[Слайд 1]
Добрый день, уважаемые слушатели! Сегодня мы с вами поговорим про модель вычисления NVIDIA CUDA.

[Слайд 2]
Этот доклад охватывает лишь основные моменты затрагиваемой темы и изученных мною источников. Поскольку тема данного доклада является чрезвычайно сложной, вопрос на экзамен, который я вам подготовил, тоже, как ни странно, сложный, все материалы, использованные в данном докладе, в том числе эта презентация, текст, который я читаю, ссылки на источники, дополнительную литературу, любая сопутствующая информация опубликована по QR коду на экране, и оформлена в виде GitHub репозитория. Обязательно загляните туда, особенно если вас заинтересует данная область. Ну, а теперь, непосредственно к теме.

[Слайд 3]
Скорее всего, многие из вас уже наслышаны про данную платформу, возможно, вы даже знаете, где вы ее используете в повседневной жизни. Тем не менее, для тех, кто все-таки не знает, сначала мы рассмотрим, где она может применяться.

[Слайд 4]
CUDA призвана значительно ускорить некоторые вычисления. Так, с CUDA вы могли столкнуться, используя такие системы как MATLAB или Wolfram Mathematica.

[Слайд 5]
Помимо этого, компания Adobe поддерживает в своих продуктах аппаратное ускорение для рендеринга графики Mercury Playback с применением CUDA. Соответствующую статью от NVIDIA вы можете прочитать по ссылке внизу слева.

[Слайд 6]
Собственно говоря, это еще далеко не все. Вы, как разработчики могли сталкиваться с вычислениями на графическом ускорителе в машинном обучении, и десятки фреймворков для данных задач оптимизированы под CUDA.

[Слайд 7]
Ну и даже в вычислительной биологии и медицине сегодня используют эту технологию. Список далеко не полный, и продолжать его можно очень долго.

[Слайд 8]
Но почему так произошло? Чем же нам так не угодил тот же CPU?
Как известно, любая современная вычислительная техника определяется такой характеристикой, как ее производительность, то есть способность быстро производить какие-либо вычисления. Так уж сложилось, что науке и бизнесу постоянно нужно, чтобы купленная ими железка работала в режиме, ну так называемого «быстрее, еще быстрее и еще быстрее», тем самым, поддерживая их уровень конкурентоспособности. Однако проблема очевидна: угодить такой хотелке от заказчиков крайне затруднительно. И как раз несколько десятилетий назад в эту проблему уперлись разработчики центральных процессоров. В чем же суть?

[Слайд 9]
Давайте немного поговорим о том, а как вообще мы можем повысить

производительность CPU? Правильно, например, поднять тактовую частоту. Так вот, суть заключается в том, что при попытке разогнать частоту тактового генератора, аномально росло энергопотребление и тепловыделение процессора. Такое решение никого не устраивало – это бы требовало еще больших затрат на системы охлаждения. Еще один способ поднять производительность CPU заключается в размещении нескольких ядер на одном чипе, а также уменьшение техпроцесса, ну и все в этом духе. Но и здесь мы очень быстро упираемся в физические ограничения, ведь печатая все больше транзисторов на одной плате, они рискуют чаще отказывать, растет число брака, это всегда требует серьезных реформ в производстве, но даже если попытаться с этим побороться, бесконечно уменьшать техпроцесс невозможно физически.

[Слайд 10]
С другой стороны, на помощь спешат графические ускорители. Обработка графики изначально строилась на том, что нам нужно обработать большую группу полигонов и преобразовать их в большое число пикселей на экране. Каждый такой элемент обрабатывается независимо от остальных, а поэтому эти вычисления крайне легко распараллелить. Поэтому на уровне архитектуры графических процессоров подразумевается порядка нескольких сотен вычислительных устройств, на которых можно очень быстро что-то посчитать.

[Слайд 11]
Отсюда и родилась идея использовать GPU для совершено «неграфических вычислений». Идея проста: у нас есть задача, которая подразумевает очень много математики с каким-то ограниченным набором данных. Чтобы сделать такое вычисление на процессоре, мы можем использовать какой-нибудь параллельный алгоритм, однако он будет работать достаточно долго: во-первых, потому что в процессоре не очень много ядер, а во-вторых, потому что переключение между контекстами для разных потоков в процессоре крайне дорогая операция. Если же точно такую же задачу дать графическому ускорителю, то он с ней справится намного быстрее, за счет того, что распараллеленные зачади будут исполняться одновременно на сотнях процессорах, а переключение между потоками происходит значительно быстрее, и это заложено на уровне архитектуры.

[Слайд 12]
К тому же графические ускорители не подвержены проблеме масштабируемости, в том виде, в котором мы ее знаем для CPU. Ведь, чтобы ускорить вычисления на графическом ускорителе, мы можем просто докинуть еще немного процессоров, и они тоже будут что-то считать в параллель вместе со всеми. Компания NVIDIA сегодня от части так и поступает, выпуская карточки для разной целевой аудитории и разного ценового сегмента. В перспективе платформа может поддерживать дополнительные вычисления (атомарные, с двойной точностью).

[Слайд 13]
Все это рождает нам график, который вы видите справа. Вычисления на видеокартах могут показывать производительность в несколько сотен раз быстрее, чем процессоры общего назначения. Тем не менее, хочется сделать замечание, что подобная производительность достигается только в синтетических случаях,

специально заточенных под параллельные вычисления. Любая программа, не оптимизированная под модель вычислений на GPU, не ощутит никакого прироста производительности, и может спокойно запускаться на центральном процессоре.

[Слайд 14]
Тем не менее, когда мир догадался до того, что можно вот так применив специальное аппаратное обеспечение кратно ускорить вычисления, графические ускорители были узко заточены под ускорение графики на цветных мониторах. В связи с чем, для достижения озвученного ранее результата приходилось, скажем, «маскировать» программы под обработку графики, часто с применением OpenGL и DirectX. Понятное дело, у всего этого костыля были большие издержки, во-первых, по написанию такого кода и по его поддержке, ведь от разработчика требовалось знать, как работать с графикой, хотя ему это совершенно было не нужно, а во-вторых, написанный код выполнял лишние действия, которые были присуще графическим вычислениям, но были бесполезны для программ, на самом деле не связанных с графикой.

[Слайд 15]
На это все обратили внимание ребята из компании NVIDIA, и в 2006 году миру была представлена первая карточка GeForce 80, которая представляла открытую и унифицированную модель вычислений на GPU для общего назначения, а разработчикам открыла в экспериментальном режиме компилятор, который позволяет писать такого рода программы, которые заточены под высокопроизводительные параллельные вычисления. Естественно, все это было доступно пока только на карточках от компании NVIDIA.

[Слайд 16]
Ключевым архитектурным решением этой серии был так называемый стриминговый процессор, который умел производить вычисления с float-типами данных, эффективно работал с потоками ввода/вывода данных, объединялся с другими стриминговыми процессорами для производства массивных параллельных вычислений, ну а за планировку задач отвечала запатентованная технология NVIDIA GigaThread. Архитектура получила название Tesla, отсюда и название карточек GT и GTX. За дополнительной исторической справкой приглашаю ознакомиться с отчетом NVIDIA о видеокарте GeForce 8800, там есть много интересного.

[Слайд 17]
Собственно, данная технология получила название Compute Unified Device Architecture, или просто CUDA. Она позволяет организовать очень эффективные параллельные вычисления общего назначения, отсюда кстати зародился такой термин как GPGPU – General Purpose GPU – вычисления общего назначения на графическом ускорителе. За фундамент CUDA была взята архитектура SIMD – Single Instruction Multiple Data. SIMD это архитектура параллельных вычислений, при которых несколько процессоров выполняют одну и ту же инструкцию для разных входных данных. NVIDIA слегка доработала эту архитектуру, получившееся решение назвали SIMT – Single Instruction, Multiple Threads.

[Слайд 18]
Собственно говоря, подобный подход к проектированию, позволяет нам как можно

меньше инструкций тратить на организацию Control Flow, вместо этого больше сосредоточиться именно на обработке данных. Сравнивая с CPU, последний сосредоточен именно на однопоточные вычисления с обработкой самых разнообразных сценариев работы, что не свойственно для графических ускорителей. Именно поэтому GPU не призваны заменить или как-то соперничать с CPU. Об этом же говорит и то, что все вычисления, запущенные на GPU с помощью CUDA, управляются именно центральным процессором.

[Слайд 19]
Прежде, чем мы продолжим, давайте определимся с терминологией.

[Слайд 20]
Начнем с того, что программист пишет функцию, которая будет исполняться на графическом процессоре, например, умножение матриц. Эта функция будет называться CUDA Kernel, или CUDA ядро.

[Слайд 21]
К сожалению, великому и богатому русскоязычному сообществу не повезло, и CUDA ядром также еще называется блок с арифметико-логическим устройством, который будет исполнять потоки на базе CUDA Kernel. По-английски этот блок называется CUDA core, и, чаще всего, как раз это и имеют в виду, когда рассказывают про CUDA ядра. В дальнейшем я буду называть вычисляемую на GPU функцию по-английски, то есть CUDA Kernel, дабы избежать путаницы. CUDA Kernel становясь той самой Single Instruction, образует огромное число CUDA потоков, то есть Multiply Threads.

[Слайд 22]
Едем дальше. CUDA потоки объединяются в блоки потоков, так называемые CUDA Thread Blocks. Один CUDA Thread Block исполняется на одном стриминговом мультипроцессоре (сокращенно SM). Ранее мы уже сталкивались с понятием стримингового процессора, так вот мультипроцессор содержит в себе эти самые стриминговые процессоры (они же CUDA ядра), также в нем присутствуют специальные ядра для операций работы с памятью, блоки для ускорения некоторых математических вычислений, например синуса, кэш память, планировщик задач, и многое другое, это в зависимости от рассматриваемой нами архитектуры конкретной видеокарты. Мы обязательно вернемся к этому вопросу более подробно…

[Слайд 23]
…А пока в дополнение скажем, что блоки потоков объединяются в трехмерную сетку, так называемая CUDA Thread Grid, и вся эта сетка уже может исполняться на CUDA-совместимой видеокарте. В сетке может в общей сложности содержаться несколько сотен, тысяч или даже десятков тысяч потоков, разбитые по блокам.

[Слайд 24]
Ну и немного поговорим про то, что Вам доступно как разработчику. В составе NVIDIA CUDA Toolkit вам предоставляется много классных плюшек для хорошего старта: компилятор промышленного стандарта для языков программирования C и Fortran, несколько очень полезных высокоуровневых библиотек для вычислений на GPU (например, для линейной алгебры), различные профайлеры и средства мониторинга, а также вся прикладная программная документация и большое количество учебных

материалов. NVIDIA заботится о своих разработчиках, и предоставляет все данные ресурсы совершенно бесплатно. И вы можете их посмотреть, как дополнительную литературу, если заинтересуетесь темой программирования на графических процессорах.

[Слайд 25]
Теперь, когда мы получили базовое введение в терминологию, поговорим поподробнее о том, как происходят параллельные вычисления.

[Слайд 26]
И начнем мы с организации памяти в CUDA-совместимых GPU. Обратите, пожалуйста, внимание, это же и будет Вашим вопросом на экзамене.

[Слайд 27]
Каждый стриминговый мультипроцессор включает в себя блок из 65536 32-разрядных регистров (256KB), разделяемая память и кэш первого уровня, которые на плате расположены в одном и том же месте и в зависимости от конфигурации могут быть распределены по разному, например, в архитектуре Ферми эта память могла делиться либо 16КБ разделяемой памяти и 48КБ кэша первого уровня, либо наоборот. В современных же видеочипах в SM располагается до 192КБ такой гибридной памяти. Также присутствует отдельная кэш-память только для чтения, в том числе память инструкции ядра, текстурные кэши и пр.

[Слайд 28]
Вне стримингового мультипроцессора располагается кэш второго уровня: в современных видеокартах расположено 40МБ такой кэш-памяти. L2 кэш доступен во всех мультипроцессорах любому исполняющемуся потоку. Ну и замыкает наш парад всякой разной памяти глобальная общая память GPU. Большая ее часть представляет из себя обычную DRAM, в разных видеокартах ее встроено по-разному: от 4 до 40ГБ. Соответственно туда загружаются все данные для вычислений, а L2 кэш существует для того, чтобы ускорить к ней доступ.

[Слайд 29]
Остановимся на каждом типе доступной нам памяти поподробнее:

30. Регистры: Самая быстрая память, доступная CUDA ядру. При размещении блока потоков на стриминговом мультипроцессоре регистры назначаются каждому потоку, и они блокируются за ними до конца его исполнения. Гарантируется, что на стриминговом мультипроцессоре будет размещено не больше блоков, чем доступно для резервирования регистров.
31. Специальные регистры: они хранятся в SM и содержат некое особое значение, заранее определенное аппаратно. Такие регистры часто хранят индекс блока, индекс потока, размер сетки и всю подобную информацию, которая позволяет удобнее работать с памятью, а также они нам доступны как локальные параметры в Kernel функции. Для каждого потока создается своя копия таких данных, даже несмотря на то, что некоторые значения могут дублироваться.
32. Разделяемая память: она также размещена на SM и предлагает возможность обмениваться данными между разными потоками (однако это требует дополнительной синхронизации, но о ней мы поговорим немного позже).

Ключевая особенность разделяемой памяти состоит в том, что это память с очень высокой полосой пропускания: одновременно к ней может иметь доступ до 32 потоков. Достигается за счет ее разбиения на 32 банка памяти. Стоит также упомянуть, что, если в один банк пытаются одновременно попасть 2 потока, возникнет конфликт, приводящий к уменьшению производительности. Разделяемая память — это крайне интересная тема, и, я думаю, что вы поймете, зачем оно нам нужно, когда мы разберем, как исполняются вычисления.

33. Локальная память: сама по себе локальная память располагается в DRAM с достаточно медленной (порядка 100 раз меньше, чем разделяемая память) скоростью доступа. Ускорение достигается за счет использования кэшей 1-го и 2-го уровня. В локальную память кладется все, что плохо лежит в регистрах – стек вызовов, массивы. Для каждого потока выделяется своя локальная память, другие потоки не имеют к ней доступа.
34. Глобальная память: глобальная память тоже располагается в DRAM и, так же, как и локальная память, ускоряется при помощи кэшей 1-го и 2-го уровня. Тем не менее, отличия все же присутствуют: содержимое глобальной памяти доступно всем потокам в любой момент времени. И здесь, к сожалению, присущи все проблемы кэширования при многопоточности, какие обычно могут на этой почве возникнуть. Например, поток может закэшировать часто получаемое значение из глобальной памяти, а второй поток его тем временем обновит – эти изменения не будут замечены. Как с этим работать, немного поговорим уже ближе к концу нашего доклада.
35. Константная память: хранится в DRAM и кэшируется константными кэшами в SM. Центральный процессор имеет возможность что-то записать в нее перед стартом вычислений, потоки же могут только обращаться к константной памяти только для чтения.
36. Текстурная память: она немного выходит за рамки нашего разговора про вычисления общего назначения, и в целом является темой для отдельного разговора, однако про нее хочется сказать, что CUDA может выиграть за счет оптимизации, основанной на способе организации данных в текстурной памяти и текстурных кэшах соответственно.

[Слайд 37]
Сводная таблица описанной мною памяти приведена сейчас на слайде. Это вся доступная нам память, как разработчикам. Также, вам, как разработчикам, хочется сказать, что компилятор NVIDIA CUDA (тот же nvcc) выполняет колоссальную работу по оптимизации ресурсов памяти, но при должной сноровке разработчика, никто вам не мешает окунуться в низкоуровневый код и управлять памятью вручную.

[Слайд 38]
Что ж, теперь перейдем к следующей нашей теме – организация потоков и как происходят сами вычисления и обработка наших данных.

[Слайд 39]
Перед тем, как мы начнем, введем некоторые уровни архитектуры CUDA. Как вы

можете понять, у любого современного железа как продукта двухэтапного производства есть некая системная платформа. Так вот, здесь она тоже есть, и носит название Parallel Thread Execution (а сокращенно, PTX). Код, компилируемый на CUDA-расширенных языках (C/Fortran), генерируется в PTX ассемблерную систему команд (ISA), лишь затем происходят непосредственно вычисление на реальной GPU. Пример PTX-кода вы можете видеть на экране. В состав PTX также входит виртуальная машина, ведь у нас может быть много разных карточек, и под каждую нативные аппаратные инструкции будут свои. За перевод PTX кода на реальную карточку отвечает драйвер.

[Слайд 40]
Ну и остальные цели PTX указаны на слайде (https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#goals-of-ptx).

[Слайд 41]
Когда мы написали нашу Kernel функцию, указали, сколько потоков должно заниматься ее исполнением, потоки группируются в блоки, блоки группируются в сетку, и все это хозяйство отправляется на GPU, где нас встречает некоторое количество стриминговых мультипроцессоров, готовых для исполнения наших потоков. Потоки из одного блока будут исполняться на одном SM. Как только потоки из блока заканчиваются, новый блок поступает на исполнение свободному мультипроцессору с помощью планировщика GigaThread.

[Слайд 42]
Как же SM исполняет выданный ему блок? Весь блок раздробляется на мелкие группы потоков, по 32 штуки, именуемые «варпом». Каждому потоку назначается ID, в первом варпе поток имеет ID 0, далее это число нарастает на 1 для каждого следующего потока. Варп разделяет 32 последовательно идущих потока, начиная с нулевого.

[Слайд 43]
Все потоки внутри одного варпа исполняют одну и ту же (буквально одну и ту же) инструкцию в параллельном режиме. Это важно, ведь из этого следует, что ваша Kernel функция будет исполняться наиболее эффективно только в том случае, если в ней сведено к минимуму количество условных операторов. Если же в потоках возникают инструкции, связанные с условными переходами, то выполнение потоков продолжится только на тех ядрах, где условие выполнилось. В итоге все варпы заканчивают свое исполнение в одно и тоже время, и на их место ставится другой варп. Если все потоки начали выполнять операцию, связанную с доступом к памяти, планировщик поставит на вычислительные мощности другой варп, который может что-то сделать без доступа к памяти. Это существенно скрывает задержку вычислений, а вообще говоря, планировщик всегда старается поставить на исполнение варп, который может немедленно совершить полезную работу.

[Слайд 44]
В то время, когда на CPU это очень дорого переключить контекст и продолжить работу, нет никакой сложности сделать то же самое на GPU, ведь информация о

варпе (текущий счетчик инструкций, состояние регистров) хранится в мультипроцессоре на протяжении всего времени жизни варпа.

[Слайд 45]
Планировщик может нагружать не только CUDA ядра, как было показано ранее. В стриминговом мультипроцессоре также присутствует 16 вычислительных блоков, отвечающие за операции с памятью, а также 4 блока для быстрого вычисления математических функций, например тригонометрии или экспоненты. Все подобные операции выполняются некоторое количество тактов, однако планировщик может запускать варпы на различных элементах SM каждый такт, что позволяет не дожидаться окончания выполнения операций на конкретном узле GPU. К сожалению, мы не знаем, как устроен алгоритм планировщика задач.

[Слайд 46]
Теперь немного поднимемся на уровень выше. Каждому блоку выделяется свой уникальный ID, все блоки формируют двумерную сетку. В блоке же, в свою очередь, потокам может присваиваться уникальный ID в 3 измерениях, и это все регулируется программистом. На слайде приведен пример кода, где регулируется количество измерений и потоков. В современных карточках существует ограничение: в одном блоке может существовать не более 1024 потоков. Но даже это не должно восприниматься как руководство к действию, ведь из-за того, что блок аллоцируется на SM полностью, GigaThread планировщик не сможет аллоцировать столь большой блок на SM, так как на мультипроцессоре может банально не хватить ресурсов для столь большого блока. Все присваиваемые ID хранятся в константном кэше и используются, например, для обработки больших массивов данных. Уникальные ID позволяют определить ту часть массива, за которую отвечает конкретный поток.

[Слайд 47]
Сетки принадлежат определенному контексту. В контексте может быть до 16 сеток, которые одновременно могут исполняться на GPU. Они имеют общее адресное пространство, как следствие потоки из разных контекстов не имеют общего доступа к памяти друг друга. Несмотря на то, что на GPU может быть создано несколько контекстов, только один контекст может быть в одно время исполняться на GPU. Напоминаю, что CUDA Kernel присваивается отдельно каждой сетке.

[Слайд 48]
В заключение поговорим про механизмы синхронизации, которые предлагает нам платформа CUDA. Зачем нужна синхронизация? Все просто, хоть и логически, выглядит все так, что Kernel функция исполняется одновременно для всех потоков, физически это далеко не так. И если нам нужно, чтобы два потока из одного блока как-то обменялись значениями в памяти, то для этого потоки нужно синхронизировать, во избежание состояния гонок, так как эти потоки скорее всего окажутся в двух разных варпах, и неизвестно, как эти варпы будут исполняться. Поэтому для вас, как для разработчиков, существует механизм синхронизации всех потоков внутри одного блока. Он представляет из себя простую барьерную синхронизацию – ни один из потоков блока не перейдет через барьер, пока не

дождется остальных потоков из блока. Синхронизировать можно как операции с разделяемой, так и с глобальной памятью.

[Слайд 49]
Однако, на этом возможности синхронизации заканчиваются. Потоки из разных блоков не синхронизированы, доступ к глобальной памяти, куда пишет другой поток, как правило, небезопасен. Присуще проблема кэширования, о которых уже упоминалось ранее.

[Слайд 50]
Тем не менее, на помощь приходят контексты. Вы можете запустить несколько сеток из разных Kernel функций в рамках одного контекста, которые последовательно обработают содержимое глобальной памяти. Поскольку это вот уже является крайне дорогостоящей операцией, стоит несколько раз подумать, нужна ли вам такая синхронизация, и, если вы можете обойтись без нее, лучше так и сделать. Тем не менее, на данный момент это единственный способ хоть как-то синхронизировать операции с глобальной памятью.

[Слайд 51]
Что же, на этом у меня все. Если среди слушателей были опытные люди, хорошо знакомые с архитектурой CUDA, и вы заметили неточности, озвученные в докладе, обязательно сканируйте QR код, обеденный в красную рамку, чтобы открыть Issue, и сделать эту работу ещё лучше. Также призываю всем присутствующим непосредственно на конференции открыть чат с Telegram ботом и прописать в нем команду /lab1feedback и обязательно оценить этот доклад. Мне будет очень приятно. Всем большое спасибо за то, что прослушали это выступление, если у вас остались вопросы, я готов постараться на них ответить прямо сейчас, ну а остальным ещё раз спасибо за внимание и до скорых встреч!