**КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2010 Т. 2 № 3 С. 295–308**

Ки&M

МАТЕМАТИЧЕСКИЕ ОСНОВЫ И ЧИСЛЕННЫЕ МЕТОДЫ МОДЕЛИРОВАНИЯ

УДК: 004.42

Основы технологии CUDA

А. М. Казённов

Московский физико-технический институт (государственный университет),
Россия, 141700, Долгопрудный, пер. Институтский, 9

E-mail: kazennov@gmail.com

*Получено 5 сентября 2010 г.,
после доработки 20 сентября 2010 г.*

Рассказывается об истории развития технологии CUDA, о принципиальных её ограничениях. Статья предназначена для читателей, не знакомых с особенностями программирования графических процессоров, но желающих оценить возможности их использования для решения прикладных задач.

Ключевые слова: CUDA, GPU, GPGPU, видеокарта, графический адаптер

Basic concepts of CUDA technology

A. M. Kazennov

Moscow Institute of Physics and Technology, 9 Institutskii per., Dolgoprudny, 141700, Russia

Abstract. — The history of the development of CUDA technology and its fundamental limitations are discribed. The article is intended for those readers who are not familiar with graphics adapter programming features but want to evaluate the possibilities for GPU computing applications.

Keywords: CUDA, GPU, GPGPU, video card, graphics adapter

Citation: *Computer Research and Modeling*, 2010, vol. 2, no. 3, pp. 295–308 (Russian).

Введение

Технология CUDA появилась в 2006 году и представляет из себя программно-аппаратный комплекс производства компании Nvidia, позволяющий эффективно писать программы под графические адаптеры. С 2006 года компания Nvidia обещает, что все графические адаптеры их производства независимо от серии будут иметь сходную архитуктуру, которая полностью поддерживает программную часть технологии CUDA. Программная часть, в свою очередь, содержит в себе всё необходимое для разработки программы: расширения языка C, компилятор, API для работы с графическими адаптерами и набор библиотек.

Чтобы понять, чем технология CUDA отличается от других существующих технологий распараллеливания, надо понять, чем стандартные многопроцессорные системы отличаются от графических адаптеров.

Рассмотрим схематическое устройство процессора Intel.

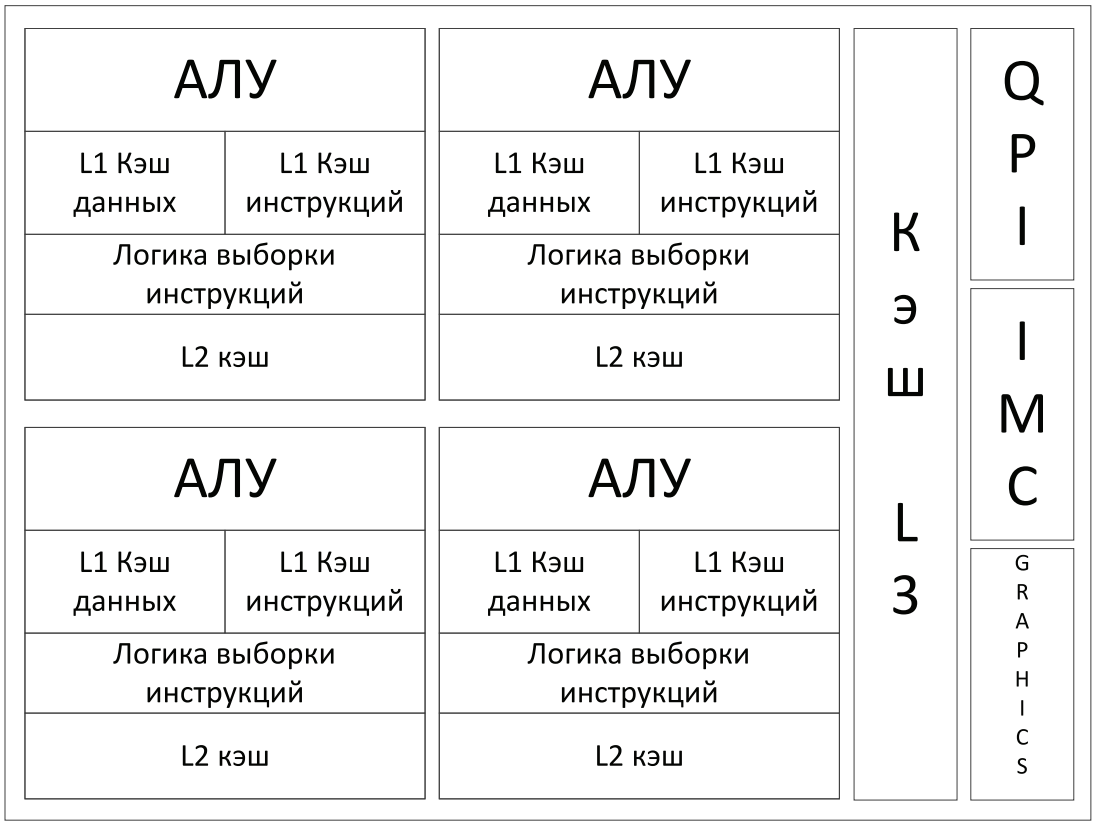

Рис. 1. Схематическое изображение архитектуры центрального процессора Intel Nehalem (CPU). Изображенная площадь устройств пропорциональна их площади на реальном кристалле

Как показано на рис. 1, бóльшую часть площади вычислительного кристалла занимает кэш-память, а вычислительные модули (АЛУ) занимают лишь четверть кристалла. Кроме того, каждый отдельный АЛУ является полноценным центральным процессором. Он способен поддерживать все аппаратные прерывания, может работать со всеми устройствами ввода/вывода, что, несомненно, полезно для его функционирования как центрального процессора, однако становится излишним для его использования как векторного вычислительного модуля.

Кроме того, потоки, выполняемые на центральном процессоре, являются очень «тяжеловесными», ими управляет операционная система, поэтому их не может быть много (максимальное количество измеряется несколькими тысячами).

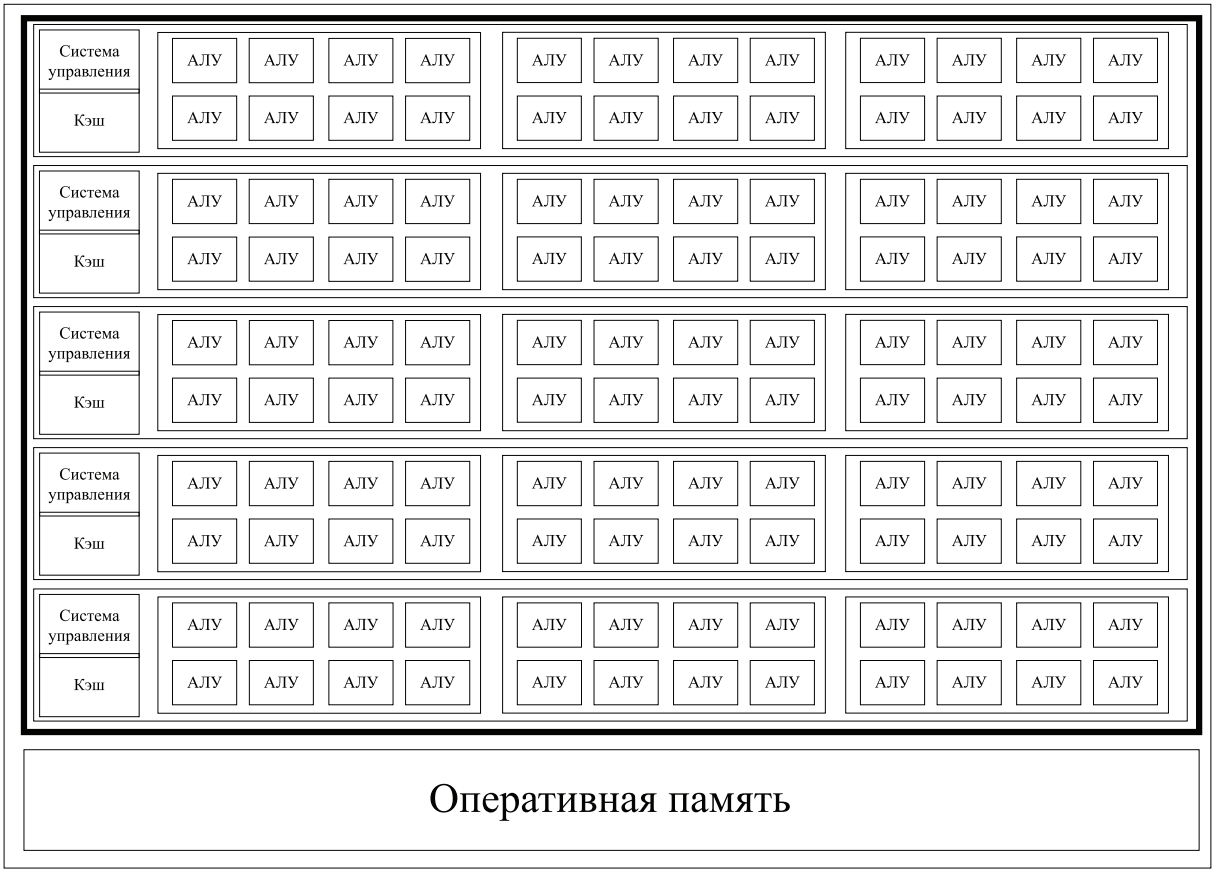

Рис. 2. Схематическое изображение графического адаптера (GPU)

Теперь сравним с графическим процессором (GPU). На рис. 2 показано, что большая часть площади вычислительного кристалла занята вычислительными устройствами, доля кэша и устройств управления очень мала. Процессоры на GPU куда проще, чем ядра CPU, они могут выполнять только математические операции и не способны к самостоятельной деятельности (являются сопроцессорами к центральному процессору).

Таблица 1. Классификация вычислительных систем по Флинну.

	Single Instruction (Одна инструкция)	Multiple Instruction (Много инструкций)
Single Data (Одиночные данные)	SISD	MISD
Multiple Data (Множественные данные)	SIMD	MIMD

Рассмотрим классификацию существующих систем по Флинну [Flynn, 1972] (Таблица 1). В соответствии с этой классификацией современные системы относятся к следующим классам:

- CPU (одноядерный) — SISD (одновременно выполняется только одна инструкция над одним набором операндов);
- CPU (многоядерный) — MIMD (Одновременно несколько ядер могут работать совершенно независимо, каждое как SISD);

- GPU (NVIDIA *ComputeCapability* версии < 2.0) — SIMD (одновременно на графическом адаптере может выполняться только один поток вычислений, который работает с большим набором данных);
- GPU (NVIDIA *ComputeCapability* версии ≥ 2.0) — MIMD (одновременно на графическом адаптере может выполняться несколько потоков вычислений, каждый из которых работает с большим набором данных).

Таким образом видно, что графические процессоры изначально предназначены для параллельного решения одной массивно-параллельной задачи.

Графический адаптер на «аппаратном» уровне

Рассмотрим устройство графических адаптеров, основанных на различных микросхемах компании NVIDIA, более подробно и сравним, чем они отличаются, а в чем они схожи. Общее устройство графического адаптера схематически представлено на рис. 3 (для старых G80, G200; новую архитектуру Fermi рассмотрим отдельно) [Боресков, Харламов, 2010].

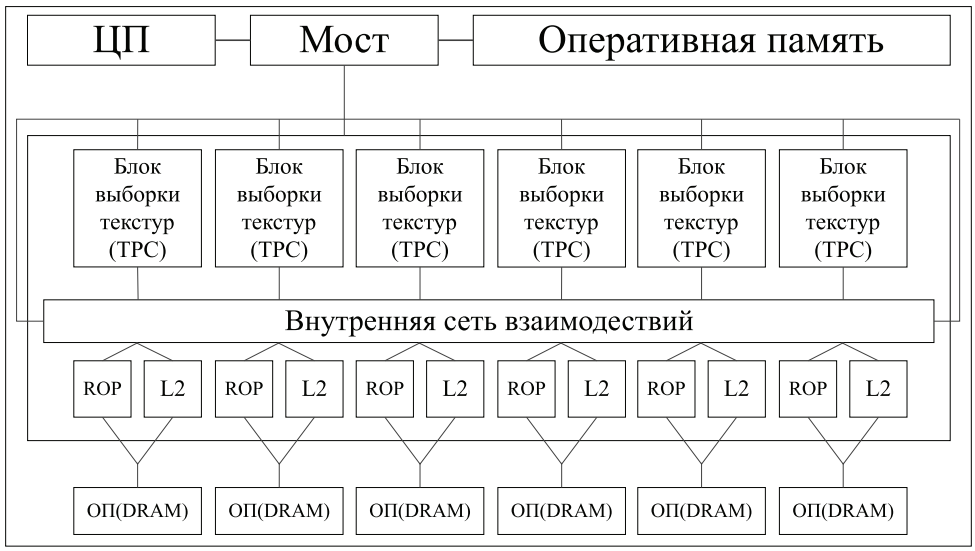

Рис. 3. Схематическое изображение устройства графического адаптера

Все вычислительные ядра на видеокарте объединены в независимые блоки TPC (Texture process cluster), количество которых определяется как принадлежностью к тому или иному семейству процессоров (G80 — максимум 8, G200 — максимум 10), так и конкретной моделью (GeForce 220GT — 2, GeForce 275 — 10). Как от видеокарты к видеокарте могут меняться количество TPC, так же может меняться тип и количество микросхем памяти DRAM и, соответственно, общий объем оперативной памяти. Микросхемы памяти имеют кэш второго уровня (L2) и конвейер растровых операций (ROP — Raster Operations Pipeline). Они объединены между собой коммуникационной сетью, в которую также подключены все TPC. Любая дискретная видеокарта взаимодействует с центральным процессором через мост, который может быть как интегрированным в CPU (Intel Core i7), так и вынесеным в чипсет (Intel Core 2 Duo).

Семейство процессоров NVIDIA G80

Первое семейство процессоров, поддерживающее возможности CUDA, — G80, его архитектура во многом определила основные ограничения и возможности всей линейки графических адаптеров.

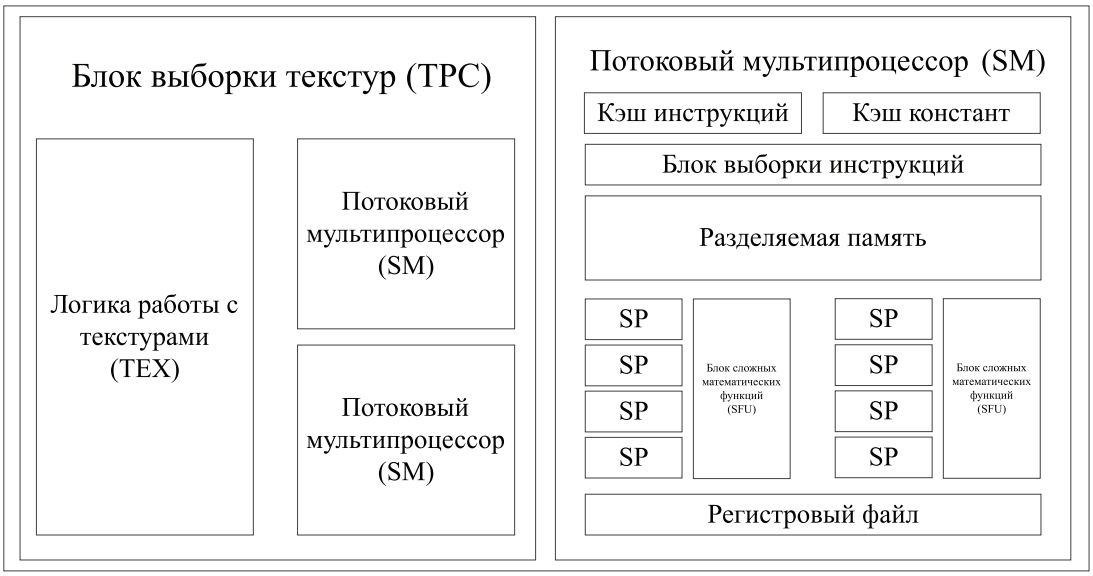

Рис. 4. Схематическое изображение устройства TPC и SM чипа G80

Семейство процессоров NVIDIA G200

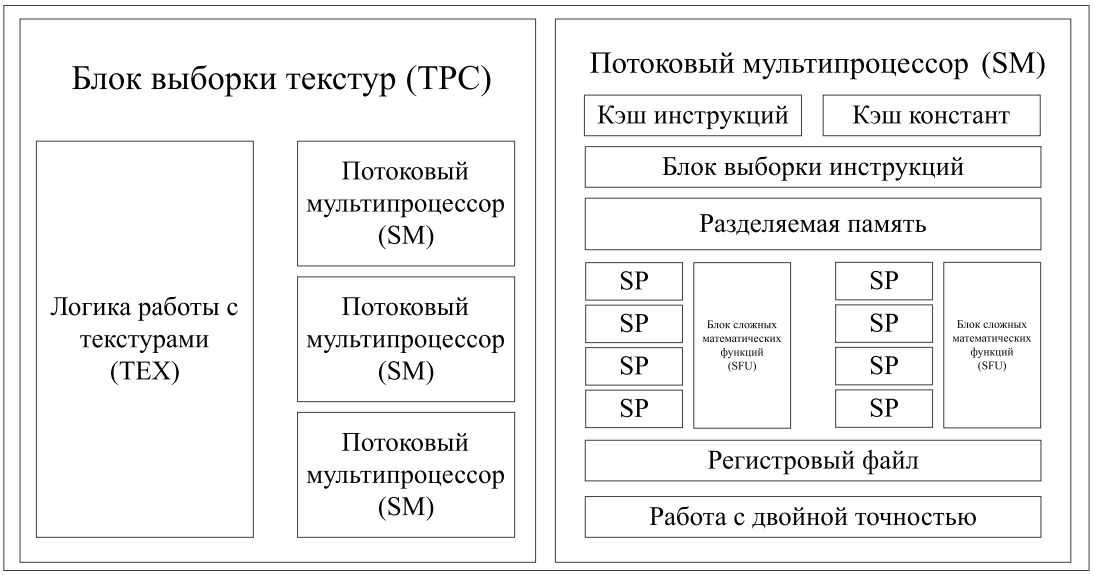

Рис. 5. Схематическое изображение устройства TPC и SM чипа G200

Как показано на рисунках 3–5, произошли как количественные изменения (увеличение количества потоковых мультипроцессоров SM в блоке выборки текстур TPC), так и качественные (в семействе G200 в потоковом мультипроцессоре SM появляется блок для расчета с двойной точностью). Одинаковыми же остаются:

- 8 потоковых процессоров SP (Streaming Processor) в мультипроцессоре SM (Streaming Multiprocessor);
- блок SFU (Super Function Unit) для расчета сложных математических функций (экспонента, корень и т. д.);
- 32 КБ регистрового файла на SM;

- 16 КБ разделяемой (shared) памяти на SM;
- кэш инструкций;
- кэш констант;
- блок выборки инструкций.

Семейство процессоров NVIDIA Fermi

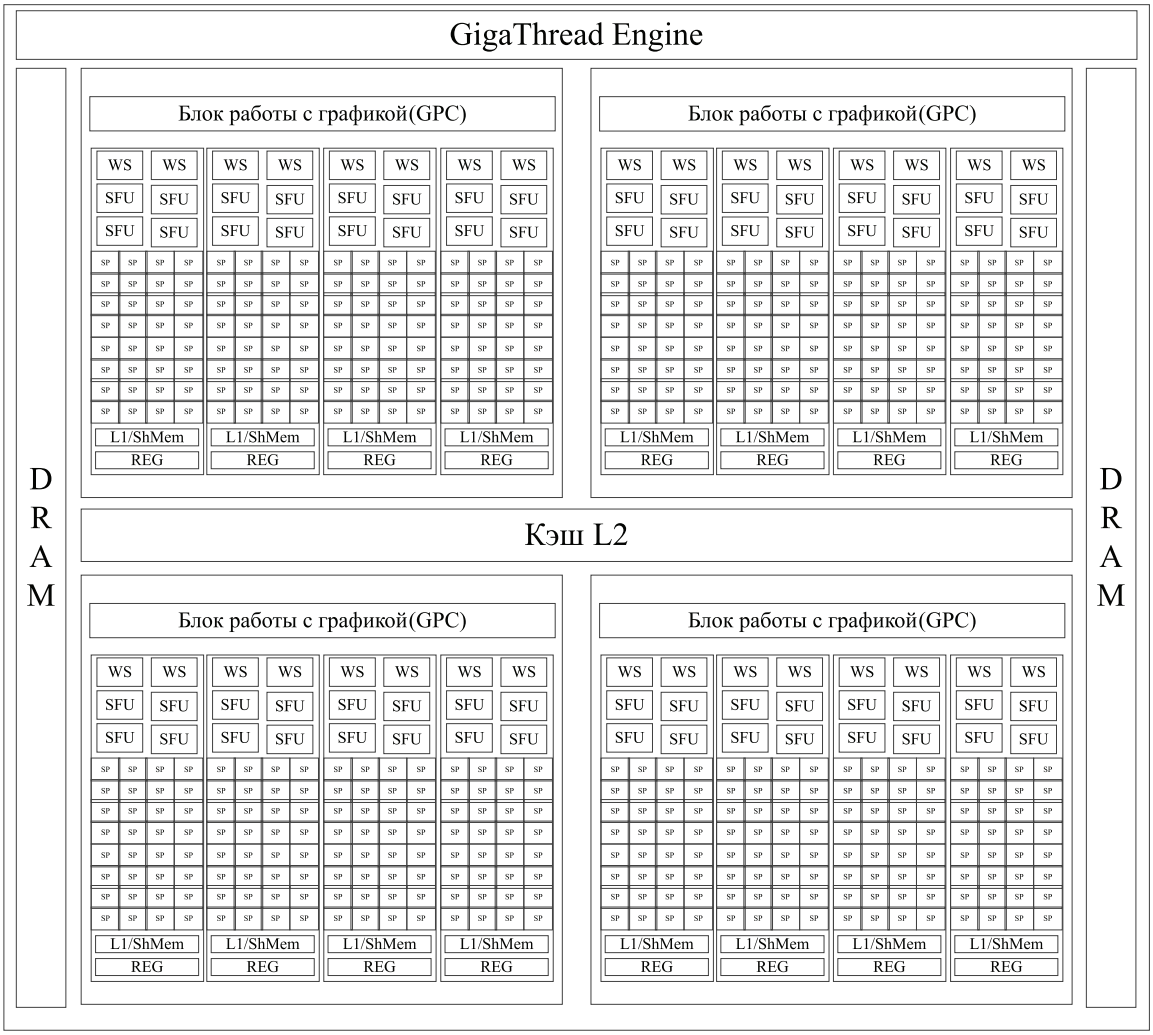

Рис. 6. Схематическое изображение устройства графического адаптера на чипе Fermi

Ключевыми архитектурными особенностями Fermi (рис. 6) являются:

- Третье поколение потокового мультипроцессора (SM):
  - 32 потоковых процессора на SM, что в четыре раза больше, чем у G200;
  - восьмикратный прирост производительности в операциях с плавающей точкой (FP) с двойной точностью по сравнению с графическими адаптерами предыдущего поколения;
  - два блока выборки инструкций (WS — Warp Scheduler) на SM вместо одного;
  - 64 КБ ОЗУ с конфигурируемым разделением на общую память и L1-кэш.
- Второе поколение набора инструкций параллельного выполнения потоков (Parallel Thread Execution, PTX 2.0):

  - унифицированное адресное пространство с полной поддержкой C++;
  - полная поддержка вычислений с плавающей точкой с 32- и 64-битной точностью в соответствии со стандартом IEEE 754-2008 [IEEE 754-2008; Яшкардин, 2009];
  - поддержка 64-битной адресации;
  - улучшенная производительность предсказаний обращений к памяти.
- Улучшенная подсистема памяти:
  - иерархия NVIDIA ParallelDataCache с конфигурируемым L1-кэшем и общим L2-кэшем;
  - поддержка памяти с кодом коррекции ошибок ECC (впервые на GPU);
  - существенно увеличенная производительность операций чтения и записи в память.
- Система управления вычислительными потоками NVIDIA GigaThread версии 3.0:
  - десятикратное по сравнению с G200 ускорение процедуры контекстного переключения;
  - одновременное выполнение нескольких потоков вычислений;

Таблица 2. Сравнение различных чипов графических адаптеров от NVidia

Архитектура	**G80**	**GT200**	**Fermi**
Год вывода на рынок	2006	2008	2009
Число транзисторов, млн.	681	1400	3000
Количество потоковых процессоров (CUDA-ядер)	128	240	512
Объем разделяемой памяти в расчете на SM, КБ	16	16	48 или 16 (конфигурируется)
Объем кэш-памяти первого уровня в расчете на SM, КБ	0	0	16 или 48 (конфигурируется)
Объем кэш-памяти второго уровня, КБ	0	0	768
Контроль ошибок ECC	Отсутствует	Отсутствует	Имеется

Возможности GPU обозначаются при помощи *ComputeCapability*, старшая цифра которой соответствует версии архитектуры, а младшая — небольшим изменениям внутри архитектуры. На данный момент существуют 1.0, 1.1, 1.2, 1.3, 2.0 *ComputeCapability*.

Таблица 3. *ComputeCapability* для различных графических адаптеров

GPU	***ComputeCapability***
GeForce 8800GTX	1.0
GeForce 9800GTX	1.1
GeForce 210	1.2
GeForce 275GTX	1.3
Tesla C2050	2.0

На этом разговор об аппаратном устройстве графических адаптеров можно закончить и приступить к обсуждению программной части технологии CUDA.

Программная часть технологии CUDA

Введем основные термины и отношения между ними [CUDA C Best Practices, 2010].

- Хост (Host) — центральный процессор, управляющий выполнением программы.
- Устройство (Device) — видеоадаптер, выступающий в роли сопроцессора центрального процессора.
- Грид (Grid) — объединение блоков, которые выполняются на одном устройстве.
- Блок (Block) — объединение тредов, которое выполняется целиком на одном SM. Имеет свой уникальный идентификатор внутри грида.
- Тред (Thread, поток) — единица выполнения программы. Имеет свой уникальный идентификатор внутри блока.
- Варп (Warp) — 32 последовательно идущих треда, выполняется физически одновременно.
- Ядро (Kernel) — параллельная часть алгоритма, выполняется на гриде.

На центральном процессоре (хосте) выполняются только последовательные части алгоритма программы, подготовка и копирование данных на устройство, задание параметров для ядра и его запуск. Параллельные части алгоритма оформляются в ядра, которые выполняются на большом количестве тредов на устройстве.

Для реализации программы под GPU компания NVIDIA выпустила свои расширения для языка C, компилятор NVCC для сборки таких программ, ввела в обиход новое расширение *.cu для файлов, которые содержат CUDA вызовы. К расширениям языка C относятся:

- спецификаторы для функций и переменных,
- новые встроенные типы данных,
- встроенные переменные (внутри ядра),
- директива для запуска ядра из C кода.

Спецификаторы для функций и переменных

- Спецификаторы функций

Таблица 4. Спецификаторы функций в CUDA

Спецификатор	Выполняется на	Может вызываться из
`__device__`	устройстве	устройства
`__global__`	устройстве	хоста
`__host__`	хосте	хоста

  - Спецификатор `__global__` применяется для функций, которые задают ядро (в них передается несколько специфических переменных). Функции `__global__` могут возвращать только `void`. Спецификатор `__global__` применяется только обособленно.
  - Спецификаторы `__host__` и `__device__` могут применяться одновременно для задания функций, выполняющихся и на хосте, и на устройстве (компилятор сам создает обе версии кода).

Для старых версий графических адаптеров (до Fermi) существуют дополнительные ограничения на выполняемые на видеокарте функции:

- нельзя брать адрес от функции (за исключением `__global__`);
- нет стека, а следовательно, нет рекурсии;
- внутри функций нельзя применять спецификатор `static` к переменным;
- не поддерживается переменное число аргументов в функциях.

- Спецификаторы переменных

Таблица 5. Спецификаторы переменных в CUDA

Спецификатор	Находится в	Доступна для	Вид доступа
`__device__`	устройстве	устройства	R
`__constant__`	устройстве	устройства и хоста	R / W
`__shared__`	устройстве	блока	RW / `__syncthreads()`

  - Спецификатор `__shared__` применяется для задания разделяемой памяти и не может быть инициализирован при объявлении. Он, как правило, требует явной синхронизации тредов внутри одного блока после любой операции с такой памятью.
  - Запись в `__constant__` память осуществляется только через специальные функции с CPU.
  - Никакие спецификаторы нельзя применять к полям структур или `union`.

Новые типы данных в CUDA

В CUDA добавлено несколько векторных типов для удобства копирования и доступа к данным.

Типы (u)`char`, (u)`int`, (u)`short`, (u)`long`, `float` могут быть 1-, 2-, 3-, 4-мерными векторами, а `longlong`, `double` — только 1- и 2-мерными. Для создания переменных таких типов требуется применять функции вида `make_`(тип)(размерность), например:

```
int2 a = make_int2 ( 1, 2 );
float4 b = make_float4 ( 1, 2, 3, 4 );
```

Для всех типов в CUDA Toolkit версии > 3.0 определены покомпонентные операции.

Также существует специальный тип `dim3`, основанный на типе `uint3`, имеющий нормальный конструктор. Этот конструктор позволяет задавать не все компоненты вектора (**недостающие компоненты инициализируются единицами**). Данный тип используется для задания параметров запуска ядра.

```
dim3 block = dim3 ( 64, 2 );
```

Встроенные переменные

В CUDA существует несколько особых переменных, которые существуют внутри каждого вычислительного ядра. Эти переменные позволяют отличать один тред от другого:

`dim3 gridDim` — содержит в себе информацию о конфигурации грида при запуске ядра,

`uint3 blockIdx` — координаты текущего блока внутри грида,

`dim3 blockDim` — размерность блока при запуске ядра,

`uint3 threadIdx` — координаты текущего треда внутри блока,

`int warpSize` — размер варпа (на данный момент всегда равен 32).

Директива запуска ядра

Для запуска ядра используются специальные директивы задания параметров запуска ядра и передачи ему аргументов.

Объявление функции для ядра с параметром `params`:

```
__global__ void Kernel_name(params);
```

Запуск ядра:

`Kernel_name<<<grid, block, mem, stream>>>( params )`, где

`dim3 grid` — размер грида для запуска. Грид может быть одно- и двухмерным, причем максимальное значение по каждому измерению **(65536, 65536, 1)**, а максимальное число блоков в гриде не может превышать **4294967296**;

`dim3 block` — размер блока при запуске. Блок может одно-, дву- и трехмерным, причем максимальное значение по каждому измерению **(512, 512, 64)**, а максимальное число тредов в блоке не может превышать 512;

`size_t mem` — количество разделяемой памяти на блок, которая выделяется для данного запуска под динамическое использование внутри ядра;

`cudaStream_t stream` — описание потока, в котором запускается данное ядро.

Программный интерфейс CUDA

Теперь, имея представление о запуске ядра и зная о расширениях языка C (вожделенных в CUDA), можно перейти к рассмотрению CUDA API. В CUDA есть два уровня API: низкоуровневый драйвер-API и высокоуровневый runtime-API. Runtime-API реализован через драйвер-API. Runtime-API обладает меньшей гибкостью, но более удобен для написания программ. Оба API не требуют явной инициализации, и для использования дополнительных типов и других расширений языка C не требуется подключать дополнительные заголовочные файлы. Все функции драйвер-API начинаются с приставки `cu`, все функции runtime-API начинаются с приставки `cuda`. Практически все функции из обоих API возвращают значение типа `t_cudaError`, которое принимает значение `cudaSuccess` в случае успеха.

Функции в API делятся на синхронные и асинхронные. Синхронные запуски являются блокирующими, асинхронные же позволяют выполнять другие операции во время их выполнения. Процессы копирования, запуска ядра, инициализация памяти могут быть асинхронными. При использовании асинхронных вызовов необходимо всегда помнить о синхронизации перед использованием их результатов.

Основные функции Runtime-API

`char* cudaGetErrorString(cudaError_t)` — функция возвращает описание ошибки по её числовому коду.

`cudaError_t cudaGetLastError()` — определение последней произошедшей ошибки, часто используется после вызова ядра для проверки правильности его работы.

`cudaError_t cudaThreadSynchronize()` — команда синхронизации, необходимая после любого асинхронного вызова.

`cudaError_t cudaGetDeviceCount(int*)` — функция, определяющая количество устройств, доступных для вычисления под CUDA.

`cudaError_t cudaGetDevicePropertis (cudaDeviceProp* props, int deviceNo)` — функция, определяющая параметры конкретного устройства по его номеру. В частности, функция позволяет определить *ComputeCapability* из полей `major` и `minor` структуры `cudaDeviceProp`.

Память в CUDA

В таблице 6 приведены сравнительные характеристики разных типов памяти, доступных на графическом адаптере.

Регистры — 32КБ на SM, используются для хранения локальных переменных. Расположены на кристалле GPU; скорость доступа самая быстрая. Выделяются отдельно для каждого треда.

Таблица 6. Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры (register)	R/W	на поток	Высокая (on-chip)
Локальная (local)	R/W	на поток	Низкая (DRAM)
Разделяемая (shared)	R/W	на блок	Высокая (on-chip)
Глобальная (global)	R/W	на грид	Низкая (DRAM)
Константная (constant)	R/O	на грид	Высокая (L1 cache)
Текстурная (texture)	R/O	на грид	Высокая (L1 cache)

Локальная — используется для хранения локальных переменных, когда регистров не хватает; скорость доступа низкая, так как расположена в микросхемах DRAM, находящихся вне кристалла. Выделяется отдельно для каждого треда.

Разделяемая — 16КБ (или 48КБ на Fermi) на SM, используется для хранения массивов данных, используемых совместно всеми тредамив блоке. Расположена на кристалле GPU; имеет чуть меньшую скорость доступа, чем регистры (около 10 тактов). Выделяется на блок.

Глобальная — основная память видеокарты (на данный момент максимально 6Гб на Tesla c2070). Используется для хранения больших массивов данных. Расположена в микросхемах DRAM и имеет медленную скорость доступа (около 80 тактов). Выделяется целиком на грид.

Константная — память, располагающаяся в микросхемах DRAM, снабжена специальным константным кэшем. Используется для передачи в ядро аргументов, превышаюших допустимые размеры для параметров ядра (для чипа Fermi — 256 байт). Выделяется целиком на грид.

Текстурная — память, располагающаяся в микросхемах DRAM, кэшируется. Используется для хранения больших массивов данных. Выделяется целиком на грид.

Обращение к памяти в CUDA осуществляется одновременно из половины варпа.

Все типы памяти, кроме регистровой и локальной, могут быть использованы в программе как эффективно, так и неэффективно. Для эффективного обращения к памяти необходимо четко понимать особенности и области применения разных её типов.

Использование глобальной памяти

Как уже говорилось выше, глобальная — это основная память в CUDA, однако она является при этом и самой медленной (она плохо работает с обращениями по случайному адресу, не имеет кэша, а следовательно, каждый раз данные надо считывать заново). Процесс работы с глобальной памятью очень похож на обычную работу с памятью на центральном процессоре:

- инициализация (выделение);
- заполнение (копирование);
- использование;
- обратное копирование;
- освобождение.

Все специальные функции работы с глобальной памятью вызываются на хосте. На устройстве работа происходит как с обычным массивом.

`cudaError_t cudaMalloc(void** devPtr, size_t size)` — выделение памяти на видеокарте. Здесь `devPtr` — указатель на участок памяти на устройстве, `size` — размер выделяемой памяти в байтах.

`cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)` — копирование данных между хостом и устройством. Здесь `dst` и `src` — указатели на приемник и источник соответственно (могут указывать на память как на хосте, так и на устройстве), `count` — размер копируемых данных в байтах, `kind` —

одно из четырех значений (`cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToHost`, `cudaMemcpyDeviceToDevice`), обозначающих направление копирования.

`cudaError_t cudaFree(void* devPtr)` — освобождение памяти на устройстве, где `devPtr` — указатель на память на устройстве.

Для повышения эффективности работы с глобальной памятью необходимо помнить, что обращение идет одновременно от 16 тредов полуварпа и что существуют особые паттерны доступа, позволяющие значительно ускорить работу с глобальной памятью. Графический процессор имеет возможность объединять запросы в блоки по 32/64/128 байт. Возможности объединения запросов различаются в зависимости от версии аппаратной архитектуры графического адаптера (см. табл. 7).

Таблица 7. Условия возникновения объединенного запроса к глобальной памяти в зависимости от версии аппаратной архитектуры графического адаптера

***ComputeCapability* 1.0, 1.1**	***ComputeCapability* 1.2, 1.3, 2.0**
Обращение идет к 32- или 64-битным словам	Обращение идет к 16-, 32- или 64-битным словам
Блок выровнен по своему размеру	Блок выровнен по своему размеру; К-я нить обращается к К-му элементу в блоке

Если условия объединения не выполнены, то графический процессор с *ComputeCapability* 1.0 или 1.1 выполнит 16 отдельных транзакций, а в случае *ComputeCapability* 1.2, 1.3, 2.0 запросы будут выполнены в несколько блоков, покрывающих весь размер запроса.

Использование разделяемой памяти

Разделяемая память находится на каждом SM и используется совместно всеми тредами внутри блока. Разделяемая память, как правило, требует использования команды явной синхронизации `__syncthreads()` после операций копирования и изменения. Для повышения пропускной способности разделяемая память разбита на 16 банков, доступ к которым идет независимо. Банки состоят из 32-битовых слов, причем подряд идущие слова расположены в идущих подряд блоках. Если возникает ситуация, что несколько тредов обращаются к одному и тому же банку (не элементу, а именно банку), то их обращения обрабатываются последовательно (происходит конфликт банков), однако если **все** треды обращаются к одному и тому же элементу, то конфликта банков не происходит.

Например,

```
__shared__ float a [N];
float x = a [base + threadIdx.x];
```

конфликта не происходит, так как размер `float` равен 32 битам и последовательно идущие элементы типа `float` попадают в последовательно идущие банки.

```
__shared__ char a [N];
char x = a [base + threadIdx.x];
```

происходят конфликты четвертого порядка, так как размер элемента типа `char` равен 8 битам и 4 последовательно идущих элемента типа `char` попадают в один и тот же банк.

Стандартная схема работы с разделяемой памятью выглядит следующим образом:

- загрузить необходимые данные в разделяемую память (из глобальной),
- `__syncthreads()`,
- выполнить вычисления с загруженными данными,
- `__syncthreads()`,
- записать результат в глобальную память.

Использование константной памяти

Константная память используется тогда, когда в ядро необходимо передать много различных данных, которые будут одинаково использоваться всеми тредами ядра.

`__constant__ float constData[256]` — объявление глобальной переменной с именем `constData` для использования в качестве константной памяти.

`cudaMemcpyToSymbol (constData, hostData, sizeof (data), 0, cudaMemcpyHostToDevice)` — копирование данных с центрального процессора в константную память.

Использование константной памяти внутри ядра ничем не отличается от использования обычной глобальной переменной на хосте.

Использование текстурной памяти

Текстурная память является особым образом выделенной областью глобальной памяти. Обращение к текстурной памяти производится с использованием кэша. Текстурная память также позволяет использовать адресацию с плавающей точкой (при этом применяется линейная или билинейная интерполяция). Соответственно, существуют дополнительные стадии конвейера (преобразование адресов, фильтрация, преобразование данных), которые снижают скорость первого обращения. Для использования текстурной памяти необходимо задать объявление текстуры как глобальной переменной, а потом связать её с требуемой областью глобальной памяти.

```
texture< float, 1, cudaReadModeElementType > g_TexRef;
```

Кроме самого объявления текстуры, требуется задать несколько параметров.

- **Нормализация адресов.** При нормализации адресов происходит перевод отрезка $[K, N]$ в отрезок $[0, 1]$.
- **Преобразование адресов.** Если координата не попадает в заданный диапазон (отрезок $[K, N]$ или $[0, 1]$), то видеокарта на аппаратном уровне производит преобразование. Существует два типа преобразования:
  - Clamp — возвращается значение на ближайшей границе диапазона;
  - Wrap — возвращается значение внутри диапазона, по сути, происходит взятие остатка от деления адреса на длину диапазона.
- **Фильтрация.** Когда обращение происходит по адресу типа `float`, а данные были заданы для целочисленных адресов, то необходимо определить, какое значение будет возвращено из текстуры. Существует два способа:
  - Point — берется ближайшее значение из массива;
  - Linear — расчет значения проводится на основе линейной (билинейной) интерполяции.
- **Преобразование данных.** Графический процессор имеет возможность преобразовывать считываемые данные, например, массив `char4` может быть преобразован в `float4`.

В CUDA существует два типа текстур — линейная и `cudaArray`.

После объявления текстуры и задания всех её параметров необходимо «привязать» данные, загруженные в глобальную память или `cudaArray`, к объявлению текстуры с помощью функций `cudaBindTexture` и `cudaBindTextureToArray` соответственно. Общая схема приведена на рис. 7.

Таблица 8. Особенности линейной и `cudaArray` текстурной памяти в CUDA

	Линейная	**`cudaArray`**
Размерность	1D	1D, 2D, 3D
Доступ	`tex1Dfetch()`	`tex1D()`, `tex2D()`, `tex3D()`
Преобразование адресов	Обращение вне диапазона адресов возвращает ноль	есть
Фильтрация	нет	есть
Преобразование данных	нет	есть

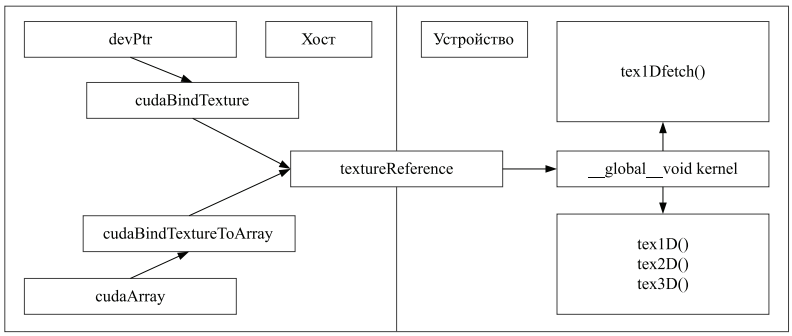

Рис. 7. Общая схема использования текстурной памяти

Заключение

Взаключение хочется сказать о том, в каких случаях использование технологии CUDA может дать хорошее ускорение по сравнению с последовательной реализацией на CPU, а в каких написание программы потребует больших усилий или будет вообще невозможно.

Вычисления с использованием графических адаптеров показывают максимальную эффективность в задачах, не требующих интенсивного обращения к памяти. Чуть хуже решаются задачи, для которых памяти требуется много, но при этом есть возможность использовать разделяемую память. Меньшее ускорение будет получена, когда требуется доступ по случайному адресу, но есть переиспользование данных (используется текстурная память). Совсем плохо будут решаться те задачи, для которых не выполняется ни одно из этих требований. Кроме того, если задача требует большого количества памяти (несколько гигабайт), то, скорее всего, на данном этапе развития технологии CUDA её вообще не целесообразно решать при помощи GPU.

Список литературы

CUDA C Best Practices Guide. — Version 3.2. — NVIDIA Corporation, 2010. — `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf`.

Flynn, M. Some Computer Organizations and Their Effectiveness. — IEEE Trans. Comput., 1972. — V.21. — N 9. — P. 948–960.

IEEE Standard for Floating-Point Arithmetic 754-2008. — IEEE, 2008.

Боресков А. В., Харламов А. В. Основы работы с технологией CUDA. — М.: ДМКПресс, 2010. — 232 с.

Яшкардин В. Л. IEEE 754 — стандарт двоичной арифметики с плавающей точкой. — www.softelectro.ru, 2009. — `http://www.softelectro.ru/ieee754.html`.