Введение в CUDA

Автор курса: Романенко А.А.
arom@ccfit.nsu.ru

Слайд 1
Данный раздел курса не претендует на полный охват темы. Его цель познакомить слушателей с основами программирования графических процессоров с использованием технологии CUDA. Будут рассмотрены: архитектура графических процессоров и за счет чего достигается высокая производительность, API и даны рекомендации по профилированию приложений и их оптимизации.

Слайд 2
Если расставить на диаграмме вычислительные системы, то получится картина представленная на слайде (рост производительности экспоненциальный). На графике можно видеть и однопроцессорные машины, многоядерные, а также кластерные системы, которые состоят из сотен тысяч процессоров.

Слайд 3
И рост производительности вычислительных систем как одноядерных так и кластеров шел в первую очередь за счет роста тактовой частоты процессоров (сейчас, возможно временно, в этом направлении достигнут предел), усложнения архитектуры, увеличения количества ядер на одном кристалле, интеграции контроллеров памяти на кристал и пр.

Слайд 4
Если посмотреть на то, с чем приходится сталкиваться при обработке графики, как трехмерной, так и двумерной, то это вектора и матрицы максимум размерности 4 (поворот вектора и масштабирование), накладывание фильтров. В общем большое количество однотипных операций. Чтобы разгрузить центральный процессор это «простые» операции стали передавать на обработку графическим картам. Тем более, что обработанную сцену все равно требовалось отображать на экране.

Слайд 5
И довольно быстро производительность видеокарт обогнала производительность центрального процессора. Конечно это наложило свои ограничения на программу и выполняемые операции, но об этом позже.

Слайд 6
Процессор на карте — GPU. Центральный процессор по шине PCI -Express передает данные на видеокарту и программу обработки. Сам GPU не может общаться с центрпльным процессором и обращаться к памяти компьютера.

Слайд 7
Если посмотреть на кристалл центрального процессора, то довольно много места занимает кэш и блок управления (предсказанием ветвлений и пр) и мощные арифметические устройства. Разработчики GPU пошли по другому пути. Они минимизировали кэш, уменьшили логику блока управления, а сэкономленные транзисторы (площадь кристалла) пустили на арифметические устройства (также упрощенные).

Слайд 8
На уровне архитектуры GPU представляет из себя набор кластеров обработки текстур (TPC), каждый из которых состоит из двух (реже трех) потоковых мультипроцессоров, каждый из

которых состоит из 8 потоковых процессоров. Эти потоковые процессоры и производят все обработку данных. У GPU есть глобальная память, константная память. Кроме того каждый потоковый мультипроцессор имеет свой небольшой объем разделяемой памяти.

Слайд 9
Количество потоковых мультипроцессоров может сильно меняться от одного графического процессора к другому. Соответственно меняется и производительность. Так пиковая производительность Tesla C1060 (240 ядер) в пике составляет 1TFLOPS.

Слайд 10
Одноюнитовый блок в 19-ти дюймовую стойку содержит 4 графических процессора и имеет соответствующую производительность в 4 TFLOPS в пике.

Наблюдательный слушатель заметит, что у Tesla C1060 нет видеовыхода, впрочем как и не видно разъемов под монитор на C1070. Дело в том, что эти карты предназначены только для счета.

Стоит отметить, что существует 3 типа устройств с GPU. Первое — это видеокарты общего назначения и игровые. Их производство компания NVidia в общем случае не контролирует. Поэтому многие производители видеокарт повышают тактовую частоту процессоров, возможно используют менее дорогие электронные компоненты, т.е. Делают все чтобы сделать карту с одной стороны дешевле, а с другой более производительную. И если на такой карте у вас будет с ошибкой (в виде какого-либо сбоя) рассчитан пиксел, то возможно увлеченные игрой вы этого даже и не заметите. Второй тип устройств — это Tesla. Это устройства предназначенные для математических расчетов и их производство происходит под контролем компании NVIDIA. При этом время наработки на отказ исчисляется годами. Третий тип — это карты Quadro. Эти карты также предназначены для счета, но имеют видеовыход и кроме того могут иметь больший объем оперативной памяти на борту чем карты Tesla, объем у которых 1 GB.

Слайд 11
На сайте NVIDIA собрано большое количество приложений, которые были перенесены на графический процессоры. Вот лишь некоторые из них. Видно, что применение графических процессоров для некоторых задач позволяет заменить вычислительный комплекс из сотен вычислительных ядер.

Слайд 12
Архитектура CUDA представлена на слайде. Приложения используя библиотечные вызовы (или напрямую) через драйвер может общаться с GPU.

Слайд 13
GPU — процессор с SIMD архитектурой. Таким образом единицей исполнения потока команд является поток. Все потоки исполняют одни и теже команды, но с разными данными. Потоки объединяются в блоки (конфигурация потокового блока задается программистом). Все потоковые блоки объединяются в сеть.

Слайд 14
Кроме того, что программа должна ложиться на SIMD архитектуру, область данных должна разбиваться на подобласти, которые можно обрабатывать независимо. Это ограничение связано с тем, что потоковый блок обрабатывается исключительно одним потоковым мультипроцессором и нет возможности синхронизовать выполнение потоков в разных потоковых мультипроцессорах. Внутри одного потокового блока синхронизация потоков

возможна. Каждый поток имеет доступ к глобальной и константной памяти, а также разделяемой памяти своего потокового мультипроцессора.

Слайд 15
Ядро запускается в виде сети потоковых блоков, которая может быть как двумерная так и одномерная. Потоки в потоковом блоке могут также укладываться в одно-\дву-\трехмерную решетку. Это сделано с тем, чтобы разработчикам проще было отображать задачу на физическую модель. А вообще это только способ именования потоковых блоков и потоков в них.

Слайд 16
Зная координаты потокового блока и потока внутри потокового блока всегда можно вычислить координаты потока в пространстве.

Слайд 17
После запуска ядра все потоковые блоки выстраиваются в очередь и распределяются по потоковым мультипроцессорам. При этом последовательность выполнения потоковых блоков не определена. Один потоковый блок исполняется строго на одном потоковом мультипроцессоре. В тоже время на одном мультипроцессоре, если позволяют ресурсы, может выполняться несколько потоковых блоков. Эти ресурсы — разделяемая память и количество регистров.

Слайд 18
После запуска потокового блока все потоки разбиваются на варпы и это распределение всегда одинаковое. Планировщик передает управление от одного варпа к другому.

Слайд 19
В набор разработчика CUDA входят драйвер, toolkit и SDK. Указанное на слайде расположение последних двух может отличаться.

Слайд 20
Сборка программ производится специальным компилятором. Этот компилятор поддерживает как Си, так и Си++. В принципе саму программу вы можете собирать любым понравившимся вам компилятором, будь то gcc илл Intel Compiler, но ядро (код для GPU) вам придется собирать именно с помощью nvcc

Компилятор nvcc самостоятельно обрабатывает файлы с расширением cu и cuh.

Как и любой проект, программы на CUDA рекомендуется собирать с помощью утилиты make. Тем более, что в CUDA SDK есть много примеров программ с заготовленными файлами сборки, которые легко можно адаптировать под себя.

Слайд 21
Когда вы пишете программу, требуется указать, что должно быть ядром. Для этого вводятся указанные на слайде модификаторы функций. Модификатор __host__ подразумевается по-умолчанию, если не поставлен никакой другой. Модификатор __global__ указывает, что функция является ядром.

Слайд 22
На модификаторы функций накладывается ряд ограничений, которые представлены на слайде. Обратите внимание, что ядро должно возвращать void.

Слайд 23
Модификаторы типов определяют то, где будет располагаться память под переменную и время жизни переменной.

Слайд 24
На модификаторы типов также накладываются ограничения.

Слайд 25
Пример определения и запуска ядра приведен на слайде.
Здесь стоит обратить внимание еще на одно расширение языка си — угловые скобки. Конечно можно обойтись и без них, но тогда придется писать программу, которая напрямую общается с драйвером. Программа при этом будет менее наглядная.

Параметр Dg и Db определяют параметры сети и потокового блока. Остальные параметры являются не обязательными и могут опускаться.

Слайд 26
Внутри ядра определены «переменные», по которым можно определить положение потока в потоковом блоке и потокового блока в сети. Эти «переменные» нельзя модифицировать и от них нельзя получить адрес.

Слайд 27
Пример ядра, складывающие два массива поэлементно и помещающие результат в третий массив. Запускается на «линейке».

Слайд 28
Для удобства разработчика предопределено большое количество типов данных.

Слайд 30
Если в компьютере установлено несколько устройств с GPU, поддерживающих CUDA, то можно запросить параметры каждого из устройств и выбрать то, на котором будут запускаться ядра. Эти вызовы должны быть сделаны до первого вызова любой другой функции по общению с GPU. Переключаться между GPU возможности нет. Чтобы задействовать несколько устройств, надо писать многопоточные программы.

Слайд 31
Параметры устройства возвращаются в структуру, поля которой представлены на слайде. Для начала вам может быть интересны только количество мультипроцессоров и объем глобальной памяти. Многие другие поля, такие как количество регистров и объем разделяемой памяти, могут потребоваться для оптимизации ядра.

Слайд 32
Поскольку потоки на графическом процессоре могут обращаться только к памяти GPU, то прежде чем запустить ядро надо на устройстве выделить память как минимум под результат. Входные данные при этом надо еще и инициализировать.

На хосте память может выделяться или стандартной функцией malloc, или специальной cudaMallocHost. В последнем случае и освобождаться тоже специальной функцией.

На устройстве память может выделяться несколькими способами в зависимости от цели. cudaMalloc выделяет непрерывную область памяти заданного размера. cudaMallocPitch используется для выделения области памяти под двумерные массивы, когда надо чтобы

начала строк массива были выровнены на определенную границу, удобную для GPU.

Слайд 32
После выделения памяти с помощью cudaMallocPitch пользователь получает растояние между началами строк в байтах.

Специальный вызов cudaMallocArray предназначен для выделения памяти, которая в дальнейшем может быть связана с текстурам. Прямой доступ к этой памяти у пользователя нет. *//TODO: проверить для последних версий CUDA*

Слайд 33
Для того, чтобы разместить исходные данные на GPU или забрать результат, надо произвести копирование памяти. Направление копирования задается последним параметром. Название констант, задающих направление информативно.

Слайд 34
Чтобы программисту не усложнять код дополнительными операциями синхронизации потоков, ему предоставлены атомарные операции. Стоит сказать, что выполнение этих операций вызывает дополнительные накладные расходы и программа существенно может терять в производительности.

Слайд 35
Стоит напомнить, что GPU ориентирован на работу с float (одинарная точность). Поэтому вычисления могут идти с большей погрешностью, чем на центральном процессоре, где компилятор неявно может приводить тип float к double на математических операциях. Информация о точности вычислений представлена в документации, равно как и быстрые аналоги некоторых математических функций.

Слайд 36
В обычном понимании текстура — это некая картинка, которая натягивается на объект. Для GPU текстура — это способ доступа к данным.

Слайд 37
Каждый кластер обработки текстур имеет свой блок работы с текстурами и текстурный кэш. За счет него скорость обращения к глобальной памяти через текстуры существенно выше, чем на прямом обращении. Кроме того в кэш помещаются данные, которые в памяти лежат рядом в 2D области.

Слайд 38
Дополнительная стадия конвейеризации в текстурах дает возможность производить интерполяцию данных между узлами сетки, сворачивать текстуры в тор, работать в истиных или нормализованных координатах.

Слайд 39
Для выборки данных из текстур пользователю доступны 4 функции.

Слайд 40, 41

Слайд 42
В зависимости от способа выделения памяти, которая в дальнейшем привязывается к текстурам, текстуры обладают, или не обладают некоторыми свойствами.

Слайд 43
Работа с текстурами ведется в последовательности, которая указана на слайде. Придерживаясь правила хорошего тона надо не забывать освобождать ресурсы (память и текстуру)

Слайд 44, 45
Пример программы, которая поворачивает на заданный угол картинку вокруг ее центра.

Слайд 46
В комплекте к CUDA toolkit идет библиотека, которая должна облегчить жизнь разработчику. По-умолчанию библиотека не собрана.

Слайд 47
Макрос CUT_DEVICE_INIT может использоваться для того, чтобы разобрав командную строку выбрать указанное GPU устройство или по-умолчанию выбрать самое быстрое.

Таймер может использоваться для профилирования участков программы.

Слайд 48
Поговорим об оптимизации программы для графических процессоров, т.е. ядер.

Слайд 49
Прежде чем оптимизировать программу, надо понять в как она себя ведет. Для этого существует несколько способов. Самый простой это замеры времени, например функцией clock(), представленной на слайде, или макросами из CUDA Utility Library, но такая информация мало что даст. Более полная информация может быть получена из счетчиков на GPU. Для этого надо установить переменную окружения CODA_PROFILE в единицу. Остальные две задают имя сайла с параметрами профилирования и имф выходного файла.

Слайд 50
Файл конфигурации — это перечень счетчиков,которые интересны. По одному счетчику на строку.

Слайд 51
Количество счетчиков постоянно увеличивается и на текущий момент их более 20. Данные со счетчиков, выделенных синим цветом, собираются по-умолчанию.

Слайд 52
После запуска программы, если она отработала корректно, то отчет о профилировании будет доступен в текстовом файле. Конечно, при большом желании его можно анализировать, но гораздо удобнее анализировать информацию в графическом виде.

Слайд 53
Профилировщик, написанный на QT идет в комплекте к CUDA SDK. Вся информация может быть представлена в виде таблиц или столбиковых диаграмм. При этом есть возможность сравнивать различные версии одной программы.

Слайд 54
Анализируя результаты профилирования стоит принимать во внимание не абсолютные значения счетчиков, а их приращение. Так, надо стремиться к уменьшению несогласованных операций обращения к памяти, уменьшению количества ветвлений и, возможно, к росту occupancy.

Слайд 55
Чтобы еще до запуска программы оценить что может являться узким местом в программе, можно использовать Occupancy калькулятор. По данным о компиляции ядра (требуемое количество регистров и разделяемой памяти) и количеству потоков в потоковом блоке калькулятор может оценить загрузку потоковых мультипроцессоров.

Слайд 56
Этот калькулятор реализован в виде таблицы Excel. Вы задаете необходимые параметры и на выходе получаете то, что лимитирует рост производительности.

Слайд 57
Также отображаются три графика из которых можно понять в какую сторону можно двигаться для устранения этого критического параметра.

Слайд 58
В конце июля 2010 года был выпущен продукт Nsight — программу для разработчиков программ на GPU под Microsoft Visual Studio. Отлаживать программы, искать и устранять узкие места в них.

Слайд 59
Прежде чем инструкция будет исполнена, необходимо выполнить чтение операндов, а после исполнения сохранит результат. Таким образом оптимизация программы заключается в оптимизации обращения к памяти и модификации кода таким способом, чтобы по возможности использовались более быстрые инструкции.

Слайд 60
По поводу времени выполнения инструкций в документации приводятся данные, представленные на слайде. Стоит обратить внимание, что операция деления чисел с плавающей точкой занимает больше тактов, чем операция умножения этих чисел. Таким образом, если где-то в коде встречается деление на 2.0f, то эту операцию лучше заменить на умножение на 0.5f.

Слайд 61
Ветвления и условные переходы так же вносят задержки в выполнение кода. Если в одном варпе есть две ветви, то сначала выполняется одна ветвь, затем другая. Можно попробовать уменьшить количество ветвлений за счет предвычислений или заменой условных операторов на бинарные.

Слайд 62
Различается и скорость доступа к разным типам памяти. Глобальная память имеет наименьшее время доступа. Поэтому к ней, если есть возможность, лучше обращаться через текстуры. Другой вариант — скопировать необходимые данные в разделяемую память, там их обработать и вернуть в глобальную.

Слайд 63
Задержки при обращении к глобальной памяти могут происходить, если два потока обращаются к одному банку памяти по разным адресам. К памяти одновременно могут доступаться 16 потоков, при этом вся глобальная память распределена по 16 банкам памяти в стиле round-robin. Чтобы минимизировать конфликты необходимо отслеживать шаблоны доступа к памяти, использовать «правильные» функции для выделения памяти под массивы, использовать текстуры.