плагина DragonEgg [17] использовать фронтенды компилятора GCC. При этом он имеет собственный GPU-бекенд NVPTX, имеет простое внутреннее представление (LLVM IR – intermediate language) и развивается намного более интенсивно, чем GCC и Open64. Из этих соображений, за основу для KernelGen был выбран LLVM.

Компилятор KernelGen работает напрямую с оригинальным приложением, не требуя каких-либо изменений ни в исходном коде, ни в системе сборки. За счёт использования фронтенда незначительно модифицированной версии GCC, он полностью совместим с его опциями, что гарантирует высокий уровень поддержки большого числа приложений. Чтобы обеспечить стандартный процесс сборки, в KernelGen используется схема, напоминающая LTO (link time optimization – инфраструктура компилятора для дополнительной оптимизации кода во время линковки): 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). Схема основных этапов преобразования кода приведена на рис. 1.

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 небольшие последовательные участки кода и производить вызовы вычислительных ядер и отдельных CPU-функций, которые невозможно или неэффективно переносить на GPU. 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 [19]. В целом, такая модель исполнения имеет много общего с native-режимом Intel MIC, но работает на GPU, где скалярные вычислительные блоки способны достигать высокой эффективности без необходимости векторизации.

2.1. 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 szargs,
    unsigned long long szargsi, unsigned int* args);
```

где name – имя или адрес функции (вместо имён в начале работы программы подставляются адреса), *args* – структура, агрегирующая аргументы вызова, *szarg* и *szargi* – размер списка аргументов и списка целочисленных аргументов (последний используется для вычисления хеша функции и поиска ранее скомпилированных ядер во время исполнения).

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

```
if (kernelgen_launch(name, szargs, szargsi, args) == -1) {
    // Launch original loop.
}
```

С помощью данного условия runtime-библиотека KernelGen может переключать выполнение между различными версиями цикла. Например, если цикл определён как непараллельный, то *kernelgen_launch* возвращает -1, и код цикла начинает выполняться основным ядром в последовательном режиме. Тем не менее, данный цикл может содержать вложенные параллельные циклы, обработка которых будет проведена аналогичным образом. В конце концов, если весь каскад тесно вложенных циклов непараллелен, несовместим (например, содержит вызовы внешних CPU-функций) или оценен как неэффективный для GPU, то вся функция выгружается для работы на хосте с помощью вызова *kernelgen_hostcall*:

```
__device__ void kernelgen_hostcall(
    unsigned char* name, unsigned long long szargs,
    unsigned long long szargsi, unsigned int* args);
```

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. Хост-часть управляющих функций компилирует и выполняет заданную функцию с помощью интерфейса FFI (Foreign Function Interface).

Одним из специфических свойств KernelGen является хранение всех данных приложения в памяти GPU. Для того чтобы обеспечить его совместимость с наличием CPU-вызовов, реализована простая система синхронизации памяти. При попытке CPU-функции обратиться к памяти по адресу из диапазона GPU возникающий сигнал сегментации обрабатывается дублированием страниц из памяти GPU в страницы CPU-памяти, расположенные по тем же адресам. После завершения работы CPU-функции, изменённые CPU-страницы синхронизируют изменения с памятью GPU.

2.2. Линковка

Во время линковки отдельных объектов в приложение или библиотеку, LLVM IR также линкуется в один общий IR-модуль для main-ядра и по одному IR-модулю на каждый вычислительный цикл. IR-код погружается в исполняемый файл и в дальнейшем оптимизируется и компилируется в GPU код по мере необходимости во время работы приложения.

Специальной обработки требуют глобальные переменные. Синхронизация глобальных переменных между ядрами потребовала бы разработки для GPU динамического линковщика. Вместо этого, в начале работы программы на CPU передаются адреса всех глобальных переменных. Во время выполнения, виртуальные глобальные переменные заменяются на соответствующие фактические адреса. Это корректно, т.к. в LLVM глобальная переменная реализована как указатель на память, содержащую её логическое значение.

2.3. 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 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 частями 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.

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, бекенд NVPTX изменён так, чтобы 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

Частью инфраструктуры LLVM является библиотека Polly [3] (от polyhedral analysis – многогранный анализ) – оптимизирующее преобразование циклов, основанное на CLooG. Оно способно распознавать параллельные циклы в IR-коде, 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), а затем проводит расщепление циклов по некоторым измерениям. Благодаря возможности расщепления частично-параллельных измерений, для исходного цикла может быть найдено эквивалентное 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.

Polly работает с частями программы, для которых можно статически (без выполнения) предсказать поток управления и доступ в память в зависимости от фиксированного набора параметров. Такие части принято называть *статическими частями потока управления* (static control parts – SCoPs). Часть программы представляет собой SCoP при выполнении следующих условий:

Рис. 2. Этапы генерации CUDA-ядер для параллельных циклов компилятором KernelGen. Оптимизация происходит сразу для всего SCoP, генерация – для каждой функции в отдельности

1. Поток управления формируется условными операторами и циклами-счётчиками;
   (a) Каждый цикл-счётчик имеет одну индексную переменную с константным шагом изменения, верхняя и нижняя границы цикла заданы афинными выражениями, зависящими от параметров и индексных переменных внешних циклов;
   (b) Условные операторы сравнивают значения двух афинных выражений, зависящих от параметров SCoP и индексных переменных;
2. Обращения в память происходят со смещениями от указателей-параметров SCoP. Смещения задаются афинными выражениями от параметров SCoP и индексных переменных циклов.
3. Содержит вызовы только функций без побочных эффектов

Первое условие означает структурированность потока управления: код можно логически разбить на иерархию вложенных блоков, имеющих один вход и один выход, каждый блок полностью вложен в объемлющий. Запрещены конструкции, нарушающие структуру потока управления (*break*, *goto*). Использование афинных выражений позволяет применять аппарат целочисленного программирования для расчёта границ циклов и обращений в память в зависимости от параметров SCoP.

Если работать с высокоуровневым представлением программы (код на языке высокого уровня, абстрактное синтаксическое дерево), то множество приёмов программирования (например, арифметика указателей, циклы *while*, операторы *goto*) будут нарушать описанные требования. Если же перейти к промежуточному, близкому к ассемблеру представлению, то арифметика указателей будет реализована как набор арифметических операций с регистрами, и любой цикл, вне зависимости от типа (*for*, *while*), будет реализован как обычный условный переход. Следствием этого являются две полезные возможности KernelGen:

- распараллеливать *while*-циклы, когда как, например, стандартом OpenACC такая возможность не предусмотрена;
- распараллеливать циклы с адресной арифметикой, что, например, не поддерживается в PGI OpenACC

При адаптации Polly для получения GPU-ядер был использован существующий генератор кода OpenMP, работающий следующим образом. Если внешний цикл является параллельным, то его содержимое перемещается в отдельную функцию, с добавлением вызовов функций библиотеки libgomp – GNU реализации OpenMP. При этом распределение итераций по ядрам производит среда исполнения, а распараллеливается только самый внешний цикл. Для KernelGen в эту логику были внесены следующие изменения:

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.

Схема этапов работы Polly, анализа и генерации кода приведена на рис. 2.

4. Дополнительные средства времени исполнения

Включение бекенда NVPTX для генерации GPU-кода в LLVM позиционировалось компанией NVIDIA как «открытие компилятора». Тем не менее, помимо того, что закрытым остаётся C/C++/CUDA-фронтенд, а clang обладает лишь минимальной поддержкой некоторых ключевых слов CUDA, недоступной также остаётся часть компилятора существенно важная для его применения в LLVM: библиотека математических функций C99. Поскольку в рамках закрытого компилятора CUDA эти функции реализованы в виде заголовочных файлов C/C++, их использование с другими языками на уровне LLVM невозможно, и пользователю NVPTX-бекенда доступны только функции, встроенные в аппаратуру (builtins), среди которых, например, нет точных версий функций sin, cos, pow. В KernelGen данная проблема решена путём конвертации заголовочных файлов в LLVM IR одним из двух способов: с помощью clang (требуется множество модификаций, полученный IR-код предположительно содержит некорректные части) или с помощью cicc (вызывается из nvcc). В последнем случае, IR-модуль можно получить, отправив на компиляцию пустой .cu-файл и выгрузив код IR-модуля из cicc с помощью отладчика. IR, сгенерированный cicc совместим с актуальной версией LLVM и позволяет производить линковку математической библиотеки и использующего её приложения на уровне IR-кода, вне зависимости от начального языка. В частности, благодаря этому KernelGen позволяет генерировать GPU-код для программ на языке Fortran, использующих любые стандартные математические функции.

Некоторые типы функций CUDA API, такие как выделение GPU-памяти и загрузка другого CUDA-модуля, всегда приводят к неявной синхронизации асинхронных операций. Поскольку схема работы KernelGen требует постоянного поддержания основного ядра в состоянии выполнения, необходимость выделения памяти или загрузки новых ядер в процессе его работы приведёт к блокировке. В этом отношении существующие версии CUDA создают для развития KernelGen определённые препятствия, вынуждая реализовывать нестандартные эквиваленты базовой функциональности.

Если синхронность выделения памяти на GPU со стороны хоста ещё можно считать разумным ограничением, то подтверждённая экспериментами синхронность вызовов *malloc*

внутри GPU-ядер явно избыточна, так как память для индивидуальных потоков выделяется заранее. Так как оба стандартных варианта не могут быть использованы, KernelGen выполняет начальную преаллокацию памяти для собственного динамического пула и управляет его работой.

JIT-компиляция вычислительных ядер предполагает, что вновь скомпилированные GPU-ядра будут загружаться на GPU в фоне работающего основного ядра. Обычно динамическую загрузку ядер можно произвести с помощью стандартных функций *cuModuleLoad* и *cuModuleGetFunction* CUDA Driver API. Однако, обе эти функции являются синхронными, предположительно, из-за неявного выделения памяти для хранения кода и статических данных. В данной ситуации при разработке KernelGen не оставалось иного выбора, кроме как реализовать загрузчик кода новых ядер вручную, предварительно создав для них пустую функцию-контейнер. Загрузчик основан на технологиях проекта AsFermi [4] и действует следующим образом. В начале работы приложение на GPU загружается достаточно больше пустое ядро (содержащее инструкции NOP). По мере того, как в процессе работы приложения требуется запускать вновь скомпилированные ядра, их код копируется как данные в адресное пространство контейнера, которое известно благодаря инструкции LEPC (получить значение Effective Program Counter). Контейнер размещает код множества небольших ядер друг за другом, создавая своеобразный динамический пул памяти для кода. При этом необходмо учитывать, что различные ядра могут использовать различное число регистров. Для этого загрузчик создаёт 63 фиктивных ядра (точки входа), использующих от 1 до 63 регистров с единственной инструкцией JMP для перехода по адресу начала требуемого ядра в контейнере.

Система синхронизации памяти между GPU и хостом использует вызов *mmap*, ограничивающий возможные диапазоны адресов величинами, кратными размеру страниц (4096 байт). Поэтому выравнивание всех данных GPU по границе 4096 было бы очень удобным упрощением на данном этапе. К сожалению, текущая реализация CUDA (5.0) учитывает настройки выравнивания данных при компиляции, но при этом игнорирует их во время исполнения. Обход этого дефекта реализован посредством выравнивания размеров всех данных по границе 4096 вручную с помощью функций библиотеки *libelf*.

5. Тестирование

KernelGen тестируется на трех типах приложений: тесты корректности, тесты производительности и работа на реальных задачах. Тесты корректности предназначены для контроля регрессивных изменений в генераторе кода, тесты производительности позволяют анализировать эффективность текущей версии KernelGen в сравнении с предыдущими сборками и другими компиляторами. При тестировании производительности предпочтение отдаётся сравнению с результатами других распараллеливающих компиляторов, поскольку в отличие от сравнения с кодом, оптимизированным вручную, это позволяет проанализировать достоинства и недостатки компилятора в своём классе систем.

Для тестирования были выбраны приложения, реализующие различные типовые алгоритмы на двумерной или трехмерной регулярной сетке с одинарной точностью (часть тестов адаптировано из материалов работы [2], описание и исходный код приведены в [22]). На рис. 3 показано насколько меняется скорость работы тестов, скомпилированных с помощью KernelGen по сравнению с версией PGI OpenACC. Соответствующие абсолютные времена и число регистров для каждой версии теста приведены в Табл. 1. Тесты *jacobi*, *matmul* и *sincos* реализованы на языке Fortran, остальные тесты – на C (также были проверены реализации тестов *wave13pt* и *laplacian* на языке C++, однако для данного сравнения они не пригодны, поскольку компилятор PGI не поддерживает директивы OpenACC в C++). KernelGen автоматически распознаёт наличие вложенных параллельных циклов внутри непараллельного цикла по числу итераций, когда как PGI делает это только при со-

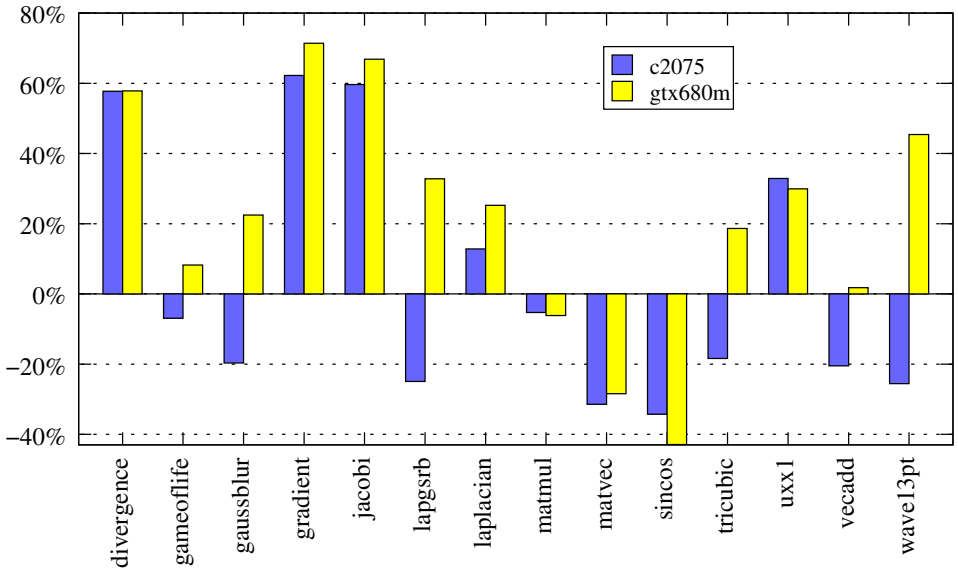

Рис. 3. Сравнение производительности вычислительных ядер некоторых тестовых приложений, скомпилированных KernelGen r1578 и PGI 12.10 на GPU NVIDIA Tesla C2075 (Fermi sm_20) и GTX 680M (Kepler sm_30)

ответствующей ручной расстановке директив OpenACC. Более низкая производительность KernelGen на тесте *matmul* обусловлена тем, что компилятор PGI реализует частичную раскрутку внутреннего цикла с редукцией на регистрах. Более низкая производительность ряда других тестов требует отдельного изучения.

Тестирование на больших вычислительных приложениях COSMO [20] и WRF [21] показало, что KernelGen способен генерировать корректные исполняемые файлы с поддержкой GPU за разумное время.

6. Заключение

В проекте KernelGen реализована оригинальная схема автоматического портирования кода на GPU, подходящая для сложных приложений. Не требуя никаких изменений в исходном коде, компилятор переносит на GPU максимально возможную часть кода, включая выделение памяти, тем самым создавая эффективную схему для преимущественно GPU-вычислений. KernelGen реализует средства автоматического анализа параллелизма циклов, основанные на LLVM, Polly и других проектах, расширяя их поддержкой генерации кода для GPU. Генератор GPU-кода основан на NVPTX-бекенде для LLVM, совместно развиваемом компанией NVIDIA и силами сообщества LLVM. Тестирование показало, что GPU-код, генерируемый KernelGen по своей эффективности сравним с коммерческим компилятором PGI.

Для того чтобы начать использовать компилятор в прикладных задачах, остается реализовать некоторые функциональные элементы. В частности, в нынешней версии отсутствует механизм накопления статистики эффективности исполнения GPU-кода для принятия решений о переключении на CPU-реализацию в неэффективных случаях. В генераторе параллельных циклов желательно добавить возможность использования разделяемой памяти и распознавание в коде идиомы редукции. Запуск вычислительных ядер на архитектуре Kepler может быть организован более эффективно за счёт использования динамического параллелизма.

Код KernelGen распространяется по лицензии University of Illinois/NCSA (за исключением плагина для GCC) и доступен на сайте проекта: `http://kernelgen.org/`.

Таблица 1. Сравнение времени исполнения (сек) и числа регистров (nregs) для вычислительных ядер некоторых тестовых приложений, скомпилированных KernelGen r1578 и PGI 12.10 на GPU NVIDIA Tesla C2075 (Fermi sm_20) и GTX 680M (Kepler sm_30)

Тест	NVIDIA Tesla C2075				NVIDIA GTX 680M			
	KernelGen		PGI		KernelGen		PGI	
	время	nregs	время	nregs	время	nregs	время	nregs
divergence	0.010920	18	0.017224	36	0.009811	20	0.015487	48
gameoflife	0.011383	21	0.010597	27	0.014631	21	0.015831	27
gaussblur	0.016835	56	0.013521	34	0.020240	51	0.024789	40
gradient	0.012687	21	0.020579	35	0.009314	22	0.015964	47
jacobi	0.009008	24	0.014380	26	0.007355	23	0.012274	31
lapgsrb	0.034975	55	0.026247	63	0.019294	40	0.025616	63
laplacian	0.009970	18	0.011246	32	0.008415	22	0.010537	48
matmul	0.001514	13	0.001434	33	0.001631	14	0.001531	37
matvec	0.049047	12	0.033639	27	0.062920	16	0.045024	27
sincos	0.012112	22	0.007962	25	0.009351	22	0.005341	29
tricubic	0.074085	60	0.060450	63	0.086248	61	0.102335	63
uxx1	0.024248	32	0.032225	53	0.019377	32	0.025174	62
vecadd	0.006269	12	0.004983	28	0.005046	12	0.005135	36
wave13pt	0.025364	34	0.018885	63	0.013564	34	0.019723	63

Литература

1. M. Christen, “Generating and Auto-Tuning Parallel Stencil Codes,” Ph.D. dissertation, University of Basel, Switzerland, 2011.

2. M. Christen, O. Schenk, Y. Cui, PATUS for Convenient High-Performance Stencils: Evaluation in Earthquake Simulations. SC ’12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.

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

4. Y. Hou et al, “AsFermi: An assembler for the NVIDIA Fermi Instruction Set,” `http://code.google.com/p/asfermi/` (дата обращения 03.12.2012).

5. T. Gysi, “HP2C Dycore,” Presentation, `http://mail.cosmo-model.org/pipermail/pompa/attachments/20120306/079fadc1/DWD_HP2C_Dycore_120305.pdf`, *Workshop on COSMO dynamical core rewrite and HP2C project*, March 2012, Offenbach, Germany.

6. J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, F. Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines, SIGGRAPH 2012.

7. А.В. Адинец, Программирование графических процессов при помощи расширяемых языков. // Вестник Южно-Уральского государственного университета. Серия: Математическое моделирование и программирование, т. 25 (242), с. 52-63, 2011.

8. The OpenACC^{TM}Application Programming Interface. Version 1.0, November, 2011, `http://www.openacc-standard.org` (дата обращения 03.12.2012).

9. OpenHMPP, New HPC Open Standard for Many-Core, http://www.openhmpp.org/en/OpenHMPPConsortium.aspx (дата обращения 03.12.2012).

10. The Heterogeneous Offload Model for Intel® Many Integrated Core Architecture, http://software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf (дата обращения 03.12.2012).

11. M. Govett, "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*, Presentation, http://gladiator.ncsa.uiuc.edu/PDFs/accelerators/day2/session3/govett.pdf, April 2009, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.

12. В.А. Бахтин, Н.А. Катаев, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Автоматическое распараллеливание Фортран-программ на кластер с графическими ускорителями // Труды международной научной конференции ПаВТ'2012, Новосибирск, с. 373-379, 2012.

13. A. Kravets, A. Monakov, A. Belevantsev, "GRAPHITE-OpenCL: Automatic parallelization of some loops in polyhedra representation," *GCC Developers' Summit,* October 2010, Ottawa, Canada.

14. S. Verdoolaege et al, "PPCG – C to CUDA processor," http://repo.or.cz/w/ppcg.git (дата обращения 03.12.2012).

15. C. Bastoul, "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.

16. M. Torquati, M. Venneschi, M. Amini, S. Guelton, R. Keryell, V. Lanore, F.-X. Pasquier, M. Barreteau, R. Barrère, C.-T. Petrisor, É. Lenormand, C. Cantini, F. De Stefani. An innovative compilation tool-chain for embedded multi-core architectures. Embedded World Conference 2012. Nuremberg, Germany, 2/2012.

17. D. Sands, "Reimplementing llvm-gcc as a gcc plugin," *Third Annual LLVM Developers' Meeting,* Presentation, http://llvm.org/devmtg/2009-10/Sands_LLVMGCCPlugin.pdf, October 2009, Apple Inc. Campus, Cupertino, California.

18. M. Wolfe, C. Toepfer, "The PGI Accelerator Programming Model on NVIDIA GPUs Part 3: Porting WRF," http://www.pgroup.com/lit/articles/insider/v1n3a1.htm (дата обращения 03.12.2012).

19. J. Squyres, G. Bosilca, S. Sumimoto, R. vandeVaart, "Open MPI State of the Union," *Open MPI Community Meeting*, Presentation, http://www.open-mpi.org/papers/sc-2011/Open-MPI-SC11-BOF-1up.pdf, Supercomputing 2011.

20. Consortium for Small-scale Modeling, http://www.cosmo-model.org/ (дата обращения 03.12.2012).

21. The Weather Research & Forecasting Model, http://www.wrf-model.org/index.php (дата обращения 03.12.2012).

22. KernelGen Performance Test Suite, https://hpcforge.org/plugins/mediawiki/wiki/kernelgen/index.php/Performance_Test_Suite (дата обращения 27.01.2013).