УДК 004.07

Р. Ф. Гибадуллин, Д. Д. Фирсова, Н. В. Кормильцев,
А. Д. Уваров, М. Ю. Перухин

РАЗРАБОТКА И ТЕСТИРОВАНИЕ ПРОГРАММНЫХ МОДУЛЕЙ ДЛЯ ОЦЕНКИ ПРОИЗВОДИТЕЛЬНОСТИ CUDA И OPENCL ТЕХНОЛОГИЙ

Ключевые слова: CUDA, OpenCL.

CUDA и OpenCL предлагают два различных интерфейса для программирования важных в обеспечении вычислительной мощности для высокопроизводительных вычислительных приложений графических процессоров. OpenCL является открытым стандартом, который может быть использован для программирования графических процессоров и других устройств от различных производителей, в то время как CUDA является специфичной технологией программирования для графических процессоров NVIDIA. Хотя OpenCL обещает портативный язык для программирования GPU, его общность может повлечь за собой снижение производительности. OpenCL и CUDA посредством драйвера API-интерфейса или через среду выполнения API, представляющую собой набор подпрограмм и расширений Clike, вызывают фрагмент кода, который запускается на GPU ядрах. Их настройка имеет существенные отличия для представленных графических процессоров. Данные различия влияют на время, необходимое для кодирования и отладки приложения GPU. В данной статье внимание в основном сфокусировано на различиях производительности во время выполнения. Наличие у OpenCL переносимого для программирования графического процессора языка повышает способность ориентироваться на очень разнородные устройства параллельной обработки, также стоит отметить, что ядро OpenCL обладает способностью компилирования во время выполнения, что несомненно добавляет время работы для OpenCL. Напротив, CUDA, разработанная той же компанией, что и аппаратные средства, на которых она выполняется, в теории должна лучше соответствовать вычислительным характеристикам GPU, предлагая больше доступа к функциям и повышенной производительности. В этой статье мы сравниваем производительность CUDA и OpenCL с использованием сложных, почти идентичных ядер. Мы показываем, что при использовании инструментов компилятора NVIDIA преобразование ядра CUDA в ядро OpenCL требует минимальных изменений. Создание такого ядра компиляции с помощью инструментов сборки ATI включает в себя больше изменений. Наши тесты производительности измеряют и сравнивают время передачи данных с графическим процессором, время выполнения ядра и время выполнения сквозного приложения для CUDA и OpenCL.

R. F. Gibadullin, D. D. Firsova, N. V. Kormiltsev,
A. D. Uvarov, M. Yu. Perukhin

DEVELOPING AND TESTING SOFTWARE MODULES TO ASSESS THE CUDA AND OPENCL TECHNOLOGIES PERFORMANCE

Keywords: CUDA, OpenCL.

CUDA and OpenCL offer two different interfaces to program GPUs that are important in ensuring the computing power for high-performance computing applications. OpenCL is an open standard that can be used to program GPUs or other devices manufactured by various companies, while CUDA is a specific-purpose technology used in programming the NVIDIA GPUs. Although OpenCL promises a portable language to program GPUs, its commonality may lead to reducing its performance. OpenCL and CUDA use the API driver or an API execution environment representing a set of the Clike subprograms and extensions, call the code fragment that is run on GPU cores. Setting them up differs essentially for the GPUs presented. These differences affect the time required for coding and debugging the GPU application. In this paper, we are mostly focused on the differences in the performance while executing the code. The availability in OpenCL of a portable language for programming a GPU increases its ability to focus on very different parallel processing devices. It should also be noted that the OpenCL core is able to compile while executing, which undoubtedly takes some more working time for OpenCL. To the contrary, CUDA developed by the same company that has developed the hardware for it to be executed, should theoretically comply even better with the computing characteristics of GPU, offering broader access to functions and enhancing the performance. In this paper, we compare the performances of CUDA and OpenCL, using complex, practically identical cores. We show that, when using the NVIDIA compiler tools, transforming the CUDA core into the OpenCL core requires just minimum changes. Creating such a compilation core using ATI assembling tools includes more changes. Our performance tests measure and compare the data transfer time with GPU, core execution time, and the execution time of a tangled applications for CUDA and OpenCL.

Введение

Графические процессоры (GPU) стали важны в обеспечении вычислительной мощности для высокопроизводительных вычислительных приложений. CUDA [1] и Open Computing Language (OpenCL) [2] – это два интерфейса для графических процессоров, представляющих схожие функции, но с помощью различных интерфейсов программирования. CUDA – это проприетарный API и набор языковых расширений, которые работают только на графических процессорах NVIDIA. OpenCL от Khronos Group, является открытым стандартом для параллельного программирования с использованием центральных процессоров, графических процессоров, цифровых сигнальных

процессоров (DSP) и других типов процессоров. CUDA может использоваться двумя различными способами: (1) через среду выполнения API, которая предоставляет набор подпрограмм и расширений Clike и (2) через API-интерфейс драйвера, который обеспечивает более низкий уровень контроля над оборудованием, но требует большего количества кода и программирования. Оба интерфейса OpenCL и CUDA вызывают фрагмент кода, который запускается на GPU ядрах. Существуют различия в том, что именно каждый язык принимает в качестве легального ядра, в нашем случае необходимо изменить исходные коды ядра, как объясняется в разделе «Приложение».

Настройка GPU для выполнения запуска на ядре существенно отличается между CUDA и OpenCL. Их API для создания контекста и копирования данных различны, и для сопоставления ядра в элементах обработки графического процессора используются различные соглашения. Эти различия могут повлиять на время, необходимое для кодирования и отладки приложения GPU, но здесь мы в основном фокусируемся на различиях производительности во время выполнения. OpenCL имеет переносимый язык для программирования графического процессора, способный ориентироваться на очень разнородные устройства параллельной обработки. В отличие от ядра CUDA, ядро OpenCL может быть скомпилировано во время выполнения, что добавит время работы для OpenCL. С одной стороны этот компилятор «точно в срок» может позволить компилятору генерировать код, который лучше использует целевой графический процессор. С другой стороны CUDA разрабатывается той же компанией, которая разрабатывает аппаратные средства, на которых она выполняется, поэтому можно ожидать, что она будет лучше соответствовать вычислительным характеристикам GPU, предлагая больше доступа к функциям и лучшей производительности. Учитывая эти факторы, интересно сравнить производительность OpenCL с показателями CUDA в реальном приложении. В этой работе мы используем вычислительно-интенсивное научное приложение для обеспечения сопоставления производительности CUDA и OpenCL на графическом процессоре NVIDIA. Чтобы лучше понять влияние производительности на использование каждого из этих программных интерфейсов, мы измеряем время передачи данных на GPU, время выполнения ядра и время работы приложения в конечном итоге. Поскольку в нашем случае ядра OpenCL и CUDA очень похожи, а остальная часть приложения идентична, любая разница в производительности может быть связана с эффективностью соответствующей структуры программирования. Было сделано не так много формальной работы по систематическому сравнению CUDA и OpenCL. Исключением является [3], где CUDA и OpenCL имеют аналогичную производительность. Базовый набор, содержащий программы CUDA и OpenCL, объясняется в [4]. Исследование производительности для графических процессоров ATI, сравнивающее производительность OpenCL с вычислительной системой Stream ATI [5], выходит за рамки этой работы. Оставшаяся часть теста организована следующим образом. В разделе «Приложение» представлено тестовое приложение, а также ядра OpenCL и CUDA. Раздел «Тестирование производительности» объясняет проведенное нами исследование и анализирует результаты.

Приложение

Приложение Adiabatic QUantum Algorthms (AQUA), использованное в этой работе, является методом симуляции Монте-Карло [6] квантовой системы спина, написанной на C++. Мы аппроксимируем конфигурацию квантового спина классической Изинг-спин системы [7]. Классическое приближение состоит из ферромагнетически связанных копий квантовой системы. Каждая копия соединяется ровно с двумя другими копиями, образуя кольцо копий. Этот процесс приближения называется разложением Сузуки-Троттера [8]. В данной работе моделируются квантовые системы размером от 8 кубитов (квантовые биты) до 128 кубитов, а количество слоев, используемых для аппроксимации квантовой системы, составляет 128 для задач всех размеров. Количество переменных в этой многоуровневой системе – это количество кубитов в каждом слое, умноженное на количество слоев. Во время развертки Монте-Карло каждая переменная в слое вероятностно перевернута, поэтому каждая развертка требует изучения всех переменных и обновления тех, которые перевернуты. В работе моделируется каждая слоистая система в разных точках во время адиабатической квантовой эволюции [9]. В каждой точке моделируется полная многоуровневая система, поэтому общее число переменных, обрабатываемых приложением, это количество переменных в каждой многоуровневой системе, умноженное на количество точек, используемых для моделирования адиабатической эволюции.

Сопоставление структур данных с потоками графического процессора и центрального процессора в AQUA подробно описано в [10], где объясняется реализация алгоритма CUDA. В этой работе мы оптимизировали шаблоны доступа к памяти ядра. Затем мы портировали ядро CUDA на OpenCL. Другой связанный код, например, для обнаружения и настройки графического процессора или для копирования данных на GPU и из него, необходимо переписать для OpenCL. Каждый многопроцессор в графическом процессоре назначается для развертки многоуровневой системы. Для 8-битной системы, например, должны быть охвачены 27 слоистых систем, потому что у нас есть 27 точек моделирования. Таким образом, у нас есть 27 рабочих групп (на языке OpenCL) или блоков потоков (на языке CUDA). Ядро генератора случайных чисел Mersenne-Twister [11], называемое вычислительным ядром, как описано в [10], требовало аналогичных изменений для компиляции и запуска в OpenCL. Мы также попытались

портировать код на графические процессоры ATI. В результате было задействовано гораздо больше изменений в ядре в основном из-за отсутствия объявлений глобальных переменных в OpenCL ATI.

С инструментами разработки OpenCL от ATI нельзя выделять память статически. Память должна быть выделена до вызова ядра, и для доступа к ней должен использоваться указатель. На рисунке 1 показан код для инициализации структур данных Mersenne-Twister для инструментов OpenCL NVIDIA.

```
__global unsigned int mt[MAX_RAND_CHAINS][NN][MAX_RAND_THREADS];
__global int mti[MAX_RAND_CHAINS][MAX_RAND_THREADS];

__kernel void ocl_init_rand(int seed) {

  mt[chain][0][thread]= seed + chain * MAX_RAND_THREADS * NN + thread;

  for (mti[chain][thread]=1; mti[chain][thread]<NN; mti[chain][thread]++) {
    mt[chain][mti[chain][thread]][thread] =
      (1812433253UL * (mt[chain][mti[chain][thread]-1][thread] ^
      (mt[chain][mti[chain][thread]-1][thread] >> 30)) + mti[chain][thread]);
  }
}
```

Рис. 1 – Код инициализации Mersenne-Twister для компилятора OpenCL от NVIDIA

На рисунке 2 показан код рисунка 1, измененный для приема указателей на динамически распределенную память. Передача массивов, как в mt[][NN][MAX_AND_THREADS], работает под инструментами NVIDIA, но не под инструментами ATI. В результате для сопоставления одномерных распределенных массивов с трехмерными массивами, необходимыми для метода Мерсенн-Твистера, оказались необходимы операции расчета индекса из рисунка 2.

```
__kernel void ocl_init_rand(int seed, __global unsigned int *mt, __global int *mti) {
  int chain = get_global_id(0); int thread = get_global_id(1);
  int base = chain * MAX_RAND_THREADS * NN + thread;

  mt[base] = seed + base;

  int index = chain * MAX_RAND_THREADS + thread;
  for (mti[index]=1; mti[index]<NN; mti[index]++) {
    int index2 = base + mti[index] * MAX_RAND_THREADS;
    int index3 = base + (mti[index] - 1) * MAX_RAND_THREADS;
    mt[index2] = (1812433253UL * (mt[index3] ^ (mt[index3] >> 30)) + mti[index]);
  }
}
```

Рис. 2 – Код инициализации Mersenne-Twister для компиляторов ATI и NVIDIA OpenCL

Следует обратить внимание, что достигнута совместимость на уровне исходного кода между ATI и NVIDIA. Результирующие исполняемые файлы не были совместимы с оборудованием другого поставщика, поэтому в настоящее время невозможно достичь совместимости во время выполнения. Чтобы уменьшить влияние шаблонов кодирования на тесты производительности, в остальной части исследования мы используем очень похожие ядра CUDA и OpenCL, скомпилированные с инструментами разработки NVIDIA, как показано на рисунке 1. Ядра содержат сочетание целочисленных, плавающих и логических операций, действующих на разные структуры данных. Эта сложность отличает их от некоторых других приложений GPU, где ядро используется для более простых операций, таких как добавление или умножение матричных элементов.

Тестирование производительности

Тестируются CUDA и OpenCL версии нашего приложения на NVIDIA GeForce GTX-260. Оба инструментария CUDA и OpenCL были в версии 2.3. В [10-13] уделено внимание сохранению оперативности работы компьютера и уменьшению нагрузки на GPU, чтобы убедиться, что компьютер остается работоспособным во время работы приложения. Для экспериментов в этой работе основной задачей является максимальная производительность, поэтому сокращено выполнение кода ЦП, а также части копии данных до минимума и увеличена нагрузка на GPU до максимума. В результате пользовательский интерфейс компьютера был очень «вялым» во время этих тестов. Никакое взаимодействие с компьютером не было предпринято во время фактических сборов данных, чтобы убедиться, что вычислительная мощность графического процессора осталась посвященной приложению AQUA. Приложение выполняет следующие шаги во время его запуска:

1) Установка графического процессора (включает обнаружение GPU, компиляцию ядра для OpenCL и т. д.)
2) Чтение ввода
3) Копирование данных на GPU
4) Запуск ядра на GPU
5) Копирование данных обратно на хост
6) Обработка возвращенных данных с помощью CPU и вывод результатов

В таблице 1 сообщается общее время, необходимое для копирования данных на GPU и из него, запуска ядра (сумма времени, необходимого для выполнения шагов 3, 4 и 5) в качестве времени операций GPU. Оба ядра выполнили 20 000 разверток переменных в каждой многоуровневой системе. В таблице 1 показано время, необходимое для запуска всего приложения от начала до конца, соответствующее времени, затрачиваемому на этапы с 1 по 6. Мы решили каждую проблему 10 раз с CUDA и OpenCL, чтобы получить повторяющееся среднее время.

Таблица 1 – Время работы приложения в секундах

Кубиты	Время операций GPU			
	CUDA		OpenCL	
	avg (сек)	stdev (сек)	avg (сек)	stdev (сек)
8	1.97	0.030	2.24	0.006
16	3.87	0.006	4.75	0.012
32	7.71	0.007	9.05	0.012
48	13.75	0.015	19.89	0.010
72	26.04	0.034	42.32	0.085
96	61.32	0.065	72.29	0.062
128	101.07	0.523	113.95	0.758

Чтобы лучше понять эффективность CUDA и OpenCL при передаче данных и операциях с ядром, таблица 2 разбивает время операций GPU на время работы ядра (шаг 4) и время передачи данных на графическое устройство и с него (шаги 3 и 5).

В таблице 3 показан объем данных, передаваемых между GPU и хостом. Тот же объем данных копируется с хоста на GPU (шаг 3) и с GPU обратно на хост (этап 5), поэтому каждый из этапов 3 и 5 передает половину суммы, указанной в таблице 3.

Таблица 2 – Время выполнения ядра и время передачи данных GPU в секундах

Кубиты	Время операций GPU			
	CUDA		OpenCL	
	avg (сек)	stdev (сек)	avg (сек)	stdev (сек)
8	1.96	0.027	2.23	0.004
16	3.85	0.006	4.73	0.013
32	7.65	0.007	9.01	0.012
48	13.68	0.015	19.80	0.007
72	25.94	0.036	42.17	0.085
96	61.10	0.065	71.99	0.055
128	100.76	0.527	113.54	0.761

Таблица 3 – Количество данных, передаваемых между GPU и хостом в КБ

Кубиты	Переданные данные
8	649.05
16	1,633.32
32	3,553.44
48	8,210.22
72	15,338.77
96	33,124.49
128	49,541.04

Чтобы сравнить время передачи данных CUDA и OpenCL, на рисунке 3 показано время передачи OpenCL, деленное на время передачи для CUDA. Как видно, затраты на перенос данных OpenCL существенно не изменяются для разных размеров задач.

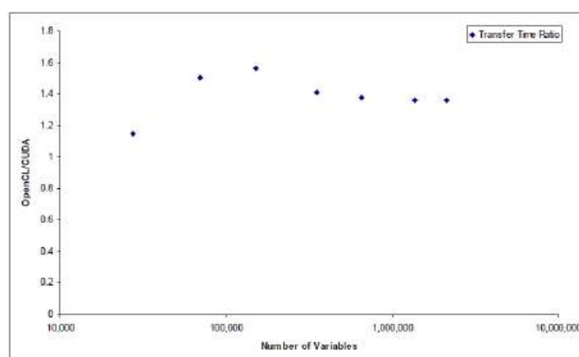

Рис. 3 – Коэффициент времени передачи данных OpenCL/CUDA

На рисунке 4 показано количество переменных, обработанных в секунду двумя ядрами в зависимости от количества переменных в задаче (обработанные переменные/время выполнения ядра). Видно, что для каждого размера задачи версия приложения CUDA обрабатывает больше переменных за секунды, чем версия OpenCL.

На рисунке 5 показана относительная разница во времени (Время OpenCL - Время CUDA) / (время CUDA) для разных размеров задач. Отображаются данные, полученные как из времени выполнения ядра, так и из конца в конец.

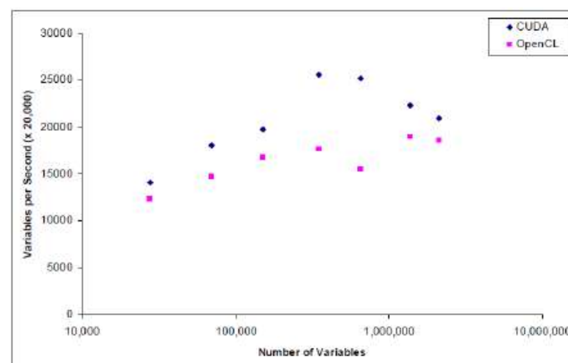

Рис. 4 – Скорость обработки для разных размеров проблем

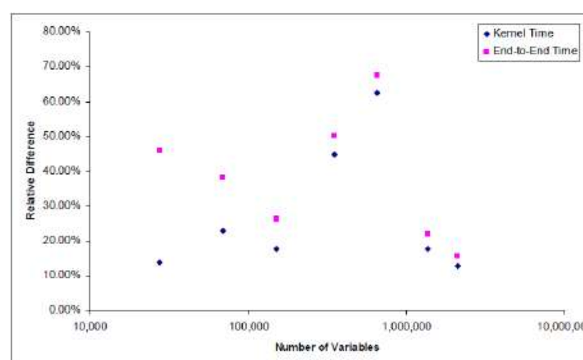

Рис. 5 – Относительная разница во времени выполнения между CUDA и OpenCL

Изменение производительности для разных размеров задач связано с различиями в размерах структуры данных и их размещением в памяти GPU. Производительность графического процессора сильно зависит от этих задач. Однако эти эффекты характерны для используемого алгоритма, поэтому здесь мы сосредоточимся на разнице в производительности между CUDA и OpenCL. Для всех размеров задач показывается значительная разница в пользу CUDA. Производительность ядра OpenCL составляет примерно от 13% до 63% медленнее, а время от конца до конца – примерно от 16% до 67% медленнее. Как и ожидалось, время работы ядра и конечного времени приближается друг к другу по стоимости с большими размерами задач, так как вклад времени ядра в общее время работы увеличивается.

Литература

1. Kirk, D. and Hwu, W., *Programming Massively Parallel Processors: A Hands-on Approach,* Morgan Kaufmann Publishers, 2010.
2. Tsuchiyama, R., Nakamura, T., Iizuka, T., and Asahara, A., *The OpenCL Programming Book,* Fixstars Corporation, 2010.
3. Khanna G., and McKennon, J., *Numerical Modeling of Gravitational Wave Sources Accelerated by OpenCL,* Fixstars Corporation, 2009.

4. Danalis, A., et al, The Scalable Heterogeneous Computing (SHOC) Benchmark Suite*, The Third Workshop on General-Purpose Computation on Graphics Processing Units,* 2010.
5. Miller, F.P., Vandome, A.F., McBrewster, J. (eds), *AMD FireStream: ATI Technologies, Stream processing, Nvidia Tesla, Advanced Micro Devices, GPGPU, High-performance computing, Torrenza, Radeon R520,* Shader, Alphascript Publishing, 2009.
6. Berg, B.A., *Markov Chain Monte Carlo Simulations and Their Statistical Analysis,* World Scientific Publishing, 2004.
7. Fischer, K.H. and Hertz, J.A., *Spin Glasses,* Cambridge: Cambridge University Press, 1993.
8. Das, A. and Chakrabarti, B.K., *Quantum Annealing and Related OptimizationMethods,* Springer-Verlag, 2005.
9. Metodi, T.S., Chong, F.T., *Quantum Computing for Computer Architects,* Morgan and Claypool Publishers, 2006.
10. Karimi, K., Dickson, N.G., and Hamze, F., *High-Performance Physics Simulations Using Multi Core CPUs and GPGPUs in a Volunteer Computing Context, International Journal of High-Performance Applications,* 2010.
11. Matsumoto, M. and Nishimura, T., *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator,* ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, 1998.
12. Р.Ф. Гибадуллин, А.Г. Савельев, М.Ю. Перухин, *Вестник технологического университета*, 19, 20, 110-116 (2016).
13. Р.Ф. Гибадуллин, А.Д. Леонов, М.Ю. Перухин, *Вестник технологического университета*, 20, 8, 83-86 (2017).

© **Р. Ф. Гибадуллин** – к.т.н.; доцент кафедры компьютерных систем Казанского национального исследовательского технического университета им. А.Н. Туполева – КАИ (КНИТУ-КАИ). e-mail: landwatersun@mail.ru; **Д. Д. Фирсова** – студентка Казанского национального исследовательского технического университета им. А.Н. Туполева – КАИ (КНИТУ-КАИ). E-mail: argalimov@kai.ru; **Н. В. Кормильцев** – специалист кафедры систем информационной безопасности Казанского национального исследовательского технического университета им. А.Н. Туполева – КАИ (КНИТУ-КАИ), e-mail: kormiltsevnv@gmail.com; **А. Д. Уваров –** специалист кафедры систем информационной безопасности Казанского национального исследовательского технического университета им. А.Н. Туполева – КАИ (КНИТУ-КАИ), e-mail: obg-96@mail.ru; **М. Ю. Перухин** – к.т.н.; доцент кафедры автоматизированных систем сбора и обработки информации Казанского национального исследовательского технологического университета (КНИТУ); к.т.н., доц. каф. «Материаловедение и технологии материалов» института электроэнергетики и электроники КГЭУ. e-mail: perukhin@inbox.ru

© **Dr. (PhD) Gibadullin Ruslan –** associate professor of computer system departmen of Kazan National Research Technical University named after A.N. Tupolev – KAI (KNRTU-KAI). e-mail: landwatersun@mail.ru; **Firsova Daria –** student of Kazan National Research Technical University named after A.N. Tupolev – KAI (KNRTU-KAI), e-mail: argalimov@kai.ru; **Kormiltsev Nikita –** specialist of information security systems department of Kazan National Research Technical University named after A.N. Tupolev – KAI (KNRTU-KAI). e-mail: kormiltsevnv@gmail.com; **Uvarov Alexander –** specialist of information security systems department of Kazan National Research Technical University named after A.N. Tupolev – KAI (KNRTU-KAI), e-mail: obg-96@mail.ru; **Dr. (PhD) Perukhin Marat** – associate professor of automated systems for the collection and processing of information department of Kazan National Research Technological University; candidate of technical sciences, associate professor of the chair «Material science and materials technology» of the Institute of electricity and electronics, Kazan state power engineering university, e-mail: perukhin@inbox.ru.