Нижегородский государственный университет им. Н.И. Лобачевского

Факультет Вычислительной математики и кибернетики

Введение в вычисления общего назначения на GPU

Бастраков С.И.

ВМК ННГУ

bastrakov@vmk.unn.ru

Молодежная научная школа «Суперкомпьютерные технологии и высокопроизводительные вычисления в образовании, науке и промышленности» 5 – 8 ноября 2014 г., г. Нижний Новгород

Содержание

- Почему GPU?
- Обзор архитектуры GPU
- Обзор средств программирования для GPU
- Одновременное использование CPU и GPU для гетерогенных вычислений

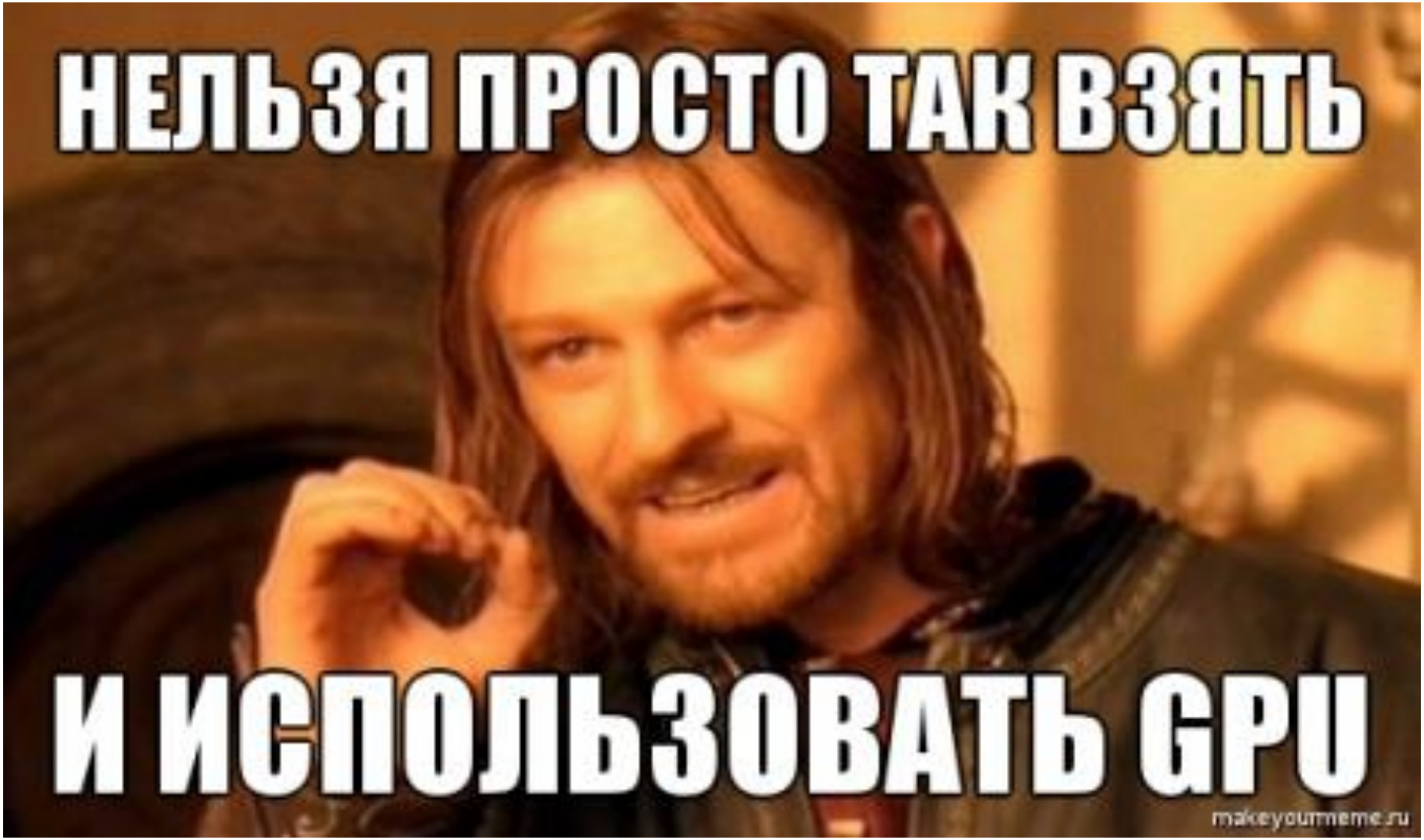
Мотивация

"GPU уже достигли той точки развития, когда многие приложения реального мира могут с легкостью выполняться на них, причем быстрее, чем на многоядерных системах. Будущие вычислительные архитектуры станут гибридными системами с графическими процессорами, состоящими из параллельных ядер и работающими в связке с многоядерными CPU".

Профессор Джек Донгарра (Jack Dongarra)
Директор Innovative Computing Laboratory
Университет штата Теннесси

Источник: http://www.nvidia.ru/page/gpu_computing.html

Демотивация



Почему GPU?

Гетерогенные вычисления

- **Гетерогенные (гибридные) вычисления** (*heterogeneous computing*) – вычисления с использованием **разнородных вычислительных устройств**.
- В качестве таких устройств могут выступать:
  - центральные процессоры (CPU);
  - Сопроцессоры Intel Xeon Phi;
  - графические процессоры (GPU);
  - специализированные процессоры (например, DSP);
  - ускорители;
  - ПЛИС.
- Наиболее популярной на текущий момент гетерогенной связкой является **CPU + GPU**.
- Набирает популярность CPU + Xeon Phi.

GPGPU

- **GPGPU** (*general-purpose computing on GPU*) – вычисления **общего** назначения на **графических процессорах**.
- В качестве вычислений общего назначения могут выступать и задачи, связанные с компьютерной графикой, если они не ограничиваются использованием стандартного графического конвейера (например, метод трассировки лучей).
- Потоковая обработка, используемая в графическом конвейере, может эффективно использоваться во многих вычислительных задачах общего характера.

Типичные области применения GPGPU

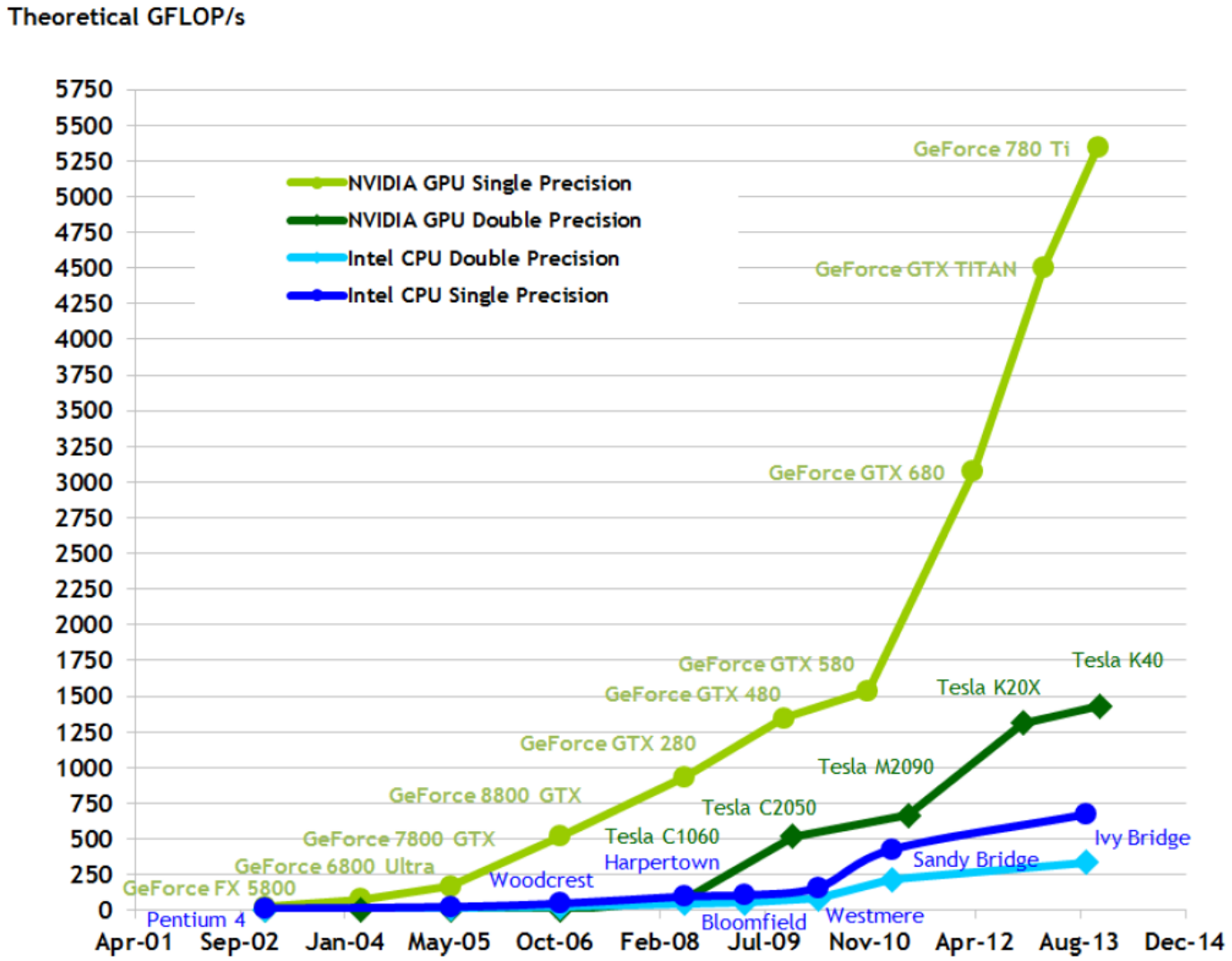
- Физическое моделирование
- Визуальные эффекты
- Финансовая математика
- Вычислительная биология, химия
- …

- Примеры практического применения:

http://www.nvidia.com/object/cuda_showcase_html.html

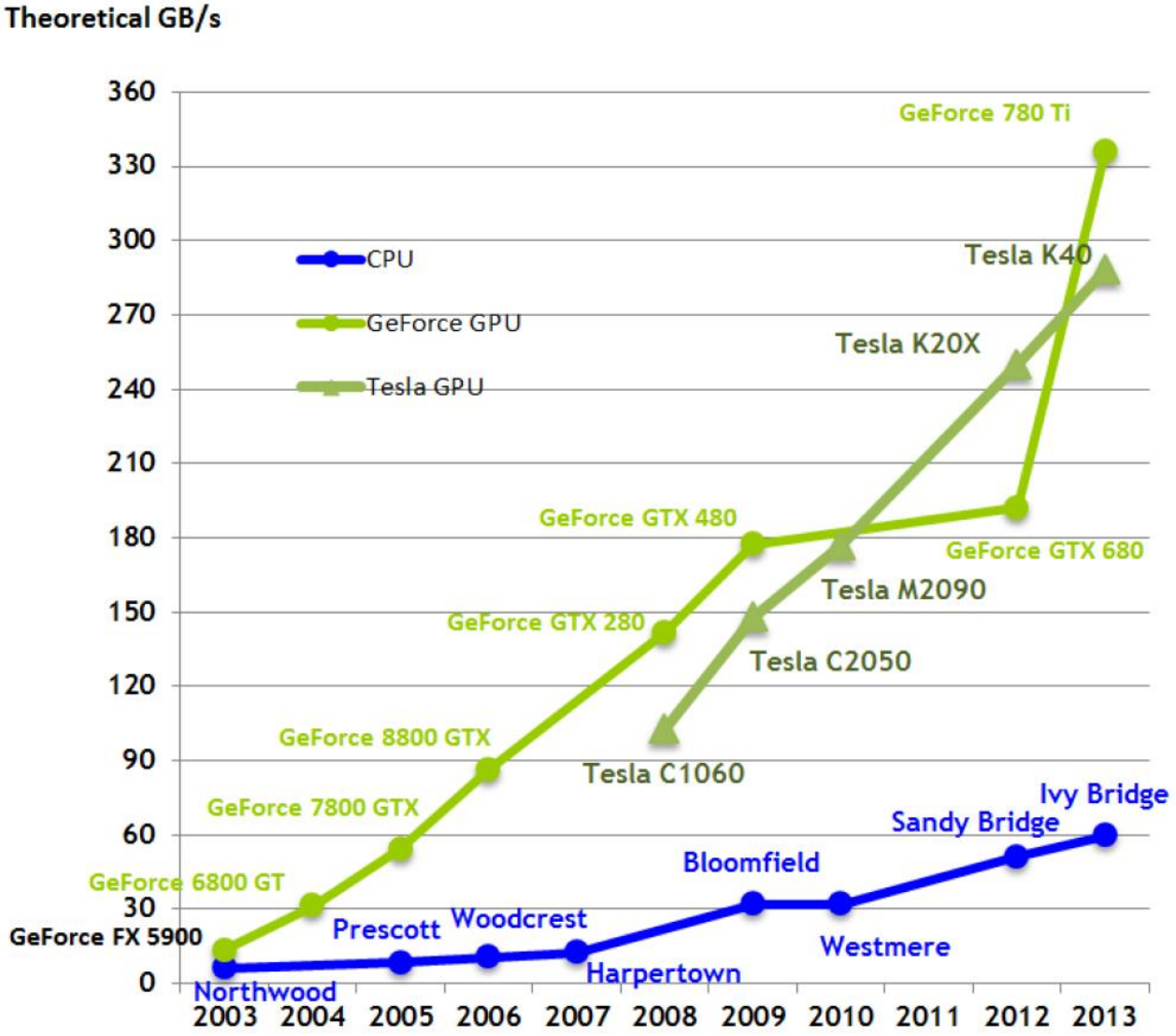
http://developer.amd.com/zones/openclzone/pages/openclappexamples.aspx

Пиковая производительность CPU и GPU



Источник: NVIDIA CUDA C Programming Guide v. 6.5

Пропускная способность памяти CPU и GPU



Источник: NVIDIA CUDA C Programming Guide v. 6.5

Использование GPU в суперкомпьютерах

- Около 10% машин в списке Top500 (июнь 2014) используют GPU.
- Они обеспечивают около 20% от общей производительности всех машин в Top500.
- Мощнейший суперкомпьютер России «Ломоносов» использует GPU.

Современные GPU

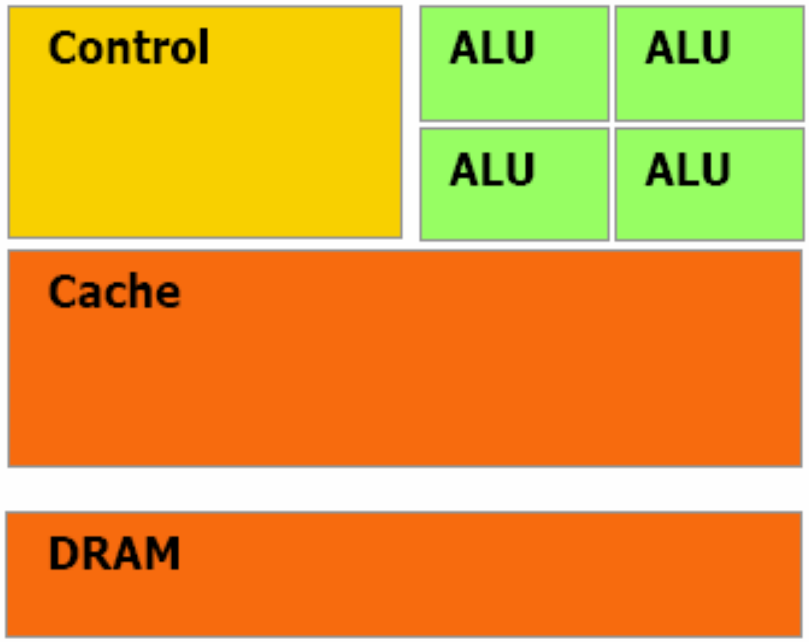
- Современный GPU – **массивно-параллельный многоядерный процессор**.
- Высокая производительность и пропускная способность памяти.
- Хорошо подходит для решения многих классов вычислительно трудоемких задач.
- Выгодное отношение производительности к цене и потребляемой мощности.
- Развитые средства программирования.

Почему не GPU?

- Не все задачи хорошо распараллеливаются.
- Не все задачи хорошо распараллеливаются на большое количество потоков.
- Не все задачи хорошо подходят для архитектуры GPU.
- Сложность разработки и оптимизации приложений по сравнению с CPU.
- Эффективный перенос всех вычислений на GPU невозможен, однако многие важные задачи могут эффективно задействовать GPU. Большой интерес представляют гибридные системы.

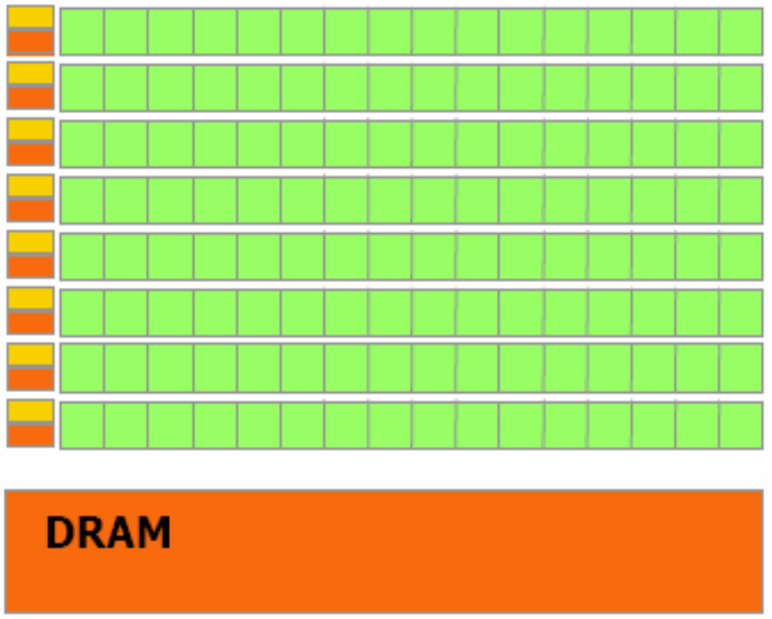
Обзор ахитектуры GPU

Архитектура CPU и GPU



CPU

“cache-oriented”



GPU

“cache-miss oriented”

Источник: NVIDIA CUDA C Programming Guide v. 3.2

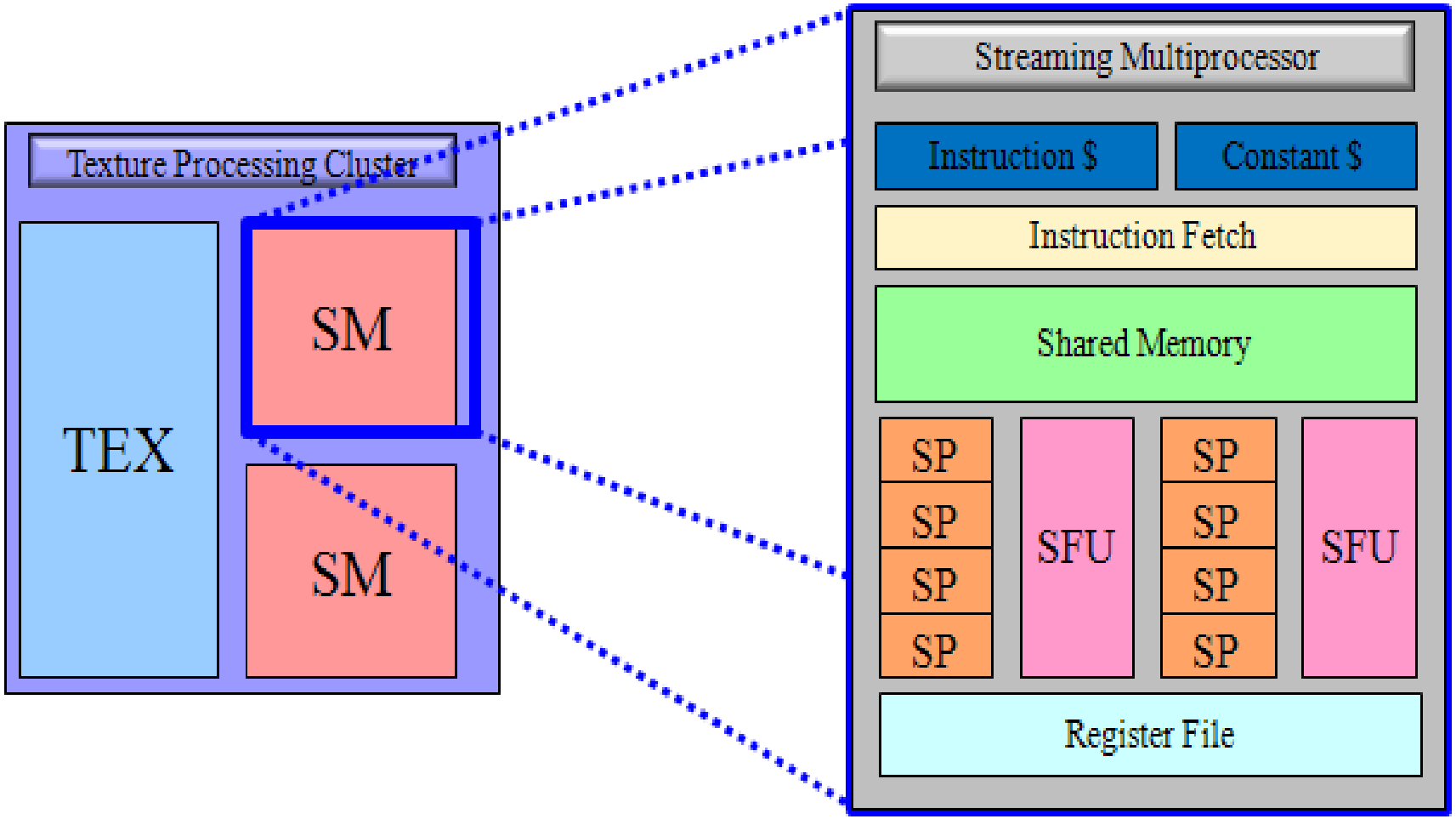
Архитектура CPU и GPU

- GPU предназначен для вычислений:
  - **параллельных по данным**: одна и та же операция выполняется над многими данными параллельно,
  - в которых отношение вычислительных операций к числу операций по доступу к памяти велико.
- По сравнение с CPU:
  - значительно меньше кэша и элементов управления
  - значительно больше вычислительных элементов.
- Латентность памяти покрывается за счет большого количества легковесных потоков.
- Постепенно степень параллелизма CPU и сложность архитектуры GPU повышаются.

Архитектура GPU NVIDIA

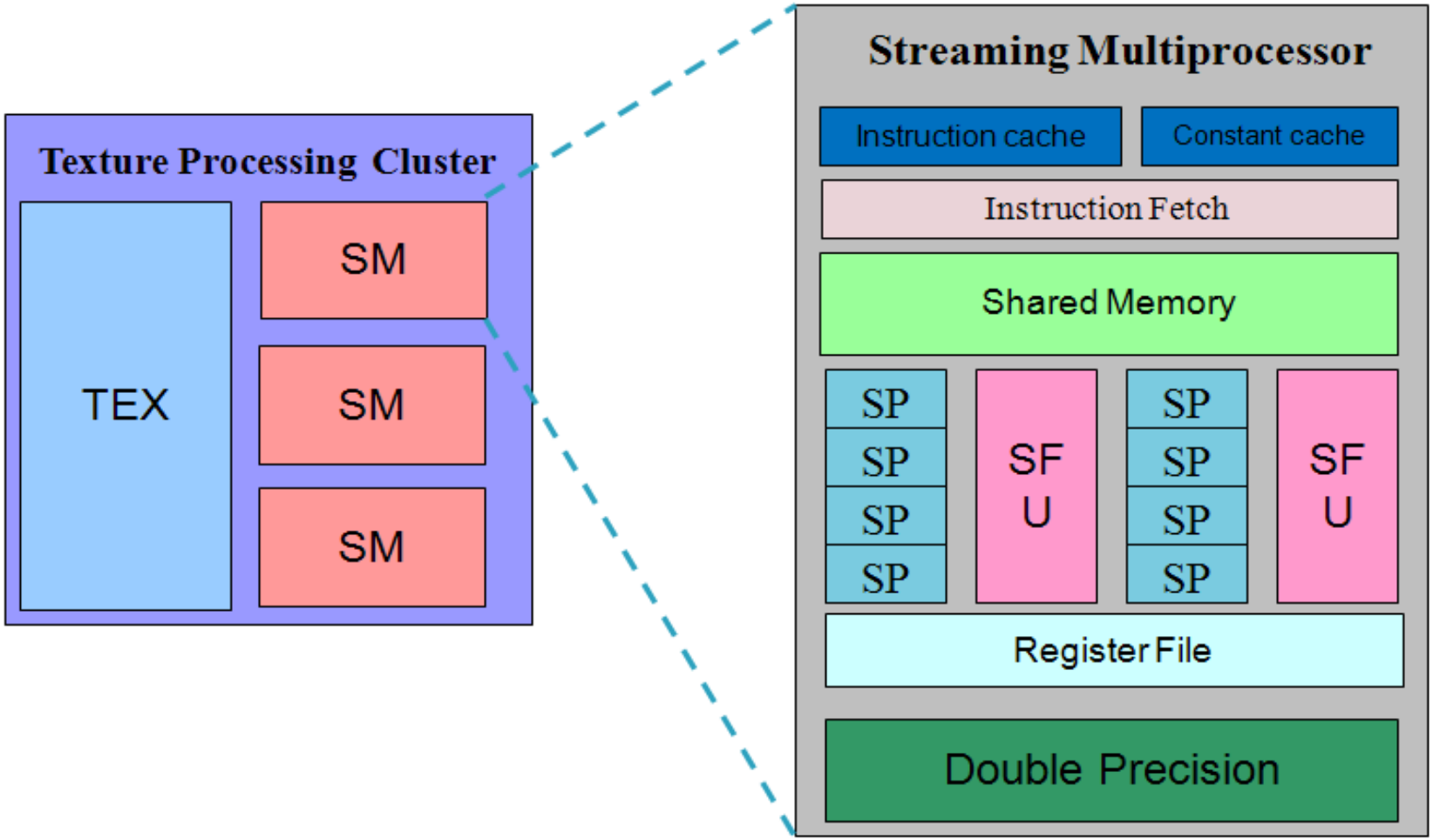
- Состоит из **мультипроцессоров** (*streaming multiprocessor*, *MP*), каждый из которых содержит несколько **CUDA-ядер** (*CUDA core*) и общую для них память.
  - В архитектурах до Fermi аналоги CUDA-ядер назывались **скалярными процессорами** (*scalar processor*, *SP*).
- CUDA-ядра внутри одного мультипроцессора работают как SIMD.
- Очень легковесные потоки, встроенный планировщик.

Мультипроцессор в NVIDIA Tesla 8



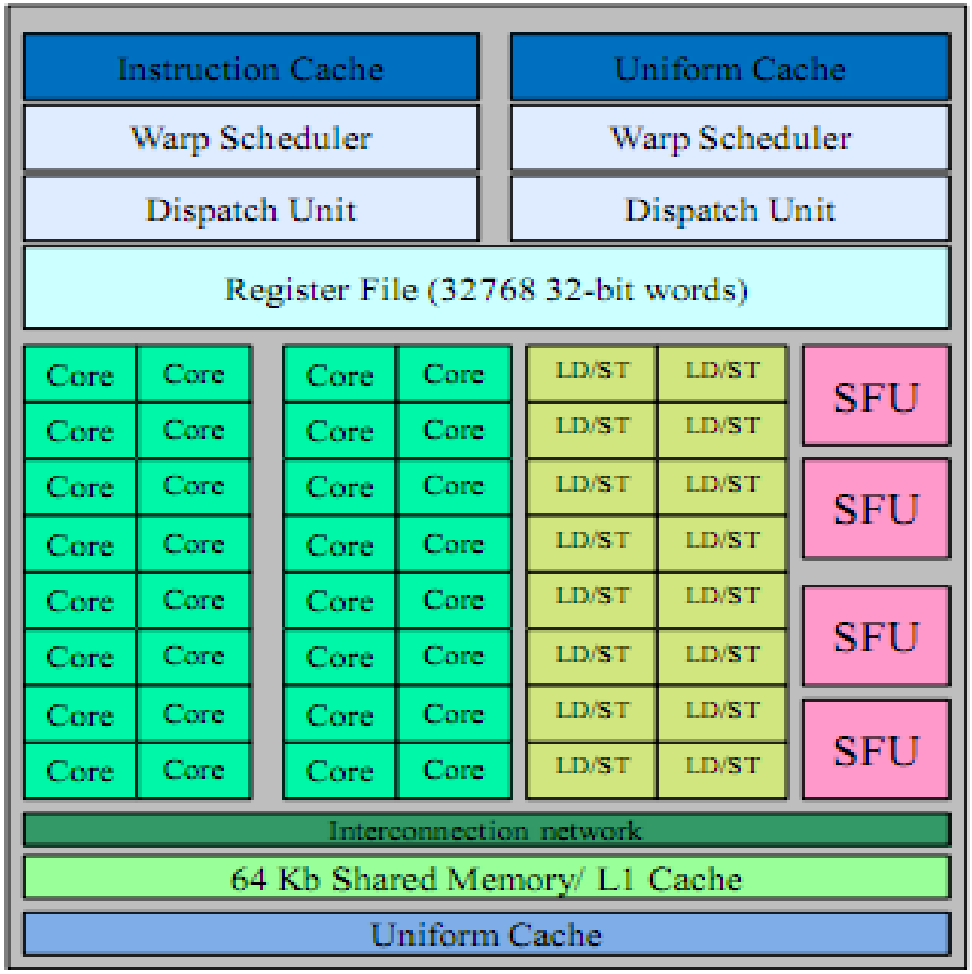
Источник: А.В. Боресков, А.А. Харламов «Архитектура и программирование массивно-параллельных вычислительных систем»

Мультипроцессор в NVIDIA Tesla 10



Источник: А.В. Боресков, А.А. Харламов «Архитектура и программирование массивно-параллельных вычислительных систем»

Мультипроцессор в NVIDIA Fermi



Источник: А.В. Боресков, А.А. Харламов «Архитектура и программирование массивно-параллельных вычислительных систем»

Архитектура NVIDIA Fermi

- Новейшая на текущий момент архитектура NVIDIA.
  - Выход следующей архитектуры Kepler планируется в начале 2012 года.
- Добавлен общий L2-кэш для глобальной памяти.
- На каждом мультипроцессоре добавлен L1-кэш для глобальной памяти.
- В каждом ядре отдельные блоки для вычислений с плавающей точкой и целочисленных вычислений.
- Значительно улучшена производительность арифметики с плавающей точкой двойной точности и SFU.
- Техническая возможность для полной поддержки C++.
- Поддержка ECC.

Иерархия памяти GPU NVIDIA

- **Глобальная** (*device/global*) – общая для устройства.
  - Ее подобласти: **текстурная** и **константная** с соответствующими кэшами.
- **Разделяемая** (*shared*) – общая для всех CUDA-ядер в одном MP.
- **Регистры** (*register*) – (логически) эксклюзивны для CUDA-ядра.
- **Локальная** (*local*) – (логически) эксклюзивна для CUDA-ядра.
- **Кэш для глобальной памяти** начиная с Fermi.

Архитектура GPU AMD

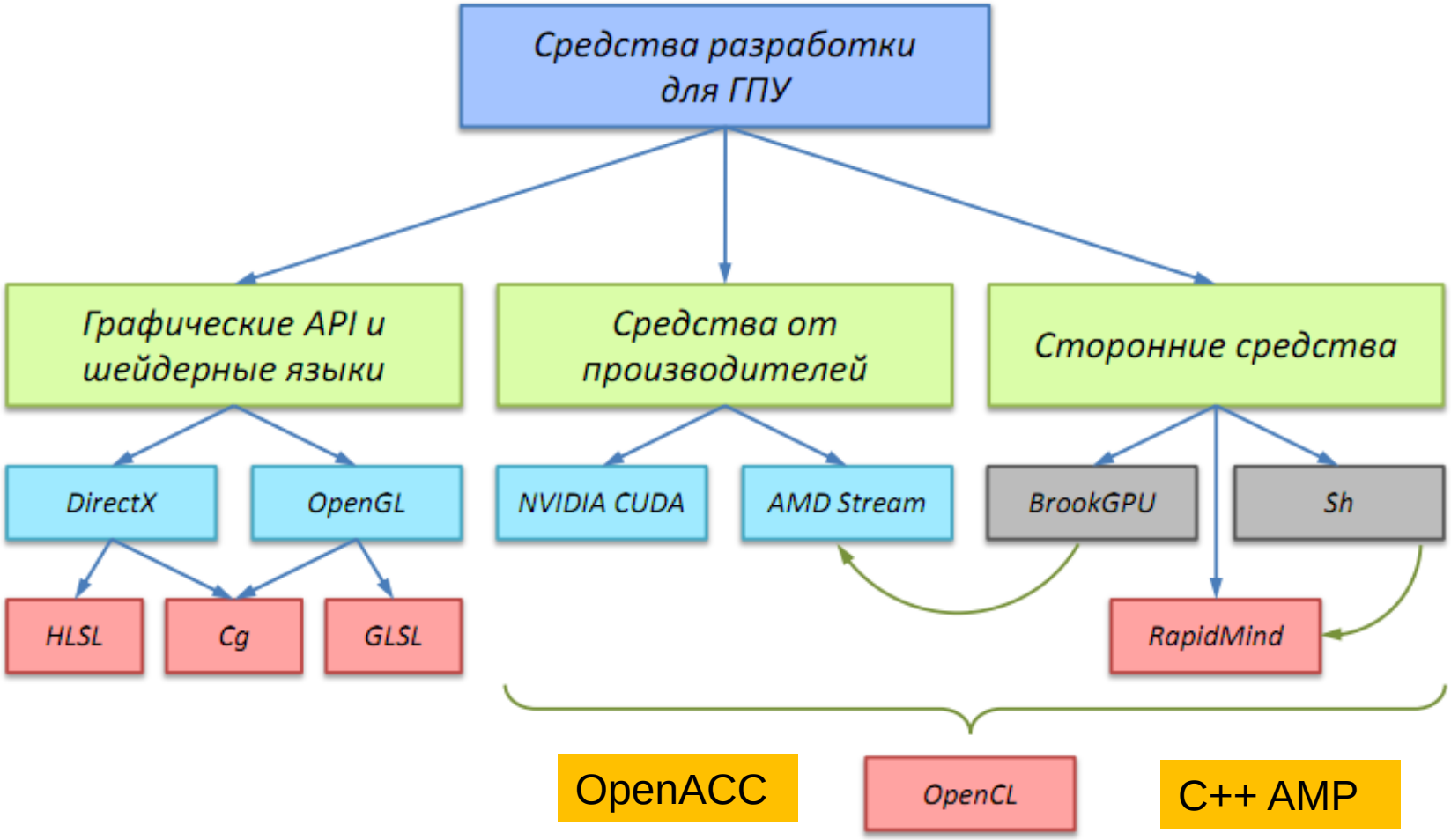
- Во многих чертах схожа с архитектурой NVIDIA с точностью до количественных характеристик и замены терминов.
- Устройство состоит из **SIMD-процессоров** (*SIMD engine*), каждый из которых состоит из нескольких **потоковых ядер** (*stream cores*).
- Потоковое ядро является **векторным** и состоит из 5 АЛУ, называемых **обрабатывающими элементами** (*processing elements*).
  - Это одно из отличий: у NVIDIA скалярные CUDA-ядра.

Обзор средств программирования для GPU

Потоковая модель вычислений

- Работа GPU хорошо укладывается в **потоковую модель вычислений** (*stream computing model*).
- **Поток** (в смысле данных) – последовательность однотипных элементов, обработка каждого элемента может осуществляться независимо от других.
- Обработка потока выполняется при помощи **ядер** (*kernel*).
- Упрощенно: тело ядра содержит вычисления над одним элементом, оно вызывается многократно (и, возможно, параллельно) для каждого элемента.
  - Пример: задача сложения векторов длины N, в теле ядра складывается пара значений, ядро вызывается N раз.
- Средства программирования для GPU используют потоковую модель.

Обзор средств



Источник: Д.К. Боголепов, В.Е. Турлапов «Вычисления общего назначения на графических процессорах»

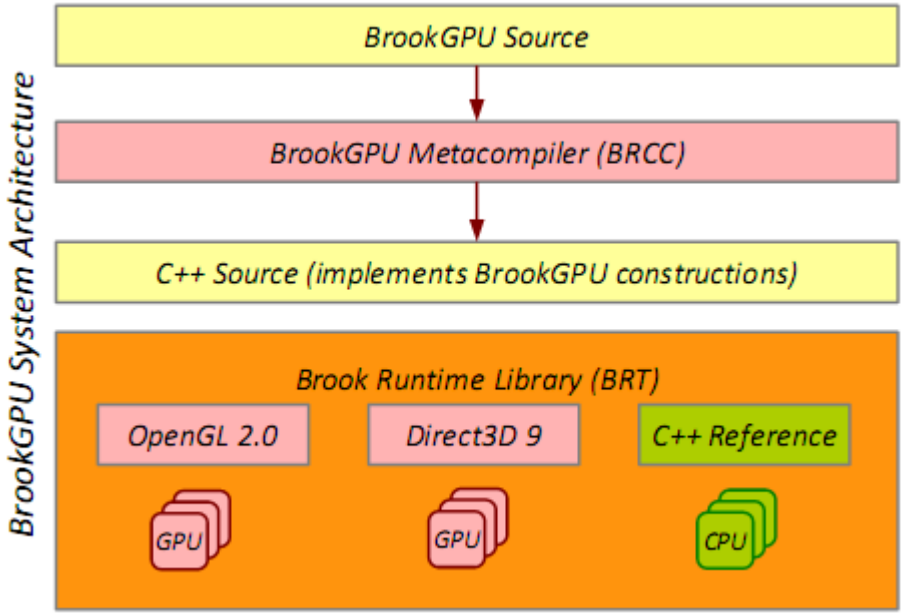
Шейдерные языки

- Изначально применялись для решения задач визуализации.
- Одним из наиболее популярных является GLSL (OpenGL Shading Language, разрабатывается Khronos Group), входящий в стандарт OpenGL с версии 2.0.
- Недостатки:
  - На шейдерных языках пишутся только шейдеры (единицы параллельного исполнения), а не целые программы.
  - Необходимо формулировать задачу в графических терминах («привязка к графике»).

Источник: Д.К. Боголепов, В.Е. Турлапов «Вычисления общего назначения на графических процессорах»

Сторонние специализированные средства

- Большинство основано на **метапрограммировании**: исходная программа разрабатывается на диалекте языка C с потоковыми расширениями, после чего транслируется в шейдеры для исполнения на GPU.
- Наиболее известными являются BrookGPU и Sh.



Источник: Д.К. Боголепов, В.Е. Турлапов «Вычисления общего назначения на графических процессорах»

Сторонние средства: BrookGPU и Sh

- Упростили разработку по сравнению с шейдерными языками.
- Проблемы с эффективностью, не выполняется оптимизация под конкретную архитектуру.
- В настоящее время устарели и не поддерживаются, однако послужили основной для дальнейших разработок:
  - Sh продолжила развитие в коммерческой платформе RapidMind, в 2009 поглощена Intel, технологии использованы в Intel Ct и Intel Array Building Blocks.
  - Переработанный вариант BrookGPU включался в AMD Stream до версии 1.4 под названием Brook+.

Источник: Д.К. Боголепов, В.Е. Турлапов «Вычисления общего назначения на графических процессорах»

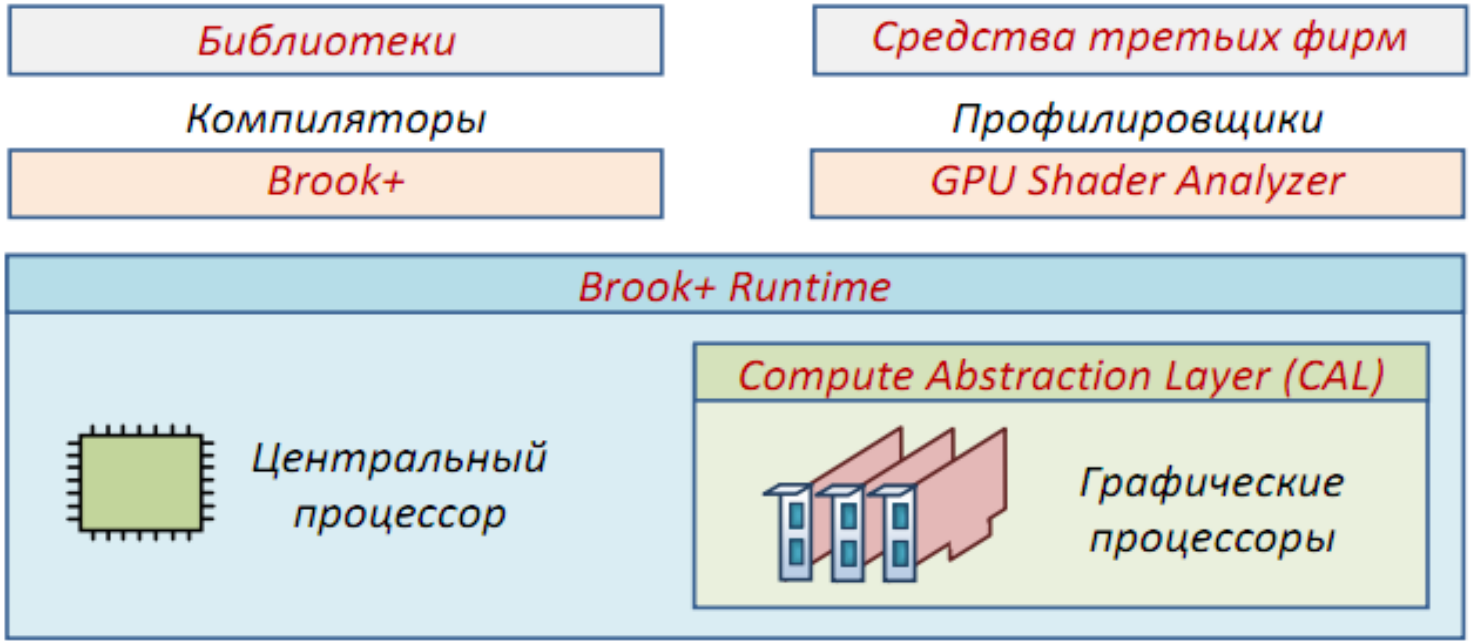
Средства от производителей

- CUDA от NVIDIA (февраль 2007 года).
- AMD Stream Computing от AMD/ATI (ноябрь 2007 года).
- Имеют схожую структуру:
  - Нижний уровень – ассемблер для графического процессора, менеджер памяти.
  - Верхний уровень – потоковые расширения языка C, средства оптимизации и отладки, высокопроизводительные библиотеки.

Средства от производителей

- Преимущества:
  - Написание программ на потоковых расширениях традиционных языков программирования без специфической терминологии компьютерной графики.
  - Доступ к специфическим функциям оборудования, широкие возможности для оптимизации.
- Недостатки:
  - Неуниверсальность, программирование требует глубоких знаний архитектуры.
  - Эффективное портирование затруднено из-за привязки к конкретному оборудованию.

AMD Stream Computing



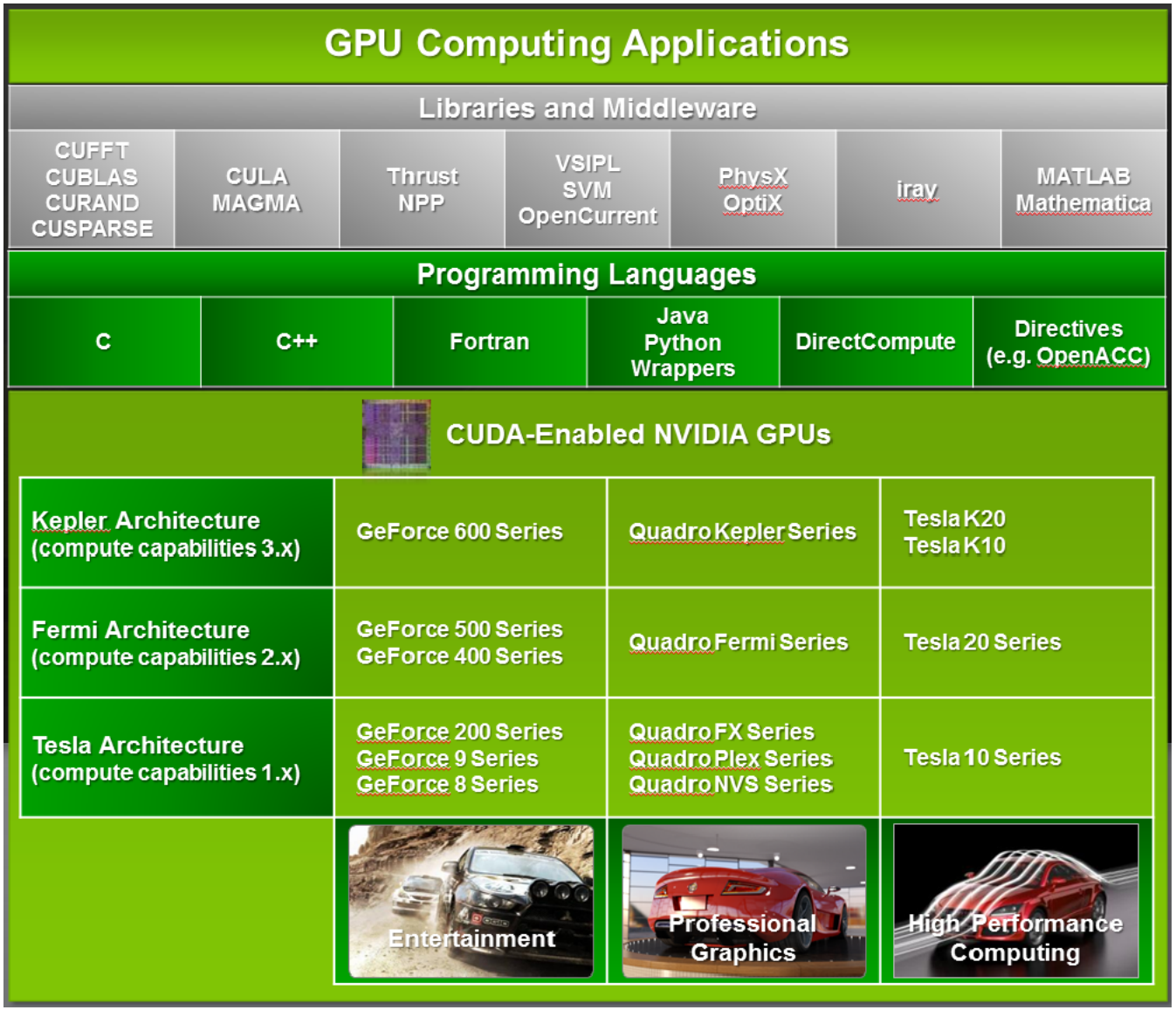
- Использует модифицированный Brook+ (потоковые расширения языка C на верхнем уровне).
- В настоящее время поддерживается интерфейс OpenCL.

Источник: Д.К. Боголепов, В.Е. Турлапов «Вычисления общего назначения на графических процессорах»

NVIDIA CUDA

- **CUDA – Compute Unified Device Architecture**.
- Программно-аппаратная платформа для параллельных вычислений от NVIDIA.
- Раскрывает потенциал GPU для вычислений общего назначения.
- Не поддерживается другими производителями.
- Наиболее широко используемое на текущий момент средство для GPGPU (по данным NVIDIA).

NVIDIA CUDA



Источник: NVIDIA CUDA C Programming Guide v. 6.5

Стандарт OpenCL

- **OpenCL – Open Computing Language**, открытый стандарт для гетерогенных вычислений, разрабатываемый Khronos Group совместно с представителями производителей устройств и ПО.
- Первая версия стандарта – ноябрь 2008 года.
- Поддерживается Apple, NVIDIA, AMD, Intel, …
- Поддержка широкого класса вычислительных устройств за счет введения обобщенных моделей (модели платформы, памяти, исполнения, …).

Основные особенности стандарта

- Исходный код приложения легко портируется на другие платформы.
- Поддержка широкого класса устройств достигается за счет введения **обобщенных моделей** данных систем:
  - **модель платформы** (*platform model*);
  - **модель памяти** (*memory model*);
  - **модель исполнения** (*execution model*);
  - **модель программирования** (*programming model*).
- Все модели являются абстрактными (не привязанными к конкретным устройствам), реализация предоставляется производителем.

OpenACC

- Стандарт для гетерогенных вычислений
- Распараллеливание за счет директив, во многом напоминает OpenMP (и #pragma offload для Xeon Phi)
- Ограничен доступ к низкоуровневой оптимизации
- Хорошо подходит для быстрого портирования больших приложений
- Требовательные к производительности участки могут быть затем переписаны на CUDA и оптимизированы дополнительно

C++ AMP

- C++ Advanced Massive Parallelism – язык для написания гетерогенных приложений от Microsoft.
- Официального релиза еще нет, доступен в Microsoft Visual Studio 11 Developer Preview.
- Будет встроен в Visual Studio 11.
- Построен на DirectCompute.

Выводы

- CUDA, OpenCL и C++ AMP опираются, в основных чертах, на одну и ту же модель вычислений. Ядра удивительно похожи.
- Существенная разница лишь во вспомогательном коде: инициализация данных устройства, копирование данных, запуск ядер, синхронизация и т.д.
- Простейший пример переноса приложения на GPU: преобразование тела цикла с независимыми итерациями в ядро.
  - В реальной жизни редко бывает так просто.

Рекомендации

- Выбор подходящих для параллельных вычислений алгоритмов. Типичные схемы декомпозиции задачи:
  - разбиение на большое число (тысячи и более) независимых подзадач;
  - разбиение на относительно небольшое число независимых подзадач, каждая из которых делится на относительно большое число подзадач.
- Предпочтительны алгоритмы с большой долей вычислений по отношению к количеству обращений к памяти.
- Использование CPU для последовательных вычислений и GPU для параллельных. Минимизация обмена между хостом и устройством и/или выполнение его асинхронно.

Одновременное использование CPU и GPU для гетерогенных вычислений

Мотивация

- Гетерогенные системы получают все более широкое распространение:
  - Появление **APU** (*accelerated processing unit*), сочетающих ядра центральных и графических процессоров.
  - Сопроцессоры **Intel Xeon Phi** (*many integrated cores*), содержащих десятки x86-ядер.
- В настоящее время наиболее популярной гетерогенной связкой является **CPU + GPU**.

Мотивация

- Пиковая производительность современных GPU существенно превосходит пиковую производительность CPU.
- Однако, на реальных задачах разрыв может быть не столь существенным.
- Во всяком случае, не столь существенным, чтобы полностью отказаться от использования ядер CPU для вычислений.
  - Особенно в свете динамично растущего количества ядер CPU и увеличивающегося разнообразия гетерогенных систем.

Мотивация

- При создании приложений с использованием CUDA и OpenCL (если GPU является устройством) обычно часть программы, исполняющаяся на хосте, выполняет лишь *служебные* функции:
  - Инициализация.
  - Запуск ядер.
  - Обмен данными.
- Выполнение этих операций обычно не несет большой вычислительной нагрузки не требует задействования всех ядер CPU.
- Следовательно, некоторые (возможно, почти все) из ядер CPU могут быть *эффективно использованы для вычислений*.

Вариант организации работы

- Создание отдельного потока для выполнения служебных операций для каждого устройства.
  - Или одного «служебного» потока для всех устройств.
- Для вычислений на CPU можно использовать количество потоков, равное разности количества ядер CPU и количества «служебных» потоков.
  - Технологии параллельного программирования для многоядерных CPU (OpenMP, TBB, Cilk Plus и др.) позволяют устанавливать число используемых потоков.
- В этом случае «служебные» потоки не будут конкурировать с «вычислительными» за процессорное время.

Балансировка нагрузки

- Естественная проблема, вызванная разнородностью используемых устройств.
- Во-первых, *существенное различие в производительности*.
- Во-вторых, *разный способ обмена данными*:
  - Ядра CPU используются общую память.
  - Необходимо явное копирование данных между CPU и GPU.
  - Обмен между двумя GPU осуществляется через CPU.
- Накладные расходы, вызванные неэффективной балансировкой и/или организацией обменов могут существенно снизить выгоду от использования CPU или вообще привести к снижению производительности по сравнению с использованием только GPU.

Возможные варианты реализации

- **Вариант 1.** Создание *отдельных версий вычислительной части кода для CPU и GPU* с использованием специализированных средств (например, OpenMP/TBB + CUDA).
- Дублирование вычислительной части кода, сложность модификации и поддержки.
- Возможно, высокая эффективность (при подходящей задаче, должной оптимизации и балансировке нагрузки).
- Возможно, проблемы с переносимостью (например, CUDA работает только на GPU NVIDIA).

Возможные варианты реализации

- **Вариант 2.** Создание *унифицированной версии вычислительного кода для CPU и GPU* с использованием OpenCL.
- Техники оптимизации приложений для CPU и GPU существенно различны.
- Вероятно, такой подход не обеспечит максимальной эффективности по сравнению с первым.
- В ряде задач за счет параметризации можно достичь *достаточно высокой эффективности для обеих платформ*.
  - Пример: параметризация размера обрабатываемой подматрицы в операциях линейной алгебры: большие подматрицы для полной загрузки GPU, маленькие кэш-дружественные подматрицы для ядер CPU.

Возможные варианты реализации

- Вопрос об эффективности приложений с использованием OpenCL относительно приложений с использованием специализированных технологий в настоящее время не имеет однозначного ответа.
- По меньшей мере в ряде задач производительность OpenCL сравнима со специализированными средствами.
- Широкие возможности для переносимости.
- Относительно легкий способ утилизации всех доступных вычислительных ресурсов.
  - А также модификации существующих приложений на OpenCL, использующих один тип вычислительных устройств.

Выводы

- Проблема эффективного использования всех вычислительных ресурсов в гетерогенных системах является актуальной и сложной.
- Возможны разные технические решения, в любом случае требуется эффективная балансировка нагрузки и организация обменов данными.
- Использование OpenCL открывает относительно легкий путь к утилизации всех вычислительных ресурсов. Вопрос в эффективности такого решения.
- Примеров успешного решения данной задачи в настоящее время очень мало (например, MAGMA http://icl.cs.utk.edu/magma/).

Материалы

- NVIDIA CUDA C Programming Guide
- Материалы курса по CUDA и OpenACC: http://www.nvidia.ru/object/cuda-openacc-online-course-ru.html
- Боресков А.В. и др. «Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие»
- Сандерс Дж., Кэндрот Э. «Технология CUDA в примерах: введение в программирование графических процессоров»
- Farber R. CUDA Application Design and Development