Казанский федеральный университет

ВЫСШАЯ ШКОЛА информационных технологий и информационных систем

Программная модель CUDA

Эдуард Храмченков

Nvidia CUDA

- CUDA – Compute Unified Device Architecture
- Программная модель CUDA включает вычислительный непосредственно в язык программирования
- CUDA – кроссплатформенная система компиляции и исполнения программ, части которых работают на CPU и GPU
- Актуальная версия API – CUDA Toolkit 8.0

Требования

- Совместимая с Nvidia CUDA видеокарта
- Драйвер Nvidia
- Установленный CUDA Toolkit
- Установленный компилятор C

Видеокарта и драйвера

- Nvidia CUDA поддерживается видеокартами Nvidia начиная с GeForce 8800
- Начиная с CUDA Toolkit версии 7.0 прекращена поддержка архитектуры Tesla – всех видеокарт до серии GeForce 400
- Если необходимо, можно установить старую версию CUDA Toolkit
- Каждая версия CUDA Toolkit имеет свой список актуальных драйверов

CUDA Toolkit

- http://developer.nvidia.com/object/gpucomputing.html
- Поддержка ОС: Windows, Linux, MacOs
- Дополнительно: примеры программ с использованием CUDA – GPU Computing SDK
- Проверка работоспособности:
  - $nvidia-smi
  - C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\bin\win64\Release\deviceQuery

Компиляторы и языки

- В настоящее время компилятор CUDA существуют для языков C и Fortran
- Компилятор для C бесплатный и поставляется в комплекте с CUDA Toolkit
- Компилятор для Fortran платный и поставляется компанией PGI
- PyCUDA – CUDA под Python
- jCUDA – решение для работы с CUDA из Java
- Существуют CUDA обертки для C#

Работа в Windows

- Для Windows систем необходима установленная VS версии 2005 или старше
- VS 2005 и 2008 не интегрируются с CUDA
- Начиная с CUDA Toolkit 8.0 полностью поддерживаются VS 2012 и старше, 2010 – частично, остальные IDE не поддерживаются
- Для создания нового CUDA-приложения необходимо в File-> New | Project… NVIDIA-> CUDA-> выбрать вашу версию CUDA Toolkit

Компиляция

- nvcc – компилятор для CUDA, исходные файлы *.cu
- Основные опции командной строки – в документации CUDA Toolkit
- Работа с компилятором похожа на работу с gcc
- Пример компиляции одного исходного файла:
  - *$nvcc -arch=sm_37 –O3 test.cu -o test*

Пример 0

- Hello world – опрос устройств совместимых с CUDA и вывод некоторых их характеристик
- *cudaGetDeviceCount(int &count)* – функция возвращающая количество устройств CUDA на машине
- *cudaDeviceProp* – структура хранящая основные характеристики устройства CUDA
- *cudaSetDevice(int &count*) – функция устанавливающая устройство №count как активное (нужна если несколько девайсов)

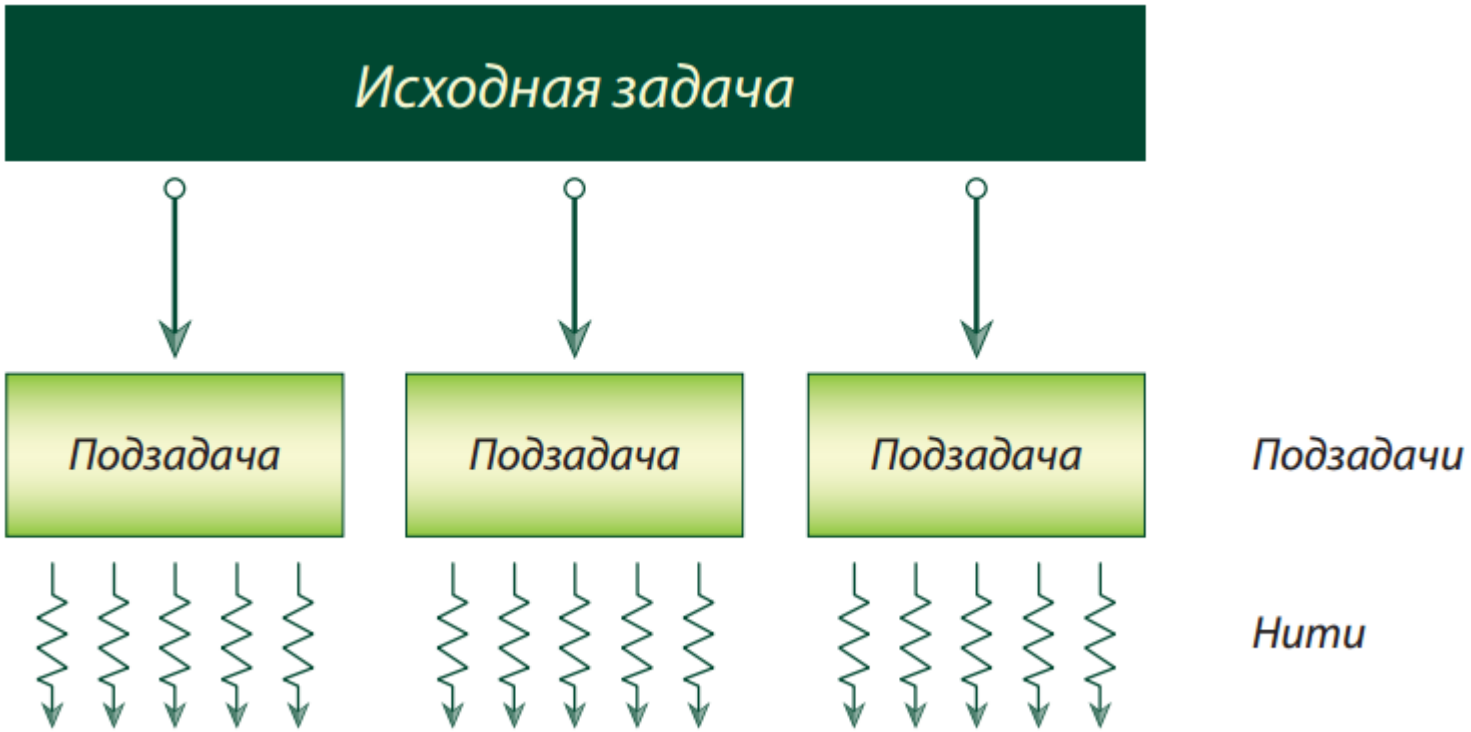
Основные принципы

- Программа на CUDA использует как CPU так и GPU
- Программа состоит из последовательных и параллельных участков кода
- На CPU, который называют host, выполняется последовательная часть кода, подготовка и вызов GPU кода
- На GPU, который называют device, выполняется параллельная часть кода

Основные принципы

- Код на GPU называется ядром (kernel)
- Ядро выполняется параллельно множеством нитей (threads)
- Каждая нить выполняет один и тот же код
- Разница между нитями CPU и GPU:
  - Нить GPU очень легкая, контекст минимален, выделение происходит быстро
  - Эффективное использование GPU – вызов тысячи нитей одновременно
  - Максимальная эффективность на CPU – количество нитей равно числу ядер, или кратно больше

Основные принципы



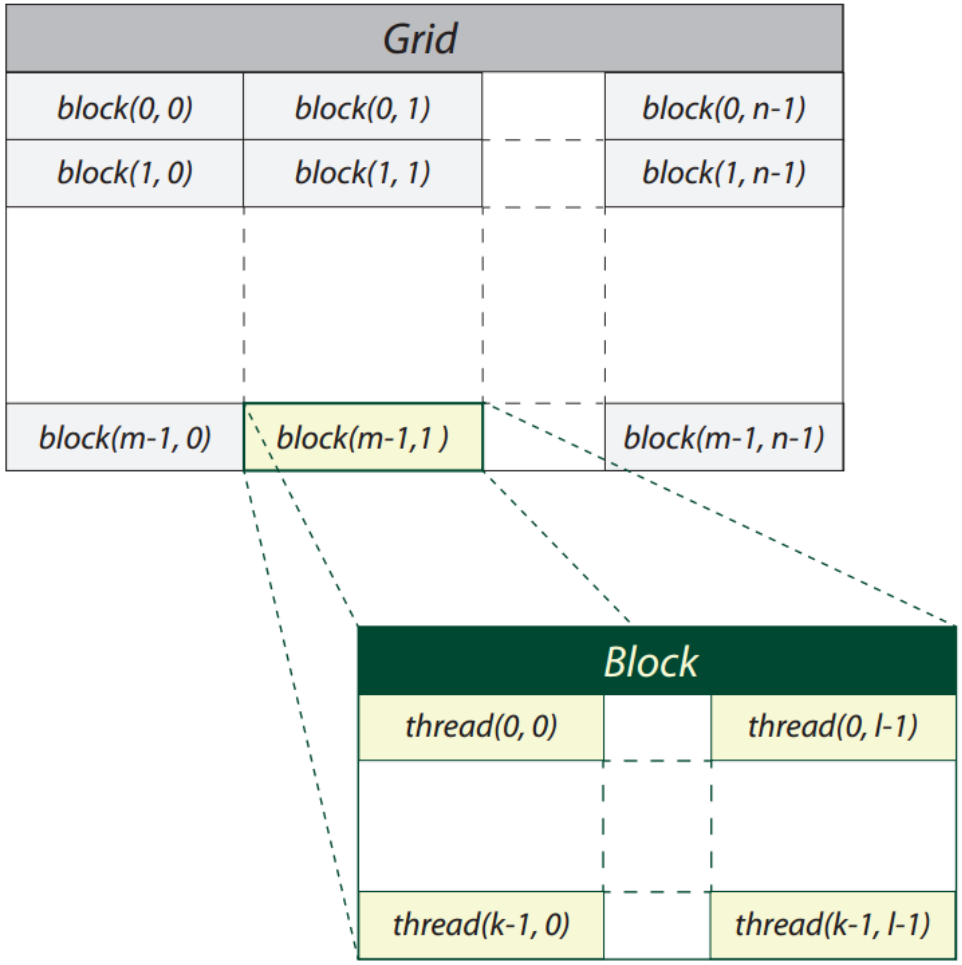
Основные принципы

- Работа нитей соответствует принципу SIMD
- Только нити в пределах одной группы (warp) выполняются физически одновременно
- В архитектуре Fermi и старше warp = 32 threads
- Управление работой варпов происходит на аппаратном уровне
- Каждая нить имеет свой идентификатор

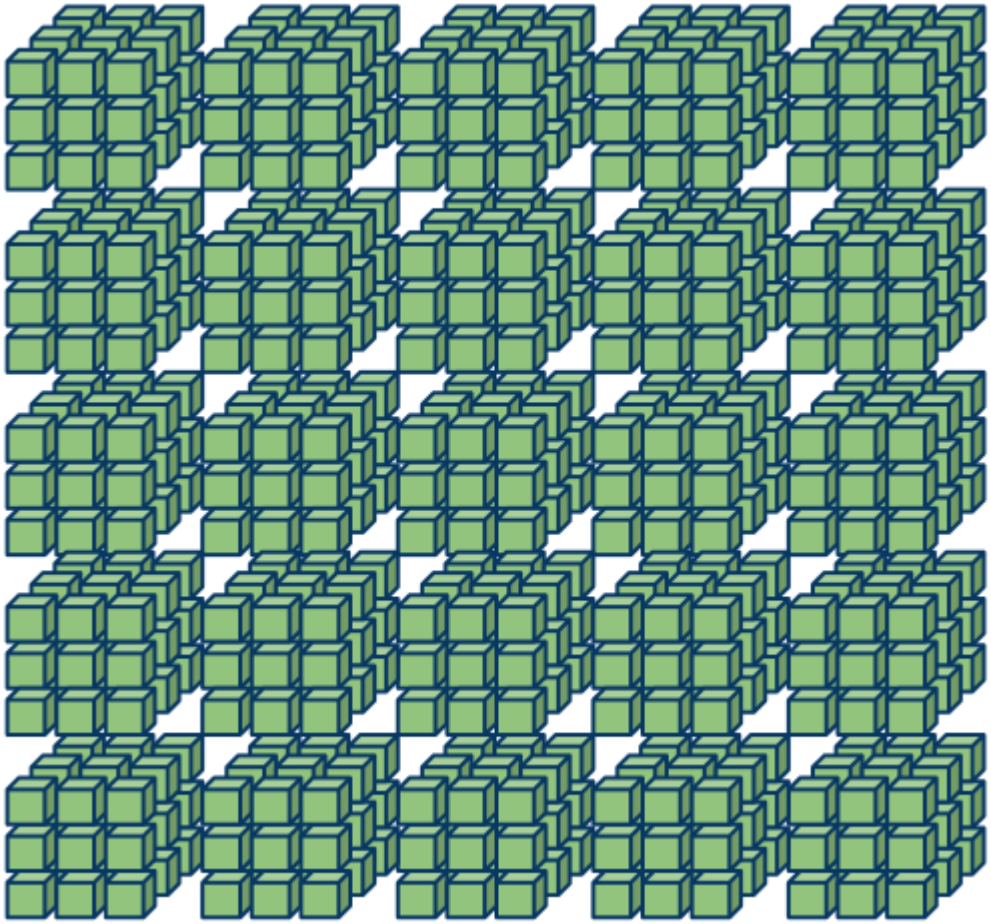
Сетки, блоки и нити

- Ядро CUDA запускает на выполнение сетку (grid) блоков нитей (thread blocks)
- Нити внутри блока взаимодействуют посредством разделяемой памяти
- Нити внутри блока могут синхронизироваться (не все потоки блока выполняются физически одновременно)
- Такая модель позволяет прозрачно масштабировать программы на различные GPU

Сетки, блоки и нити



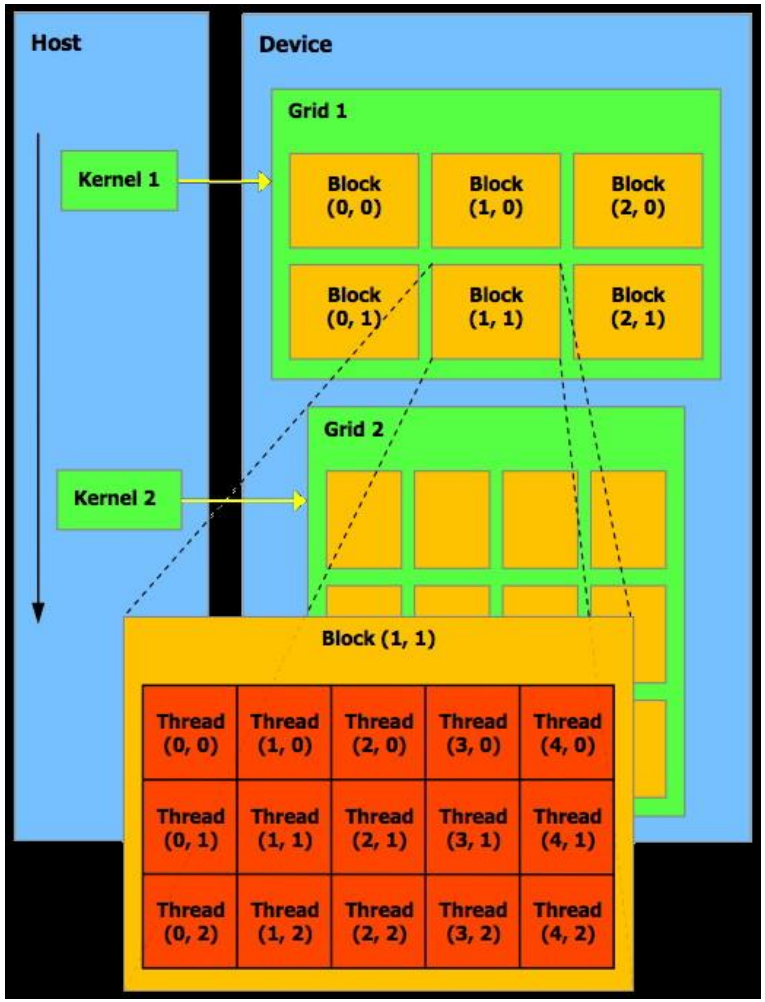
Сетки, блоки и нити



Сетки, блоки и нити

- Размеры блоков и сеток задаются программистом
- Нить GPU имеет координаты во вложенных трехмерных декартовых равномерных сетках «индексы блоков» и «индексы нитей внутри каждого блока»
- В контексте каждой нити значения координат и размерностей доступны через встроенные переменные *threadIdx*, *blockIdx* и *blockDim*, *gridDim*

Сетки, блоки и нити



Расширения языка C

- **Атрибуты функций** – показывают где будет выполняться и откуда вызывается функция
- **Атрибуты переменных** – задают тип памяти
- **Встроенные переменные** – содержат информацию относительно текущей нити
- **Дополнительные типы данных** – определяют несколько новых векторных типов
- **Оператор запуска ядра** – определяет иерархию нитей, очередь команд и размер разделяемой памяти

Атрибуты функций

Атрибут	Функция выполняется на	Функция вызывается из
__device__	device (GPU)	device (GPU)
__global__	device (GPU)	host (CPU)
__host__	host (CPU)	host (CPU)

Атрибуты функций

- Атрибут *__global__* обозначает ядро, и соответствующая функция CUDA C должна возвращать значение типа *void*
- На функции *__device__* и *__global__* накладываются ограничения:
  - не поддерживаются *static*-переменные внутри функции
  - не поддерживается переменное число входных аргументов

Атрибуты переменных

Атрибут	Размещение	Доступна	Вид доступа
__device__	device (GPU)	device (GPU)	R
__constant__	device (GPU)	device (GPU)/host (CPU)	R/W
__shared__	device (GPU)	block	RW

Атрибуты переменных

- Атрибуты не могут быть применены к полям структуры (*struct* или *union*)
- Переменные могут использоваться только в пределах одного файла, их нельзя объявлять как *extern*
- Запись в переменные типа *__constant__* может осуществляться только хостом при помощи специальных функций

Атрибуты переменных

- __*shared*__ переменные не могут инициализироваться при объявлении
- CUDA не поддерживает модульную сборку – каждая __*global*__ функция должна находиться в одном исходном файле вместе со всеми __*device*__ функциями и переменными, которые она использует

Встроенные типы

- 1/2/3/4-мерные векторные типы на основе *char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, longlong, float* и *double*
- Компоненты векторных типов имеют имена x, y, z и w
- Пример:
  - int2 a = make_int2 (1, 7); // Создает вектор (1, 7)
  - float3 u = make float3 (1, 2, 3.4f); // Создает вектор (1.0f, 2.0f, 3.4f)

Встроенные типы

- Для этих типов не поддерживаются векторные операции
- Добавлен тип *dim3*, используемый для задания размерностей блоков потоков и сеток блоков
- Этот тип основан на *uint3*, элементы по умолчанию инициализируются 1:
  - dim3 blocks **(** 16**,** 16 **);** // то же что blocks (16, 16, 1)
  - dim3 grid **(** 256 **);** // то же что grid (256, 1, 1)

Встроенные переменные

- *gridDim* – размер сетки (имеет тип *dim3*)
- *blockDim* – размер блока (имеет тип *dim3*)
- *blockIdx* – индекс текущего блока в сетке (имеет тип *uint3*)
- *threadIdx* – индекс текущего потока в блоке (имеет тип *uint3*)
- *warpSize* – размер warp'а (имеет тип *int*)

Оператор вызова ядра

- *kernel_name <<<Dg,Db,Ns,S>>> (args)*
- *kernel_name* – это имя или адрес соответствующей *__global__* функции
- Параметр *Dg* типа *dim3* задает размерности сетки блоков (число блоков в сетке блоков)
- Параметр *Db* типа *dim3* задает размерности блока нитей (число потоков в блоке)
- Суммарный размер параметров функции ядра должен быть ⩽ 4КБ

Оператор вызова ядра

- Необязательный параметр *Ns* типа *size_t* задает дополнительный объем разделяемой памяти в байтах (по умолчанию – 0), которая должна быть динамически выделена каждому блоку (в дополнение к статически выделенной)
- Параметр *S* типа *cudaStream_t* ставит вызов ядра в определенную очередь команд (CUDA Stream), по умолчанию – 0

Оператор вызова ядра

- Для CUDA реализованы математические функции, совместимые с ISO C
- Также имеются соответствующие аналоги, вычисляющие результат с пониженной точностью например, $\_\_sinf$ для sin
- Полный список функций – в документации CUDA Toolkit

Асинхронность в CUDA

- Многие функции CUDA являются асинхронными, управление в вызывающую функцию возвращается до завершения требуемой операции:
  - Запуск ядра
  - Функции копирования и инициализации памяти, имена которых оканчиваются Async
  - Функции копирования памяти device ↔ device внутри устройства и между устройствами

Асинхронность в CUDA

- С асинхронностью связаны объекты CUDA streams (потоки исполнения), позволяющие группировать последовательности операций, которые необходимо выполнять в строго определенном порядке
- При этом порядок выполнения операций между разными CUDA streams не является строго определенным и может изменяться

Обработка ошибок

- Каждая функция CUDA runtime API (кроме запуска ядра) возвращает $cudaError\_t$
- При успешном выполнении функции возвращается значение $cudaSuccess$, иначе возвращается код ошибки
- Текстовое описание ошибки и последняя ошибка:
  - char* cudaGetErrorString (cudaError_t code);
  - cudaError_t cudaGetLastError();

Пример 1

- Программа складывает 2 массива
- Код ядра начинается с определения глобального индекса массива, зависящего от координат нити
- Соответствие нитей и частей задачи может быть любым, например, одна нить может обрабатывать не один элемент массива, а определенный диапазон
- Глобальной памятью GPU можно управлять с хоста

Пример 1

- В функции gpu_sum память выделяемая на GPU заполняется копией данных из памяти хоста, затем производится запуск ядра sum_kernel, синхронизация и копирование результатов обратно в память хоста
- В конце производится высвобождение ранее выделенной глобальной памяти GPU
- Такая последовательность действий характерна для любого CUDA-приложения

Атомарные операции

- Атомарные операции предназначены для обеспечения корректного доступа к разделяемому ресурсу
- В случае CUDA разделяемый ресурс – это переменная, доступная множеству параллельных нитей
- При атомарном изменении запросы обрабатываются так, чтобы исключить одновременное чтение/запись

Атомарные операции

- Все атомарные операции, за исключением *atomicExch* и *atomicAdd*, работают только с целыми числами
- *atomicAdd* поддерживает числа типа *float* на устройствах с compute capability ⩾ 2.x и числа типа *double* начиная с compute capability 6.x
- *atomicExch* не поддерживает тип *double*

Атомарные операции

- *atomicAdd* – увеличивает значение переменной на заданное число (T – double, float, либо целочисленный тип)
  - T atomicAdd(T *adress, T val)
- *atomicExch* производит обмен значениями: новое значение записывается по указанному адресу, предыдущее возвращается как результат (T – float либо целочисленный тип)
  - T atomicExch(T *adress, T val)

Атомарные операции

- Прочие атомарные функции (T – целочисленный тип)
  - T atomicSub(T *adress, T val)
  - T atomicMin(T *adress, T val)
  - T atomicMax(T *adress, T val)
  - T atomicInc(T *adress, T val)
  - T atomicDec(T *adress, T val)
  - T atomicCAS(T *adress, T compare, T val)
  - T atomicAnd(T *adress, T val)
  - T atomicOr(T *adress, T val)
  - T atomicXor(T *adress, T val)

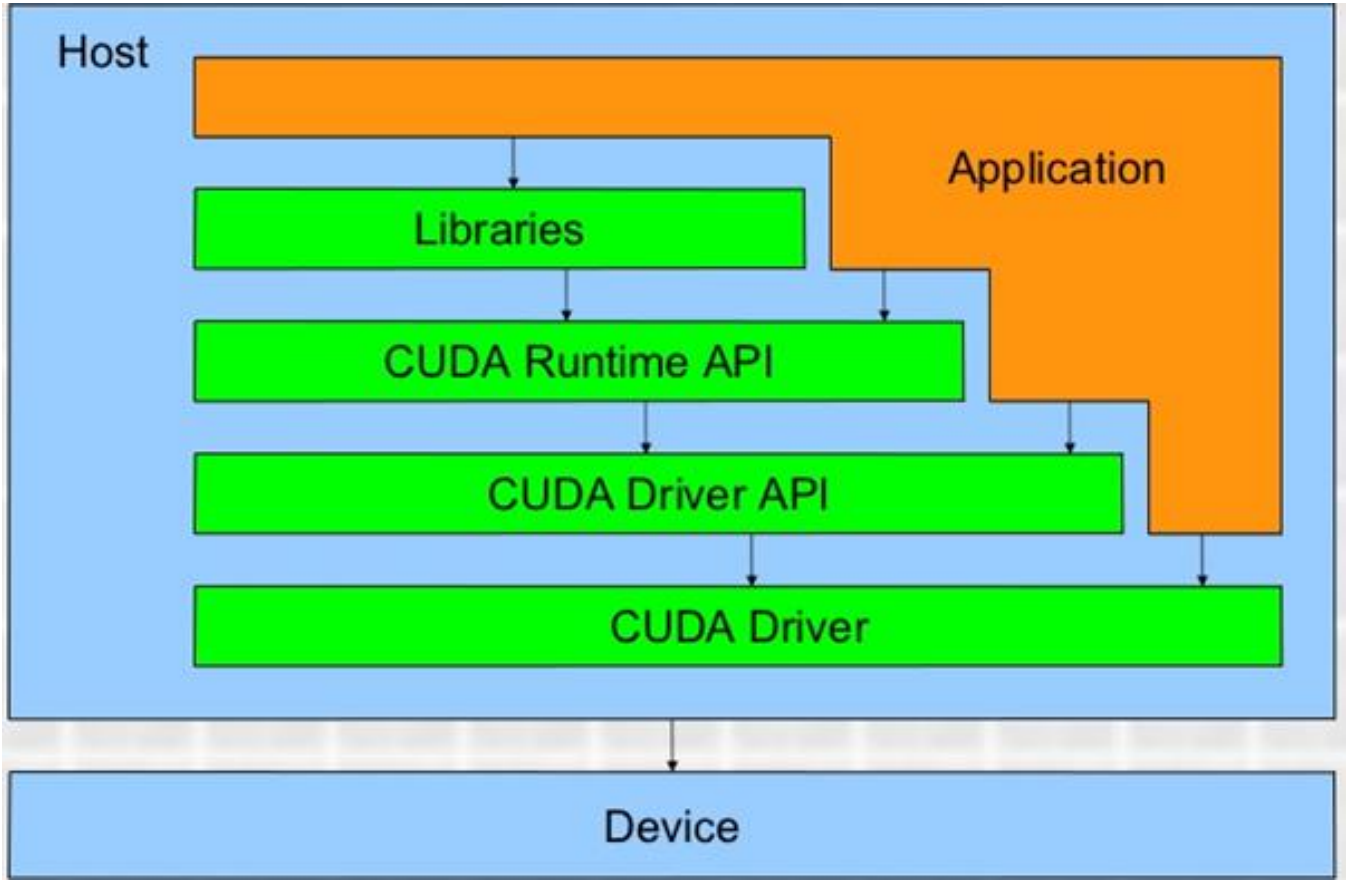
CUDA Runtime API

- CUDA Runtime API – библиотека функций, обеспечивающих:
  - управление GPU
  - работу с контекстом
  - работу с памятью
  - работу с модулями
  - управление выполнением кода
  - работу с текстурами
  - взаимодействие с OpenGL и Direct3D

CUDA Runtime API

- CUDA Runtime API делится на два уровня: driver API и CUDA API
- driver API является более низкоуровневым и требует явной инициализации устройства
- В CUDA API инициализация происходит неявно при первом вызове любой функции библиотеки
- Напрямую с устройством работает драйвер, на нем базируется CUDA Runtime API

Программный стек CUDA



Казанский федеральный
УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА
информационных технологий
и информационных систем

Вопросы

ekhramch@kpfu.ru