УДК 004.272.3

ИССЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ РУЧНОЙ И АВТОМАТИЧЕСКОЙ ОПТИМИЗАЦИИ ТИПОВЫХ АЛГОРИТМИЧЕСКИХ СТРУКТУР ДЛЯ ГРАФИЧЕСКИХ УСКОРИТЕЛЕЙ

П.А. Швец, А.В. Адинец

НИВЦ Московского госуниверситета им М.В. Ломоносова

shvets.pavel@gmail.com

Поступила в редакцию 30.11.2012

Статья посвящена реализации набора типовых алгоритмических структур на графических ускорителях и исследованию способов их оптимизаций. Представлены результаты по сравнению эффективности различных методов оптимизации для различных ядер на графических ускорителях от NVIDIA и AMD. Для математических ядер проведено исследование расчётов с одинарной и двойной точностью.

Ключевые слова: графические ускорители, параллельные вычисления, оптимизация.

Введение

В последнее время часто для решения вычислительных задач используются программируемые ускорители. В их число входят графические процессорные устройства (ГПУ), а также кластерные системы на их основе, что в совокупности составляет класс неоднородных (гетерогенных) вычислительных систем. Наиболее перспективными в настоящее время представляются гетерогенные системы на основе ГПУ, что определяет актуальность задачи исследования и анализа производительности таких программ, выявления в них узких мест, определения потенциала для оптимизации и тонкой настройки.

Архитектура графических ускорителей сильно отличается от традиционных процессоров, и для написания высокоэффективных приложений необходимо не только понимать архитектуру, но и уметь предсказывать эффективность разных программных конструкций, учитывая особенности работы графического ускорителя. Многие приёмы программирования и алгоритмы решения типовых задач на центральных процессорах при переносе на графический ускоритель будут работать крайне неэффективно. Для программирования графических ускорителей необходимо модифицировать такие алгоритмы и это является основной сложностью при изучении технологий GPGPU. Для обретения этого навыка желательно иметь представление не только об особенностях архитектуры, но и знать те подходы, решения, которые будут эффективно работать на графическом ускорителе.

Целью настоящей работы является реализация базовых и оптимизированных версий типовых алгоритмических структур с использованием различных методов оптимизации (даже если они не дают выигрыша в производительности) и дальнейшая систематизация эффективности использованных методов.

Существует похожая работа от разработчиков AMD [1]. Они рассматривают другой набор задач и другой набор более низкоуровневых оптимизаций, в то время как настоящая работа концентрируется на своих методах оптимизации и на использовании аннотаций для автоматического подбора параметров этих оптимизаций.

Остальные похожие системы не в полной мере реализуют требуемый функционал. Набор тестов SHOC [2] является переносимым набором бенчмарков, но он исследует другие свойства программ и обладает меньшим набором вычислительных ядер. Другие же, например [3], подробно исследуют задачи, но с использованием других методов.

Классификация типовых алгоритмических структур

Для классификации алгоритмических структур была использована довольно известная система «13 вычислительных гномов» [4], основанная на более ранней системе «7 вычислительных гномов». Эта система основывается на шаблоне вычислений и коммуникаций алгоритмов и позволяет классифицировать подавляю-

Работа выполнена при финансовой поддержке Федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009–2013 гг., госконтракт №14.B37.21.0176.

щее большинство задач. Она не обладает полнотой (например, сложно отнести некоторые алгоритмы сортировки к какому-то классу), но может служить хорошей начальной точкой.

На основе этой системы были выбраны следующий задачи:

1. Вычисление автокореляционной функции.
2. Свёртка изображений.
3. Вычисление кулоновских потенциалов на решётке.
4. Обход графа в ширину.
5. Вычисление гистограммы.
6. Итерационный метод Якоби.
7. Умножение матриц.
8. Алгоритм шифрования AES.
9. Вычисление MD5 хешей.
10. Гравитационная задача N-тел.
11. Задача N ферзей.
12. Алгоритм Смита–Ватермана.
13. Умножение разреженной матрицы на вектор.
14. Моделирование работы сети Петри.
15. Вычисление интеграла функции методом Монте-Карло.

Этот список задач покрывает все классы «гномов», кроме двух: конечные автоматы и задачи на неструктурированной сетке. Задачи на моделирование конечных автоматов обладают недостаточным ресурсом параллелизма для исполнения на ГПУ, а из алгоритмов на неструктурированной сетке не удалось подобрать подходящей демонстрационной задачи для выполнения на одном ГПУ.

Для математических ядер проведено исследование расчётов с одинарной и двойной точностью, также для некоторых задач были проверены разные реализации и подходы, такие как: упаковка данных, разные стратегии хранения структур и пр.

Средства разработки

Разработка велась на языке программирования Nemerle с набором расширений NUDA [5], содержащим в себе набор классов, функций и макросов, реализующих возможность писать программы, которые будут исполнены на ГПУ с помощью частичной трансляции в OpenCL или CUDA кода. Также NUDA существенно облегчает разработку GPGPU программ, автоматически производя операции с контекстом, управление памятью, использование нескольких ГПУ и многое другое.

Большая часть расширений NUDA используется с помощью аннотаций: пользователь пишет привычный ему ЦПУ код, добавляет ключевое слово перед этим кодом (аннотацию) и получает код, который будет транслироваться особым образом средствами NUDA во время компиляции.

Методы оптимизации

Полная развёртка циклов

Полная развёртка цикла — распространённая оптимизация, применяемая для удаления или уменьшения накладных расходов, связанных с выполнением управляющих команд цикла. Для этой оптимизации необходимо знать размер цикла во время компиляции, причем компиляция происходит гораздо дольше и требует больше памяти.

Глубокая развёртка циклов

Частичная развёртка цикла не удаляет цикл, но уменьшает количество итерации за счёт выполнения нескольких итерации за один шаг цикла и не требует знания размера цикла на момент компиляции. Для оптимизации используется её модифицированная версия – глубокая развёртка цикла, которая не дублирует внутренние циклы (если они есть), а дублирует команды внутри них. Такая модификация существенно меняет профиль доступа к памяти и кешу среди соседних нитей, а также позволяет лучше использовать графические процессоры с VLIW архитектурой. Ещё одной особенностью является то, что ядра, исполняемые на графическом ускорителе, в NUDA представляются как циклы. Так что развёртка внешнего цикла носит дополнительный эффект — задействование меньшего количества нитей, но увеличенное количество работы для каждой нити.

Подбор оптимального размера блока нитей

Выполнение алгоритма (ядра) на графическом ускорителе требует указания размера блока нитей. Выбор неверного размера блока может привести к существенной потере производительности; не существует явного алгоритма, позволяющего узнать оптимальный размер блока, особенно при использовании других оптимизаций. Для определения самого оптимального размера пока возможен только полный перебор всех возможных размеров. Однако некоторые комбинации размеров блока будут заведомо плохи на определённых классах задач — их можно заранее проверить, что позволит сузить область полного перебора.

Использование локальной памяти

Локальная память гораздо быстрее глобальной, но меньше по размеру. Использование данных, расположенных в локальной памяти, может дать существенный прирост производи-

тельности и позволяет использовать быстрые локальные атомарные операции. Такая оптимизация сложна для реализации и очень сильно зависит от базового алгоритма, так что пока возможно лишь ручное её применение. Также использование локальной памяти позволяет использовать локальные атомарные операции — их «атомарность» распространяется лишь на блок потоков, но они работают гораздо быстрее глобальных атомарных операций.

Использование текстурной памяти

Текстурная память имеет другие алгоритмы хранения и кеширования в отличие от глобальной памяти, что может улучшить производительность некоторых специфичных классов задач, например обработки изображений. Прирост производительности может быть существенным для карточек прошлых поколений, но для новых карточек он не так существенен.

Изменение шаблона доступа к глобальной памяти

Глобальная память обладает одной важной особенностью – возможностью объединять последовательные запросы к памяти от нитей одного варпа (варп – это набор из 32 или 64 нитей – зависит от конкретного ГПУ) в одну транзакцию, что существенно повышает скорость работы. Для учёта этой особенности иногда приходится менять логику работы алгоритма с памятью. Например, если алгоритм работает с диагоналями матрицы, то имеет смысл хранить матрицу по диагоналям или выбирать между хранением матрицы по строкам/столбцам. Это может усложнять логику адресации, но положительно влияет на производительность работы алгоритма.

Использование различных реализаций

Разные подходы к реализации одной задачи, имеющие близкую производительность на центральном процессоре, могут показывать себя совершенно по-разному на графическом ускорителе. Например, в задаче обхода графа в ширину можно использовать очередь вершин, а можно на каждом шаге проверять все вершины. В первом случае возникают накладные расходы, связанные с реализацией очереди, но он требует меньше памяти, чем второй. Такие оптимизации сложно выделить в отдельный класс, поэтому в каждом примере они будут указываться отдельно.

Автоматизация проверки оптимизаций

Часть вышеописанных оптимизаций применяется в виде параметров или аннотаций к циклам, не меняющих сам код алгоритма. Если аннотации имеют численные параметры, то возможен их автоматизированный перебор для определения оптимальной комбинации параметров, имеющих максимальную производительность. Был реализован макрос «AutoTune» производящий такой перебор и вычисляющий максимальную производительность работы функции с указанными в ней аннотациями.

Пример использования макроса «AutoTune»

```
[AutoTune(
(block_size in vars.WARP_SIZE in vars.BLOCK_SIZE),
(dmine_outer in 1 in 8, dmine_inner in 1 in 8), (),
(dmine_outer * dmine_inner < 32),
(size :> float) / 1e9 * size * 2)]
public AutoCorrelationDmineOuterInner
(signal_in : nuarray1d[float],
   signal_out : nuarray1d[float], size : int) : void
      {
            nuwork(block_size)
            dmine(dmine_outer) nfor(thread_n
                                    in size)
      {
            mutable sum = 0;
            dmine(dmine_inner) nfor(i in size)
            sum += signal_in[i] *
            sigal_in[(thread_n + i) % size];
            signal_out[thread_n] = sum;
            }
      }
```

Макрос производит запуск шаблонной программы с разными комбинациями параметров и выводит лучшую комбинацию и производительность, достигающуюся на этой комбинации параметров.

Результаты эффективности оптимизаций

Вышеописанные методы оптимизации были протестированы на реализованном наборе ядер на графических ускорителях NVIDIA C2050 и Radeon HD5830. Эти карточки не являются равноценными, но являются полноценными представителями современных вычислителей двух разных архитектур. Целью исследования является не их сравнение, а изучение возможного прироста производительности приложений относительно производительности базовой версии.

На рис. 1 показано, насколько сильно влияет полная развёртка циклов, у которых число итераций известно во время компиляции, на производительность разных примеров. В целом выигрыша в производительности почти нет и даже наблюдается резкое падение производительности (предположительно это связано с какими-то

особенностями работы драйвера или компилятора).

На рис. 2 показано влияние развёртки внешнего цикла на ускорение оптимизированных версий примеров. Наибольший эффект достигается на видеокарточках AMD, но в некоторых случаях и на видеокарточках NVIDIA есть значительное ускорение.

Рис. 3 похож на рис. 2, но прирост производительности меньше и влияет на меньшее число

Рис. 1. Ускорение базовых версий от применения полной развёртки циклов

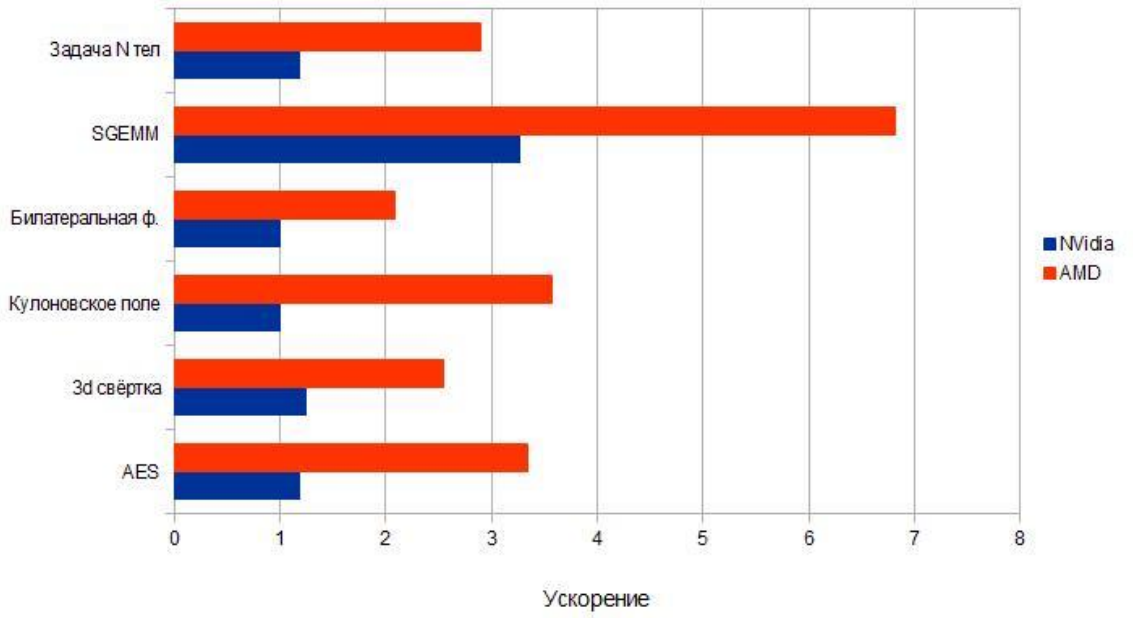

Рис. 2. Ускорение базовых версий от применения развёртки внешнего цикла

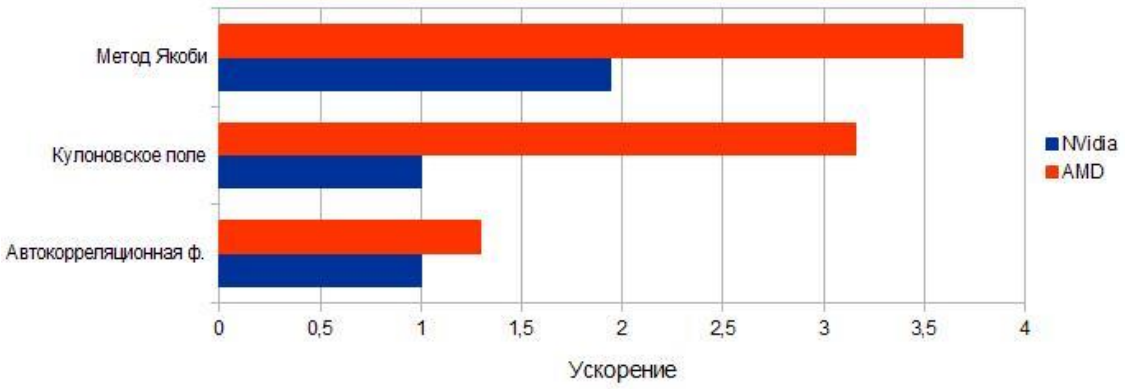

Рис. 3. Ускорение базовых версий от применения развёртки внутреннего цикла

примеров.

Рис. 4 показывает эффект от суммарного применения развёртки внешнего и внутреннего циклов – на большинстве примеров именно в этом случае удавалось достигнуть наибольшей производительности. Три вышеописанные оптимизации особенно эффективны на ускорителях AMD.

Рис. 5 демонстрирует эффективность применения локальной памяти в алгоритмах. Для её применения может быть две причины – какие-то данные часто переиспользуются в ядре или требуется использование локальных атомарных операций вместо глобальных. В этих двух случаях ускорение от использования локальной памяти может быть очень значительным.

Для изменения шаблона доступа в нельзя указать какой-то универсальный алгоритм – оно является, по сути, адаптацией исходного алгоритма под модель памяти ГПУ и должно быть сделано на этапе проектирования алгоритма. Но и разница в производительности между базовой версией и адаптированной версией может быть очень значительной, если приложение очень интенсивно общается с памятью. Результат можно видеть на рис. 6.

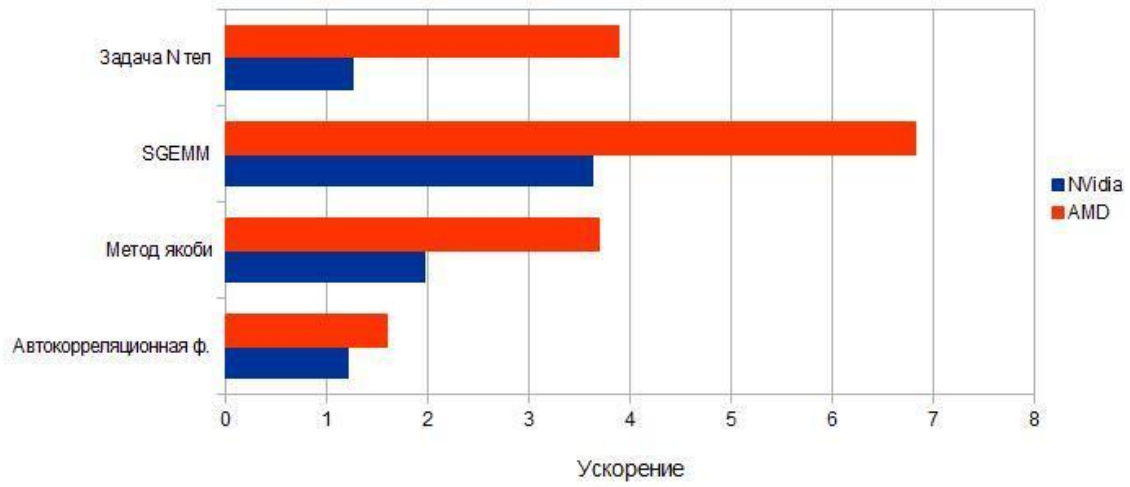

Рис. 4. Ускорение базовых версий от применение развёртки внешнего и внутреннего цикла вместе.

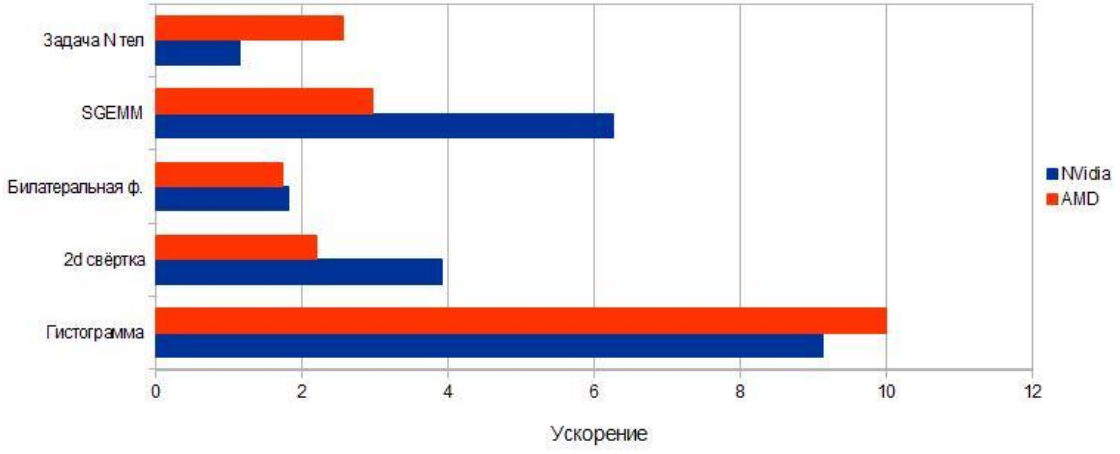

Рис. 5. Ускорение базовых версий от применение локальной памяти.

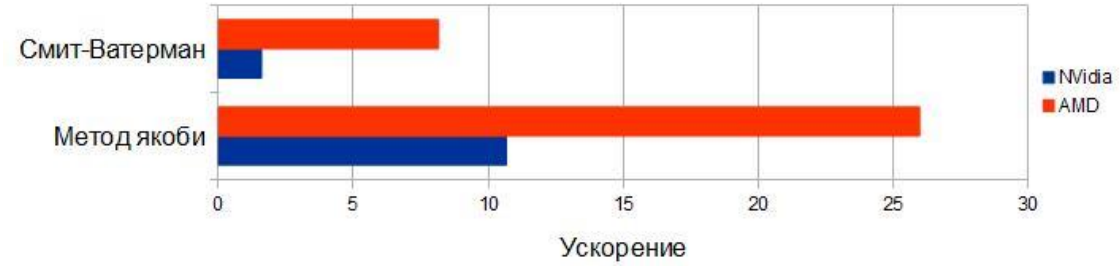

Рис. 6. Ускорение базовых версий от изменения шаблона доступа к памяти.

Список литературы

1. Feng W., Lin H., Scogland T., Zhang J. OpenCL and the 13 Dwarfs: A Work in Progress // Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering. 2012. P. 291–294

2. Danalis A., Marin G., McCurdy C. et. al. The Scalable Heterogeneous Computing (SHOC) benchmark suite

3. Manavski S., Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment // BMC Bioinformatics 2008. Vol. 9, № 2.

4. Krste A., Ras B., Bryan C. at. al The Landscape of Parallel Computing Research: A View from Berkeley // EECS Department, University of California, Berkeley. 2006. UCB/EECS-2006-183, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

5. Adinetz A.V. NUDA: Programming Graphics Processors with Extensible Languages // Nuclear Electronics & Computing. 2011.

RESEARCH OF MANUAL AND AUTOMATED OPTIMIZATION APPROACHES FOR TYPICAL ALGORITHMIC STRUCTURES FOR GRAPHICS PROCESSORS

P.A. Shvets, A.V. Adinetz

The work is devoted to implementation of a set of typical computational algorithms on GPU and research on how to optimize them. The results show an efficiency of different optimizations applied to various kernels. Kernels were tested on NVIDIA and AMD GPUs. Mathematical kernels were tested in single and double precision mode.

Keywords: graphical accelerators, parallell computing, optimizations.