Математические структуры и моделирование
2012, вып. 26, с. 73–88

УДК 004.428:519.684

ОБЗОР ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ ПЛАТФОРМ ДЛЯ ГРАФИЧЕСКИХ УСКОРИТЕЛЕЙ

А.В. Багма

В статье рассматриваются основные платформы для высокопроизводительных вычислений на графических ускорителях, их основные особенности и характеристики.

Введение

Графические ускорители становятся одной из главных движущих сил в высопроизводительных компьютерных вычислениях. Они используются как в национальных, так и в коммерческих установках суперкомпьютеров. Благодаря им появляется такое понятие как персональный суперкомпьютер, что меняет смысл самого этого понятия. Появляются новые системы, языки и средства разработки, позволяющие использовать графические ускорители для построения систем различной сложности.

На текущий момент существует сравнительно большое количество библиотек, позволяющих производить вычисления на графических ускорителях. Среди наиболее известных и часто используемых можно назвать CUDA от nVidia, OpenCL, разрабатываемый Khronos Group, Stream как собственная разработка AMD, а также наборы библиотек, позволяющие программировать видеокарты с использованием языка Fortran.

CUDA — параллельная вычислительная архитектура и программная модель, разрабатываемая компанией nVidia. Архитектура CUDA включает в себя сборочный язык (assembly language) PTX и технологию сборки, на которой основывается множество параллельных языков и API интерфейсы, встроенные в графические ускорители nVidia, включая C (C++) для CUDA, OpenCL, Fortran и DirectX Compute shaders.

C для CUDA использует стандартный C с расширениями и использует возможности техники, которые недоступны в традиционном OpenCL или Direct3D (за исключением вычислительных возможностей шейдеров). Наиболее важной из этих возможностей является разделяемая память, которая позволяет значительно увеличить производительность приложений с ограниченной пропускной

Омский государственный университет им. Ф.М. Достоевского
E-mail: abagma@live.com

способностью и плавающих вычислений с двойной точностью. Также важной особенностью является возможность использования произвольной модели загрузки и хранения информации, что позволяет осуществить реализацию алгоритмов, невозможную, либо весьма затруднённую ранее.

Stream — технология от ATI. Это набор аппаратных и программных средств и технологий, который позволяет графическим процессорам AMD работать совместно с центральным процессором для ускорения множества приложений, не ограничивающихся графикой. Это позволяет создавать реализации тех или иных задач, работающие со значительно большей скоростью.

OpenCL — первый открытый свободный стандарт для параллельного программирования разнородных систем. OpenCL предоставляет универсальную программную среду для разработчиков программного обеспечения для создания эффективного и переносимого кода для высокопроизводительных серверов, настольных систем и портативных устройств. Стандарт позволяет использовать разнородную смесь многоядерных центральных процессоров, графических ускорителей, Cell архитектуры и других параллельных процессоров вроде DSP.

1. CUDA

Платформа CUDA используется практически повсеместно и на данный момент фактически является стандартом. Особенностью технологии является её доступность для чрезвычайно широкого круга платформ. Использовать её возможно практически везде, начиная с персонального компьютера и заканчивая суперкомпьютерными системами. Также с использованием технологии связано возникновение нового класса гибридных CPU-GPU серверов. В качестве примера можно привести Super Micro 14 GPU Server и Ball Bullx Blade Enclosure.

Одной из причин широкого распространения данной платформы является чрезвычайно высокое отношение производительности к стоимости. Графический процессор nVidia Tesla C2050 имеет 448 вычислительных элементов, достигает пиковой производительности в 515 GFLOP/сек. для вычислений с плавающей точкой двойной точности и до 1030 GFLOP/сек. для вычислений одинарной точности [1].

Для технологии характерно разнородное программирование — использование различных процессоров для различных задач. Одной из причин подобного разделения является различие в функциональных возможностях работающих в системе процессоров. Процессоры, используемые в графических ускорителях, специализированы и слабо подходят для обеспечения внешнего взаимодействия системы. Основной отличительной особенностью технологии CUDA является её неразрывная связь с продукцией nVidia. Технология поддерживает открытый стандарт OpenCL, но использует более тесную связь с GPU от nVidia и учитывает специфические особенности аппаратного обеспечения данного производителя. Это позволяет несколько расширить и улучшить работу на соответствующих графических ускорителях.

Технология как таковая влючает в себя набор библиотек и средств разработки, адаптированных для использования с различными языками программи-

рования и платформами. В настоящий момент система содержит следующие библиотеки: CUDA C — базовая платформа для разработки приложений на C/C++, включающая драйвер с поддержкой стандарта OpenCL. На основе этой библиотеки происходит реализация вычислительных возможностей CUDA на других языках.

Технология Direct Compute предоставляет пользовательский интерфейс вычислений на графических ускорителях. Данная технология разрабатывается Microsoft и поддерживает все CUDA совместимые архитектуры графических ускорителей, начиная с G80 (DX10 и DX11).

Среди средств разработки на фортране следует отметить следующие библиотеки и системы разработки: CUDA Fortran, PCI Accelerator, NOAA Fortran buildings (ftoc), Flagon. Также существуют решения для других платформ: Python, Java, .Net — это pyCUDA, JaCUDA, CUDA.Net и BSGP. Кроме того существует огромное количество портов на сторонние платформы. Cublas — CUDA Accelerate Basic Linear Algebra Subprogramms — библиотека подпрограмм для матриц и векторов в пространстве памяти графического ускорителя, заполнения объектов данными, вызов последовательностей функций и получение данных из GPU. Среди возможностей библиотеки можно выделить вычисления как с одинарной, так и с двойной точностью, комплексные вычисления с одинарной точностью.

CUFFT (CUDA FFT) — библиотека, предназначенная для параллельных вычислений FFT (Fast Fourier Transform) на графических устройствах от nVidia. Использует «плановую» (plans like) FFTW. План содержит информацию об оптимальной конфигурации для переданной трансформации. План может быть постоянным для предотвращения повторных вычислений, что может оказаться весьма полезным, особенно учитывая, что различные типы преобразований могу потребовать различных конфигураций потоков, блоков и ядер.

Библиотекой поддерживаются 1-о, 2-х и 3-х мерные трансформации комплексных и действительных данных. Возможности пакетной обработки позволяют выполнять множество одномерных трансформаций параллельно. Допустимый размер одномерной трансформации можеть достигать 8 миллионов элементов. 2-х и 3-х мерные трансформации допускают размеры от 2-х до 16384 элементов.

MAGMA (Matrix Algebra on GPU and Multicore Architectures) — данная библиотека основана на библиотеке LAPACK (Linear Algrbra PACKage), расширенной для использования в разнородных системах, функционально идентична базовой по функционированию интерфейса и хранению данных. Задумывалась как простой в использовании порт библиотеки LAPACK с учетом преимуществ GPU имногоядерных архитектур.

PyCuda — открытый проект, реализующий все возможности CUDA с использованием Python. Имеется управление памятью и распределение ресурсов. Также имеется возможность интеграции с NUMPY, набором библиотек для научных вычислений с использованием Python. В качестве отличительной особенности можно указать то, что CUDA программы представляются в виде строковых переменных языка Python, что позволяет использовать некоторые

дополнительные возможности: метаморфизм — изменение исходного кода на лету, полную предобработку кода.

Trust – Deep Dive — стандартная библиотека шаблонов для CUDA, разработанная в nVidia Research. Является проектом с открытым исходным кодом. Данная библиотека использует API CUDA. Для повышения эффективности работы широко используются шаблоны C++. Данная библиотека имитирует стандартную библиотеку шаблонов (STL) и предлагает ряд дополнительных шаблонных алгоритмов и контейнеров.

Библиотекой поддерживается шаблонное метапрограммирование: поддержка встроенных, либо заданных пользователем типов, изменение алгоритмов в соответствии с оперируемым типом данных, переопределение операций. Имеются специализированные и оптимизированные алгоритмы поиска, сортировки, генерации псевдослучайных чисел и др. Данная библиотека по сути выступает в качестве прослойки между самим хостом и устройством GPU.

1.1. Архитектура CUDA

Отличительной особенностью модели параллеллизма, используемой в CUDA, является использование сотен ядер и тысяч параллельных потоков и совместное использование cpu и gpu, не смотря на то, что они являются различными устройствами с раздельной памятью. В действительности происходит разделение на хост(cpu) и устройство(gpu). Хост контролирует процесс выполнения в целом, а устройство выполняет роль высокопроизводительного сопроцессора. При этом функции работы с ядром CUDA вызываются cpu, а фактически выполняются на графическом ускорителе.

Принцип работы самого графического ускорителя возможно описать следующим образом: каждое устройство gpu имеет множество вычислительных ядер, каждое из которых выполняет свой собственный блок потоков. Программа, использующая CUDA, автоматически масштабируется для использования доступной конфигурации ядер с максимальной эффективностью, т. е. CUDA обладает расширяемой моделью для программирования параллельных задач.

Параллельная модель программирования CUDA использует 3-и базовые абстракции:

- иерархия групп потоков,
- разделяемая память,
- барьерная синхронизация.

Эти абстракции обеспечивают параллелизм данных и потоков и позволяют разбивать задачи на отдельные части, которые могут быть выполнены независимыми друг от друга блоками потоков.

Каждая выделенная подпроблема решается совместно и параллельно отдельными потоками в соответствующем блоке. Подобная реализация, с одной стороны, сохраняет выразительность конечного языка разработки, позволяя потокам совместно решать некоторую подзадачу, а с другой стороны, автоматически

увеличивает расширяемость программной платформы, т. к. каждый блок может быть размещен на любых доступных процессорных ядрах, в любой последовательности, как последовательно, так и параллельно, единственное, что нужно знать системе при этом, — это необходимое количество ядер.

Основным языком разработки для платформы CUDA является CUDA C — расширение языка C. Это расширение позволяет разработчику определять специальные функции, называемые ядрами, которые будут вызваны требуемое количество раз, выполняясь параллельно в некотором количестве потоков. Каждый поток, выполняемый ядром, получает уникальный идентификатор, доступный через встроенный объект threadIdx — трёхмерный вектор. Соответственно, потоки могут идентифицироваться им как одно, двух или трёх мерными по формируемым ими пространствам, что позволяет использовать их естественным образом формируя одно, двух либо трёхмерные блоки для соответствующей структуры вычислений (векторы, матрицы и др.). Существует ограничение, налагаемое на количество потоков в одном блоке, т. к. все потоки одного блока выполняются на одном процессорном ядре и должны разделять ограниченные ресурсы ядра. Для существующих на данный момент GPU оно равно 1024.

Так как программное ядро может быть запущенно одновременно в виде нескольких эквивалентных блоков, то общее число потоков определяется как количество блоков, умноженное на количество потоков в блоке. Потоки в блоках организуются в одно или двухмерные системы (сети). Количество блоков в сети определяется размерами данных, которые необходимо обработать, либо количеством процессоров в системе. При запуске ядра задаётся количество блоков в сетке и количество потоков в блоке.

Каждый блок в сетке обладает, в зависимости от конфигурации, одно либо двухмерным идентификатором blockIdx, размерность определяется встроенной переменной blockDim.

Блоки должны иметь возможность использоваться независимо друг от друга в любом порядке как последовательно, так и параллельно, что позволяет создавать программы, автоматически расширяющиеся с увеличением количества ядер в системе. Потоки внутри блока могут взаимодействовать друг с другом через разделяемую память и синхронизировать своё выполнение для координации доступа к памяти.

Потоки могут получать доступ к данным из нескольких пространств памяти. Каждый поток имеет доступ к разделяемой памяти блока, доступной всем потокам, находящимся в одном блоке. Также есть два дополнительных пространства памяти, доступных всем потокам: пространство констант и текстурная память. Эти пространства оптимизированы для использования в различных задачах и имеют различные цели. Текстурная память имеет отличающуюся систему адресации, а также невозможность размещения данных некоторых типов. Глобальная память является общей для всего приложения.

Написание программ для CUDA подразумевает использование одного из двух интерфейсов: CUDA C или CUDA driver API. CUDA C является моделью программирования, использующей расширение языка C. Исходный код, написанный на CUDA C, должен быть скомпилирован при помощи специали-

зированного компилятора (nvcc). Введённые расширения позволяют определять вычислительное ядро как функцию C. При этом задается требуемое количество измерений, блоков и сеток.

CUDA API — это низкоуровневый API на языке C, предоставляющий функции для загрузки ядер в воде готовых сборок и ассемблерного кода для их параллельного исполнения.

CUDA C предоставляет также API времени исполнения. Оба вида API (runtime и driver) предоставляют возможности для управления памятью, предачи данных между памятью хоста и устройства, функции управления системой с несколькими устройствами. Использование runtime API позволяет вести разработку значительно быстрее, в то время как использование driver.api позволяет получить больший уровень контроля, хотя и требует написания большего количества кода и усложняет отладку.

Ядро может быть написано согласно набору инструкций PTX. Результатом компиляции может быть как ассемблерный код (PTX код), так и бинарный объект. При чем для хоста и устройства код генерируется отдельно, но в дальнейшем код устройства может быть привязан к коду хоста в виде глобального массива данных либо запущен самостоятельно на устройстве.

Компилятор обладает функциональностью для обеспечения соответствия итогового кода версии устройства и PTX инструкций, а также задавать минимально допустимые версии устройств и наборов инструкций. Среда выполнения CUDA инициализируется при первом обращении к функциям CUDA. Как только среда инициализировалась в потоке хоста, любые ресурсы, размещённые посредством среды, становятся доступны только в её контексте, при этом поток хоста имеет возможность вызывать только функции среды для оперирования размещёнными ресурсами. Происходит это из-за того, что контекст CUDA, создаваемый при инициализации, доступен только в конкретном потоке хоста и ни в каком другом. В общем случае в системе с несколькими устройствами ядра запускаются на устройстве с индексом 0.

Как было ранее указано, программная модель CUDA представляет собою композицию хоста и устройства, каждый из которых имеет собственную память, при чем вычислительные ядра могут оперировать только с памятью устройства. Память устройства может быть представлена как в виде линейной памяти, так и в виде массива CUDA (данное размещение изначально предназначено и оптимизировано для текстурной выборки).

Линейная память представляет собою 32-х битное адресное пространство для устройств первого поколения (1.x) и 40-а битное для устройств второго поколения (2.x). Также нужно отметить, что данные можно размещать в виде отдельных сущностей, которые могут ссылаться друг на друга через указатели. Для работы с 2-х и 3-х мерными массивами возможно использовать специально оптимизированные функции.

Несколько потоков хоста могут выполнять код на одном и томже устройстве, но предполагается, что один поток хоста может исполняться только на одном устройстве в каждый момент времени.

CUDA поддерживает набор средств для работы с текстурами, используемы-

ми GPU в графических операциях. Чтение данных из текстурной памяти может дать некоторый прирост производительности. Ссылки на текстуры представляют собою ссылки на некоторую область памяти. На одну и туже текстуру может быть назначено несколько ссылок. Элементы текстуры называются текселеми. Текстура может быть представлена любой областью линейной памяти, либо массивом CUDA. Массив CUDA — это размещение памяти оптимизированное для использования текстур, может быть одно, 2-х и 3-х мерным и состоять из 1-го, 2-х, 4-х компонентов на каждый элемент. Каждый компонент может быть знаковым либо беззнаковым 8-и, 16-и или 32-х битным целым, либо 16-и или 32-х битным числом с плавающей точкой единичной точности.

Массивы CUDA читаемы только ядрами посредством выбора текстуры и могут быть только привязаны к текстурам с соответствующей структурой компонентов.

Имеется возможность асинхронно производить следующие операции:

- запуск ядер,
- копирование памяти между устройством и хостом,
- выполнять специализированные функции (помечены префиксом Asynk),
- вызовы функций установки памяти.

Контекст CUDA во многом аналогичен процессу CPU. Все ресурсы и действия обрабатываются в рамках driver API и инкапсулируются внутри контекста CUDA. Система автоматически очищает все занимаемые ресурсы при уничтожении контекста. Каждый контекст обладает собственным 32-х битным адресным пространством, соответственно, данные из разных контекстов ссылаются на различные области памяти. Поток хоста может взаимодействовать только с одним контекстом одновременно, как только контекст создан, он становится текущим для потока.

Вычисления могут проводиться в соответствии с несколькими моделями:

- Default — устройство может быть использовано множеством потоков хоста,
- Exclusive — устройство может использоваться только одним потоком хоста,
- Prohibited — потоки хоста не могут использовать устройство.

Архитектура CUDA построена вокруг расширяемого массива многопоточных Streaming мультипроцессоров. Когда CUDA программа на центральном процессоре хоста запускает на выполнение набор ядер, блоки её составляющие передаются на выполнение доступному количеству процессоров. Все потоки в одном блоке выполняются параллельно на одном процессоре, также несколько блоков могут быть запущены на одном и том же мультипроцессоре. Как только один блок завершает своё выполнение, запускается другой.

Мультипроцессоры созданы таким образом, чтобы выполнять сотни потоков одновременно. Управление этими потоками и их выполнение происходит в соответствии с принципом вычислений SIMT (Single Instruction Multiple Thread). Для максимизации загрузки реализуется параллелизм уровня потоков и параллелизм инструкций в рамках одного потока. Мультипроцессор создаёт и управляет выполнением группами потоков по 32 параллельных потока в каждой. Такие группы называются «основами» (wraps). Отдельные потоки внутри основы стартуют с одного и того же адреса инструкции, но имеют собственные счётчики адреса инструкции, собственные состояния регистров и имеют возможность ветвиться и выполняться независимо друг от друга.

Когда мультипроцессору передают один либо более блоков потоков на выполнение, он разделяется на основы, которые распределяются на выполнение планировщиком основ. Блоки разделяются всегда одним и темже способом. В каждой основе потоки расположены последовательно, с последовательно увеличивающимся идентификатором, при чем первый поток имеет идентификатор равный нулю. Основа выполняет только одну инструкцию одновременно, соответственно, максимальная производительность достигается, когда все 32-а потока в основы совместно перемещаются по инструкциям на выполнение. Если потоки в основе расходятся на зависящие от данных ветви, основа последовательно выполняет каждую ветвь, при этом потоки, не входящие в данную ветвь, находятся в неактивном состоянии. Когда все ветви завершены, потоки объединяются в единый путь выполнения. Разделение на ветви происходит только внутри основы, различные основы выполняются независимо, не смотря на то, выполняют они одни и те же или различные части кода.

Архитектура SIMT сходна с SIMD — векторной организацией, при которой одна инструкция контролирует несколько обрабатывающих элементов. Ключевое отличие в том, что векторная организация SIMD переносит свою структуру и на программное обеспечение, в то время как SIMT инструкции определяют выполнение и поведение ветвления отдельного потока. В противоположность SIMD векторным машинам, SIMT позволяет создавать параллельный код уровня потоков для независимых скалярных потоков также, как и параллельный по данным код для связанных потоков. Программа может даже игнорировать SIMT поведение, если этого требует правильность реализации.

Как бы то ни было значительный прирост скорости возможен только при учёте того, что код должен очень редко требовать от потоков в основе разделяться. Векторные архитектуры в тоже время требуют, чтобы программное обеспечение сливало данные в векторы и самостоятельно обрабатывало ответвления от основного ствола выполнения.

Если неатомарная инструкция, выполняемая основой, пишет в тоже место разделяемой памяти, количество записей произведённое в эту область памяти зависит от вычислительной способности устройства. То какой поток будет последним записавшим, неизвестно. Если атомарная инструкция читает, пишет в одну и ту же локацию глобальной памяти для более чем одного потока, каждое действие для данной локации будет произведено и все они будут сохранены, но порядок, в котором это произойдёт, неизвестен.

1.2. Многопоточность аппаратного обеспечения

Контекст выполнения для каждой основы обрабатывается отдельным мультипроцессором полностью аппаратно на всем протяжении времени жизни основы. Переключение с одного контекста на другой не имеет накладных расходов, т.к. планировщик основ выбирает основы, все потоки которых полностью готовы к выполнению (являются активными) и выполняет следующую инструкцию уже для них каждый из мультипроцессоров имеет набор из 32-х регистров, которые распределены между основами, и параллельный кэш данных/разделяемую память.

Число блоков и основ, которое может быть размещено и обработано вместе на мультипроцессоре для данного ядра зависит от количества регистров и разделяемой памяти, доступной мультипроцессору. Также существует максимальное количество размещаемых блоков и основ для мультипроцессора. Эти ограничения совместно с количеством регистров и разделяемой памяти, доступной мультипроцессору, — являются основой для определения вычислительной способности устройства. В случае, если недостаточно регистров или разделяемой памяти, доступной для мультипроцессора для обработки хотя бы одного блока, ядро не сможет запуститься. Общее количество основ в блоке возможно вычислить по формуле

$$W_{block} = \text{ceil}\left(\frac{T}{W_{size}}, 1\right),$$

где T — число потоков на блок, W_{size} — размер основы (равен 32). Результатом является значение, округлённое к ближайшему целому.

Общее число регистров, доступное в блоке для устройств версии 1.x

$$R_{block} = \text{ceil}(\text{ceil}(W_{block}, G_W) \times W_{size} \times R_k, G_t),$$

для устройств версии 2.x

$$R_{block} = \text{ceil}(R_k \times W_{size}, G_t) \times W_{block},$$

где G_w — гранулярность основы (равна 2), R_k — число регистров для использования ядром, G_t — гранулярность размещения потоков (для устройств версий 1.0 и 1.1 равна 256, для устройств версий 1.2, 1.3 равна 512, для устройств версии 2.0 равна 64), W_{size} — размер основы (равен 32).

Количество размещаемой памяти на блок:

$$S_{block} = \text{ceil}(R_k, G_s),$$

S_k — количество используемой разделяемой памяти в байтах, G_s — гранулярность размещения памяти (512 для моделей 1.x, 128 для моделей 2.0).

В системах с несколькими графическими ускорителями все поддерживающие технологию устройства доступны при помощи драйвера по отдельности.

2. OpenCl

Open CL (Open Computing Language) — открытый api для вычислений в гетерогенной среде. Предоставляет единообразную среду программирования для разработки переносимого кода, использующего разнородную смесь из многоядерных CPU, GPU, процессоров архитектуры Cell и других параллельных процессоров, как DSP.

Процессоры CELL — это совместная разработка компаний Sony, IBM и Toshiba, ориентированная, в основном, на мультимедийные приложения. Процессор состоит из центрального процессора общего назначения (PowerPC), называемого PPE, и восьми процессоров специального назначения (SPE), доступных исключительно для численных вычислений. Каждый SPE может выполнять векторные операции, которые может выполнять на множестве наборов данных для каждой инструкции. По причине гетерогенного принципа построения CELL процессор значительно отличается от традиционных мультипроцессоров. Результатом такого дизайна является то, что один CELL процессор с частотой 3,2 ГГц может достигнуть пиковой производительности более 200 GFLOP/сек в вычислениях с плавающей точкой одинарной точности и до 15 GFLOP/сек для двойной точности (до 100 GFLOP/сек при использовании более современного PowerXCell процессора). Но при этом разработчикам программного обеспечения для данного типа процессоров требуется вручную управлять передачей данных между SPE и PPE.

Стандарт OpenCL во многом похож на платформу CUDA от nVidia, но данный стандарт существенно отличается по кругу систем, к которым он применим. CUDA разрабатывался исключительно для использования с GPU устройствами nVidia, в то время как Open CL должен поддерживать значительно более широкий круг устройств.

Стандарт состоит из API для координации параллельных вычислений в среде разнородных процессоров и кроссплатформенного языка программирования с хорошо специфицированной вычислительной средой. Стандарт поддерживает как программную модель параллелизма данных, так и задач. В качестве базового языка программирования используется разновидность ISO C99 с расширением для поддержки параллелизма. Численные типы данных, используемые стандартом, соответствуют стандарту IEEE 754. В рамках стандарта определены конфигурационные профили для портативных и встраиваемых устройств. Также определены правила взаимодействия с OpenGL, OpenGL ES и другими графическими API.

Сама спецификация разделена на три части: базовая спецификация, которую должен поддерживать каждый OpenCL компилятор, профили для портативных и сменных устройств, позволяющие использовать подобные устройства, учитывая их технологические особенности, и набор опциональных расширений, которые планируется добавить в базовую спецификацию в следующих ревизиях стандарта. Платформа, основанная на этих разработанных спецификациях, включает в себя язык программирования, API, библиотеки и среду выполнения.

2.1. Архитектура OpenCL

Модель платформы представляет собою хост, соединённый с одним либо несколькими OpenCL устройствами. Каждое устройство разделяется на один либо более вычислительных элементов, на которых и производятся все вычисления. Приложение OpenCL выполняется на хосте соответственно родной для данного хоста модели. Приложение вызывает команды с хоста для выполнения вычислений на устройстве. Каждый вычислительный элемент выполняет один поток инструкций как SIMD или SPMD устройство(где каждый PE имеет собственный счётчик инструкций).

OpenCL разработан для поддержки устройств с различными вычислительными возможностями в рамках единой платформы. Что включает устройства, поддерживаемые различными версиями спецификаций. Основными показателями являются: версия платформы, версия устройства и версия языка OpenCL C, поддерживаемая устройством.

Выполнение любой OpenCL программы состоит из двух частей: ядер, выполняемых на одном или нескольких устройствах, и программы хоста. Программа хоста определяет контекст ядер и управляет их выполнением. Когда ядро утверждается на выполнение, определяется индексное пространство. Экземпляр ядра выполняется для каждой точки из этого пространства. Такой экземпляр ядра называется рабочим элементом и определяется своими координатами в индексном пространстве — глобальным идентификатором рабочего элемента. Каждый рабочий элемент выполняет тот же код, за исключением специфических путей выполнения и различия данных между конкретными рабочими элементами.

Индексное пространство, поддерживаемое OpenCL, называется NDRage. NDRange — это N-мерное индексное пространство с размерностью, равной одному, двум или трём. Глобальный и локальный идентификаторы рабочего элемента являются N-мерными кортежами. Рабочим группам идентификаторы присваиваются сходным с глобальными идентификаторами образом. Рабочие элементы, назначенные в рабочие группы, получают локальный идентификатор из диапазона от нуля до размера рабочей группы в данном измерении минус один. Комбинация идентификатора группы и локального идентификатора рабочего элемента однозначно определяет рабочий элемент. Каждый рабочий элемент определяем двумя путями: с помощью глобального идентификатора или с помощью комбинации идентификатора группы и локального идентификатора внутри данной рабочей группы.

На подобную программную модель возможно отобразить большое количество программных моделей. OpenCL явно поддерживает две модели: параллелизм данных и параллелизм задач.

Контекст Выполнения ядра определяется хостом, при этом контекст включает в себя такие ресурсы как:

- устройства: коллекция OpenCL устройств доступных для использования хостом,
- ядра: функции OpenCL, запускаемые на OpenCL устройствах,

- программные объекты: программные коды, реализующие ядра,
- объекты памяти: набор объектов памяти, видимых хосту и устройствам. Объекты памяти содержат данные, над которыми могут провести операции экземпляры ядер.

Контекст создаётся и управляется устройством посредством функций, предоставляемых OpenCL API. Хост создаёт структуру данных, называемую «очередью команд», для координации выполнения программных ядер на устройствах. Хост помещает команды в командную очередь, которые затем планируются для конкретного устройства в контексте.

Команды включают в себя:

- команды выполнения ядра: запуск ядра на вычислительных элементах устройства,
- команды памяти: передача данных как между хостом и объектами памяти, так и между ними самими, включает отображение объектов памяти как на адресное пространство хоста, так и из него,
- команды синхронизации: определяют порядок выполнения команд.

Очередь команд планирует выполнение команд на устройстве. Команды могут выполняться относительно друг друга в одном из двух вариантах:

- по порядку: команды запускаются в порядке своего добавления в очередь выполнения и начинают выполняться после завершения предыдущей,
- беспорядочно: команды запускаются по порядку, но не ожидают завершения предыдущей команды перед собственным запуском.

Модель выполнения OpenCL поддерживает два типа ядер.

Ядра OpenCL, написанные при помощи языка программирования OpenCL C и скомпилированные соответствующим компилятором. Каждая реализация OpenCL поддерживает такие ядра.

Ядра, написанные для конкретных платформ (native), доступные через указатель на функцию хоста. Такие ядра выполняются последовательно, как и ядра OpenCL, и разделяют с ними объекты памяти. Родными объектами могут быть ядра определённые, как функция в коде приложения или экспортированы из библиотеки. Возможность запускать такие ядра является функциональной и зависит от конкретного устройства и реализации ядра. Тем не менее в API OpenCL включены функции для проверки возможности запуска ядра на конкретном устройстве.

Рабочие элементы, выполняющие ядро, имеют доступ к четырём различным областям памяти:

- глобальная память — область памяти, доступ как на чтение, так и на запись в которую имеют все рабочие элементы всех групп, при этом обращения к этой области памяти у некоторых устройств могут кэшироваться,

Таблица 1. Размещение данных и доступ к различным видам памяти

	Глобальная	Констант	Локальная	Частная
Хост	Динамическое размещение Доступ на чтение и запись	Динамическое размещение Доступ на чтение и запись	Динамическое размещение Нет доступа	Нет возможности размещения Нет доступа
Ядро	Нет возможности размещения Доступ на чтение и запись	Статическое размещение Доступ только на чтение	Статическое размещение Доступ на чтение и запись	Статическое размещение Доступ на чтение и запись

- область памяти констант — область глобальной памяти, содержащая константы во время выполнения ядра (данные внутри этого объекта размещаются и инициализируются хостом),
- локальная память — область памяти рабочей группы, может использоваться для размещения переменных и данных, разделяемых всеми рабочими элементами группы (реализацией этой области может быть как выделенная область памяти, так и отображение на некоторую часть глобальной памяти),
- частная память — область памяти, используемая непосредственно некоторым рабочим элементом (данные, размещённые в этой области, невидимы нигде за её пределами).

В таблице 1 описаны возможности записи в различные области памяти для хоста и устройства.

Приложение, запущенное на хосте, использует API для создания объектов в глобальной памяти и управления ими. По большей части хост и память устройства независимы друг от друга. Так как устройство определяется вне OpenCL, но хосту необходимо с ним взаимодействовать, то приходится это делать одним из двух способов: явным копированием областей памяти от устройства к хосту и наоборот, и отображением участков памяти и объектов.

Для копирования данных устройством выполняется команда передачи данных между объектом памяти и памятью хоста. При чем команды передачи могут быть как блокирующими, так и неблокирующими. Блокирующая передача возвращает данные, как только ими можно будет безопасно пользоваться, неблокирующая возвращает данные непосредственно после запуска вне зависимости от безопасности их использования.

Методы отображения позволяют хосту отразить некоторую область объекта памяти в своё адресное пространство. Эти команды также могут быть блокирующими или неблокирующими. Как только некоторая область памяти была связана с хостом, он может свободно писать в неё, либо читать из неё. Хост отвязывает область памяти, как только его взаимодействие с ней завершилось. OpenCL использует слабосвязную модель памяти, т. е. не гарантируется, что

состояние объекта, видимое некоторым рабочим элементом, все время совпадает с состоянием, видимым любым другим рабочим элементом.

OpenCL поддерживает модели параллельности данных и задач, основной является модель параллельности данных. Параллелизм данных определяет вычисления как последовательность инструкций, применяемую на множество объектов в памяти. Индексное пространство, связанное с моделью выполнения OpenCL, определяет рабочие элементы и то, как данные связываются с ними. В структурированной модели параллелизма данных существует однозначное соответствие между рабочими элементами и данными, направленными на параллельную обработку. OpenCL реализует облегчённую модель параллелизма данных, при котором не требуется соответствия один к одному. Предоставляется иерархическая модель параллелизма данных. Существует две возможности определения иерархического подразделения данных. В случае явной модели программист определяет общее количество вычислительных элементов, которые должны быть запущены параллельно, а также то, как они будут распределены среди рабочих групп. В неявной модели программист определяет только общее количество рабочих элементов на запуск, а их разделение на группы управляется средой OpenCL. Параллелизм задач определяет модель, в которой любой экземпляр ядра выполняется вне зависимости от какого бы то ни было индексного пространства. Логически это соответствует выполнению ядра на вычислительном узле с рабочей группой, содержащей только один рабочий элемент. В данной модели параллелизм выражается:

- использованием векторных типов данных, реализуемых устройством,
- выполнением нескольких задач,
- выполнением родных ядер с использованием программной модели ортогональной к OpenCL.

2.2. Синхронизация в OpenCL

В OpenCL существует два домена синхронизации:

- среди рабочих элементов в общей рабочей группе,
- команды, запущенные в одном контексте выполнения.

Синхронизация в пределах одной рабочей группы реализуется при помощи барьеров рабочей группы. Все рабочие элементы рабочей группы должны выполнить барьер, прежде чем они смогут выполнять инструкции за ним. Нужно отметить, что не предусмотрено возможностей синхронизации между рабочими группами.

Синхронизационными точками между командами в очереди команд являются: барьеры командной очереди гарантируют, что все ранее выполнявшиеся команды завершены и результаты произведённых ими изменений видны последующим командам до начала их выполнения. Этот тип может быть использован

только для синхронизации между командами в последовательной очереди команд.

Ожидание события завершения выполнения команды — это событие генерируется каждой командой после того, как выполнение завершено и состояние данных обновлено. Следующая команда ожидает появления этого события, после чего запускается сама.

3. Сравнение быстродействия программ, использующих платформы CUDA и OpenCL

Ядра OpenCL в отличие от ядер CUDA могут быть скомпилированы во время выполнения программы. С одной стороны, эта компиляция «на лету» позволяет генерировать код, оптимизированный для конкретного целевого устройства, но при этом на саму компиляцию тратится дополнительное время. В тоже время CUDA разработана непосредственно производителем аппаратного обеспечения и в ней учитываются все особенности продукции, при этом архитектура программной системы тесно связана с аппаратной архитектурой. Это позволяет добиваться большей оптимизации итогового кода. Тем не менее программная архитектура систем практически идентична, и, в случае запуска систем на одном и том же целевом вычислительном устройстве, различия в производительности будут вызваны различиями в программной реализации: оптимизацией драйверов устройства, уровнем оптимизации кода компилятором.

Для проверки выбираются алгоритмы, наиболее легко ложащиеся на обе платформы. Процесс вычислений организуется таким образом, чтобы максимально загрузить видеокарту. Для этого прибегают к минимальному взаимодействию ядра с центральным процессором во время вычислений, минимизируют передачу данных и передаваемые объёмы данных.

В работе [2] для проведения проверки используется адиабатический квантовый алгоритм, написанный на C++. Оригинальное ядро было разработано для CUDA и далее модифицировалось для использования в OpenCL таким образом, чтобы минимизировать вносимые изменения.

Общие результаты выполнения приведены в таблице 2. Совокупность результатов, полученных авторами, показывает, что CUDA лучше справляется с передачей данных между хостом и графической картой, также реализация CUDA быстрее справляется со своей задачей, чем аналогичная реализация с использованием OpenCL. Различия в производительности по результатам измерений составляют примерно 10-20% (хотя отставание в производительности ядра может превышать и 60%), что позволяет авторам говорить о большей пригодности CUDA для высоконагруженных приложений, но при этом универсальность OpenCL позволяет использовать большее количество целевых устройств.

В таблице 2 stdev — среднеквадратичное отклонение результатов измерений.

В тоже время нужно отметить, что даже при некотором отставании в скорости вычислений от CUDA OpenCL выигрывает тем, что предоставляет возможности переносимости без потерь в вычислительных возможностях приложения.

Таблица 2. Сравнительные результаты работы программы

Кубиты	Время выполнения на GPU				Общее время выполнения			
	OpenCL		CUDA		OpenCL		CUDA	
	среднее	stdev	среднее	stdev	среднее	stdev	среднее	stdev
8	1,97	0,027	2,23	0,004	2,94	0,007	4,28	0,164
16	3,87	0,006	4,73	0,013	5,39	0,008	7,45	0,023
32	7,71	0,007	9,01	0,010	10,16	0,009	12,84	0,006
48	13,75	0,015	19,81	0,085	17,75	0,013	26,69	0,016
72	26,04	0,036	42,17	0,062	23,77	0,025	54,85	0,103
96	61,32	0,065	71,99	0,055	76,24	0,033	92,97	0,064
128	101,07	0,527	113,54	0,758	123,54	1,091	142,97	1,080

Сравнительные результаты вычислений при оптимизации под различные платформы приведены в работах [3, 4].

Литература

1. Tesla C2050 performance bench mark. URL: `http://www.microway.com/pdfs/TeslaC2050-Fermi-Performance.pdf` (дата обращения: 7.11.2011).
2. Karimi K., Dickson N.G., Hamze F. A Performance Comparison of CUDA and OpenCL. URL: `http://ru.arxiv.org/abs/1005.2581v3` (дата обращения: 15.09.2011).
3. Khannaa G., McKennonb J. Numerical modeling of gravitational wave sources accelerated by OpenCL. URL: `http://arxiv.org/abs/1001.3631` (дата обращения: 7.10.2011).
4. Pang B., Pen U., Perrone M. Magnetohydrodynamics on Heterogenous architectures: a perfomance comparison. URL: `http://arxiv.org/abs/1004.1680` (дата обращения: 11.09.2011).