Нижегородский государственный университет им. Н.И. Лобачевского

Факультет Вычислительной математики и кибернетики

Введение в CUDA C

Горшков А.В., Бастраков С.И.

anton.v.gorshkov@gmail.com

CUDA C

- CUDA C – расширение языка C, включающее
  - квалификаторы функций;
  - квалификаторы типов памяти;
  - встроенные переменные.
- Содержит элементы C++.
- Терминология:
  - **хост** (*host*) = CPU;
  - **устройство** (*device*) = GPU;
  - **ядро** (*kernel*) – подпрограмма, параллельно выполняемая потоками на GPU.

Квалификаторы функций

Квалификатор	*Выполняется на:*	*Вызывается с:*
__host__	*host*	*host*
__global__	*device*	*host*
__device__	*device*	*device*

- **__host__** (по умолчанию) – функция, вызываемая с хоста и выполняемая на нем.
- **__global__** – функция, вызываемая с хоста и выполняемая потоками на устройстве (ядро).
- **__device__** – функция, вызываемая (одним потоком) и выполняемая на устройстве.

Пример синтаксиса

```
__host__ float hostSquare(float a) {
   return a * a;
}

__device__ float deviceSquare(float a) {
   return a * a;
}

__global__ void kernel(float a) {
   float a2 = deviceSquare(a);
}
```

__global__ функции

- Тип возвращаемого результата всегда void.
- Аргументы передаются через разделяемую/константную память, размер не больше 256 байт на compute capability 1.x, не больше 4kB на compute capability 2.x.
- Не может быть переменного числа аргументов.
- Не могут содержать статических переменных.
- Не могут содержать рекурсию **(для arch < Kepler)**
- Указатели на __global__ функции со стороны хоста поддерживаются.

__device__ функции

- __device__ может использоваться совместно с __host__, в этом случае функция компилируется в 2 видах.
- Не может быть переменного числа аргументов.
- Не могут содержать статических переменных.
- Рекурсия поддерживается с compute capability 2.0.
- Указатели на __device__ функции со стороны устройства поддерживаются с compute capability 2.0, указатели со стороны хоста не поддерживаются.
- Встраиваются по умолчанию в compute capability 1.x, есть директива компилятору __noinline__.

Встроенные векторные типы

- [u]char[1..4], [u]int[1..4], [u]long[1..4], float[1..4], double2 – являются структурами, доступ через .x, .y, .z, .w.
- Нет конструкторов, но есть make_<type name>.
- Нет встроенных векторных арифметических операций.
- dim3 = uint3 + конструктор; по умолчанию заполняется 1
  – dim3 используется для задания числа блоков и потоков при вызове ядер.

Встроенные переменные

- В коде на стороне GPU доступны следующие переменные:
  - **gridDim** – размер решетки блоков;
  - **blockIdx** – индекс блока потоков внутри решетки;
  - **blockDim** – размер блока потоков;
  - **threadIdx** – индекс потока внутри блока потоков;
  - **warpSize** – размер варпа.
- Все они являются 3-мерными векторами, доступ к компонентам через .x, .y, .z.
- (0, 0, 0) ≤ blockIdx < gridDim, (0, 0, 0) ≤ threadIdx < blockDim.
- Данные переменные предназначены только для чтения.

Вычисление уникального индекса потока

- threadIdx является «локальным» индексом потока внутри блока. Нет встроенной переменной для «глобального» индекса потока (т.е. среди всех потоков всех блоков).
- Он может быть вычислен через значения других встроенных переменных.
- Для простоты будем считать, что используется только x-компонента индексов.

idx = blockIdx.x * blockDim.x + threadIdx.x;

Искомый индекс

Смещение нулевого потока данного блока относительно нулевого потока нулевого блока

Смещение данного потока относительно нулевого потока данного блока

Пример: ядро для сложения векторов

```
/* Считаем, что ядро будет вызываться столько
  раз, какова длина векторов, и используется
  только x-компонента индексов; a, b, result –
  указатели на память GPU */
__global__ void vecAdd_kernel(
     const float * a, const float * b,
     float * result) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  result[i] = a[i] + b[i];
}
```

Пример: ядро для сложения матриц

```
// Матрицы хранятся по строкам и имеют размер m x n
__global__ void matAdd_kernel(const float * a, const
   float * b, float * result, int m, int n)

{

  // Поток вычисляет result(i, j)
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int idx = i * n + j;
  result[idx] = a[idx] + b[idx];
}
```

Вызов ядер

- Функция ядра должна быть вызвана с указанием **конфигурации исполнения**.
- Конфигурация определяется использованием выражения специального вида **<<< Dg, Db >>>** между именем функции и списком ее аргументов.
- **Dg** – определяет размер сетки, общее количество блоков равно Dg.x * Dg.y * Dg.z.
- **Db** – определяет размер блока потоков (все блоки имеют одинаковый размер), общее количество потоков равно Db.x * Db.y * Db.z.
- Кроме Dg и Db есть еще два параметра, их значения могут быть использованы по умолчанию, в данной лекции они не рассматриваются.

Вызов ядер

- Размер решетки блоков и размер блока потоков являются переменными типа dim3 (встроенный тип в CUDA).
- Конструктор по умолчанию заполняет неуказанные размерности значением 1. Таким образом, если в ядре используется только x-компонента (логически одномерная декомпозиция), в <<< … >>> можно указывать просто переменные или константы типа int.
- Пример:

```
some_kernel <<<  201, 500 >>> (some_args);
```

Запуск ядра some_kernel с аргументами some_args на решетке из 201 блока по 500 потоков в каждом, всего

201 * 500 = 100500 потоков.

Пример: вызов ядра для сложения векторов

- Используем ядро из примера.
- Будем считать, что размер блока фиксирован и равен 256.
- Необходимо вычислислить количество блоков.

```
void vecAdd(const float * a, const float *
  b, float * result, int n)
{
  const int block_size = 256;
  int num_blocks = ?;
  vecAdd_kernel <<< num_blocks, block_size
   >>> (a, b, result);
}
```

Пример: вызов ядра для сложения векторов

```
void vecAdd(const float * a, const float * b,
  float * result, int n)
{
  const int block_size = 256;
  int num_blocks =
   (n + block_size - 1) / block_size;
  vecAdd_kernel <<< num_blocks, block_size
  >>> (a, b, result);
}
```

- Все верно?

Правильное ядро для сложения векторов

```
__global__ void vecAdd_kernel(
     const float * a, const float * b,
     float * result, int n)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    result[i] = a[i] + b[i];
}
```

Способы борьбы с невыровненностью

- Брать число блоков с запасом и проверять, не выходим ли мы за данные.
- Выравнивать данные вручную, тогда можно не проверять.
- Сделать количество работы на поток нефиксированным.

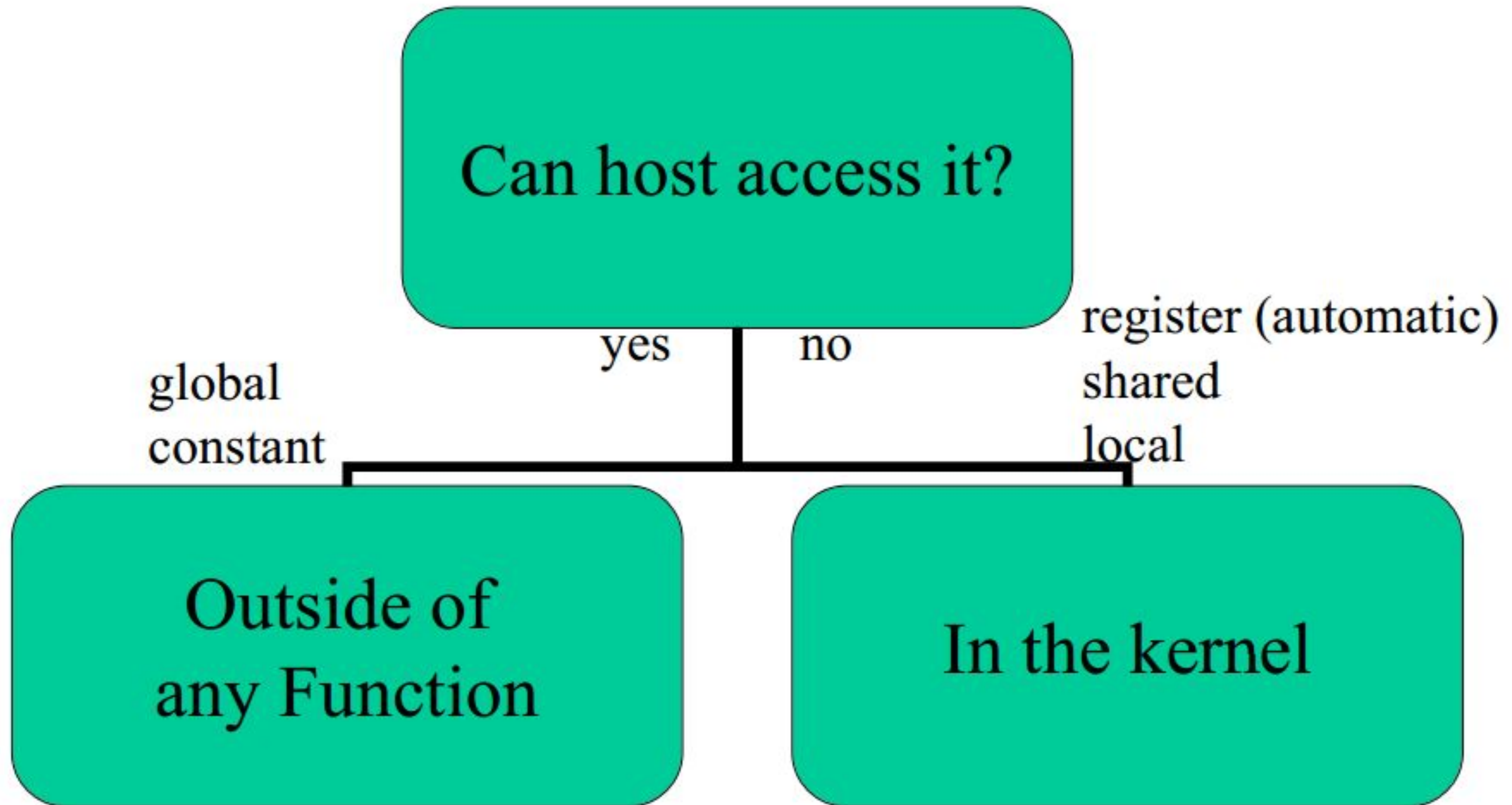
- Оптимальный вариант зависит от ситуации.

Квалификаторы переменных

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- Использование **__device__** опционально, если есть **__local__**, **__shared__** или **__constant__**.
- Переменные без какого-либо квалификатора (**автоматические**) размещаются в регистрах:
  - кроме статических массивов, которые размещаются в локальной памяти.

Квалификаторы переменных



CUDA API

- Состав CUDA API:
  - управление устройствами;
  - управление памятью;
  - управление процессом выполнения:
    - Streams;
    - Synchronization;
    - Events;
  - взаимодействие с графическими API;
  - обработка ошибок.
- Уровни API:
  - низкоуровневое (CUDA driver API): cu*;
  - высокоуровневое (C runtime for CUDA): cuda*.

CUDA API: обработка ошибок

- Все функции возвращают значение типа **cudaError_t**, **cudaSuccess** в случае успешного завершения функции.
- Получить код выполнения последней операции можно с помощью функции:

```
cudaError_t cudaGetLastError();
```

- Получить текстовое описание ошибку позволит функция:

```
const char* cudaGetErrorString(
 cudaError_t error);
```

Управление устройствами

- Перечисление устройств:
  - cudaError_t **cudaGetDeviceCount**(int* count) – возвращает число доступных устройств;
  - cudaError_t **cudaGetDevice** (int* dev) – возвращает используемое устройство;
  - cudaError_t **cudaGetDeviceProperties** (struct cudaDeviceProp* prop, int dev) – возвращает структуру, содержащую свойства устройства.

Управление устройствами

- Выбор устройства:
  - cudaError_t **cudaSetDevice** (int dev) – устанавливает устройство с заданным номером;
  - cudaError_t **cudaChooseDevice** (int* dev, const struct cudaDeviceProp* prop) – выбирает устройство, в наибольшей степени соответствующее конфигурации.

Управление памятью

- Выделение и освобождение памяти на устройстве:
  - cudaError_t **cudaMalloc** (void** devPtr, size_t count) – выделяет память на устройстве и возвращает указатель на нее;
  - cudaError_t **cudaFree** (void* devPtr) – освобождает память на устройстве.
- Копирование данных между хостом и устройством:
  - cudaError_t **cudaMemcpy** (void* dst, const void* src, size_t count, enum cudaMemcpyKind kind) – осуществляет копирование данных между хостом и устройством (блокирующий вариант);
  - **cudaMemcpyAsync** – неблокирующий вариант.

Синхронизация

- void **__syncthreads**() – барьерная синхронизация в для потоков внутри одного блока (вызывается внутри ядра).
- Атомарные арифметические функции для глобальной и разделяемой памяти (**mul** и **div** не поддерживаются!):
  - `int atomicCAS(int* address, int compare, int val);`
  - `int atomicAdd(int* address, int val);`
  - `int atomicMin(int* address, int val);`
  - …
- **cudaDeviceSyncronize**() – барьерная синхронизация для всех вызовов функций на устройстве (вызывается хостом).
- Возможна более сложная синхронизация на основе cudaStream_t и cudaEvent_t.

CUDA “Hello, World!”…

```
#include <cstdlib>
#include <iostream>
#include <cuda_runtime.h>
__global__ void vecAdd_kernel(
     const float * a, const float * b,
     float * result, int n)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    result[i] = a[i] + b[i];
}
```

CUDA “Hello, World!” …

```
int main() {
  int n = 1000;
  float * a = new float[n], * a_gpu;
  cudaMalloc((void**)&a_gpu, n *
  sizeof(float));
  float * b = new float[n], * b_gpu;
  cudaMalloc((void**)&b_gpu, n *
  sizeof(float));
  float * result = new float[n], * result_gpu;
  cudaMalloc((void**)&result_gpu, n *
  sizeof(float));
```

CUDA “Hello, World!” …

```
for (int i = 0; i < n; i++)
    a[i] = b[i] = i;

cudaMemcpy(a_gpu, a, n * sizeof(float),
   cudaMemcpyHostToDevice);
cudaMemcpy(b_gpu, b, n * sizeof(float),
   cudaMemcpyHostToDevice);
```

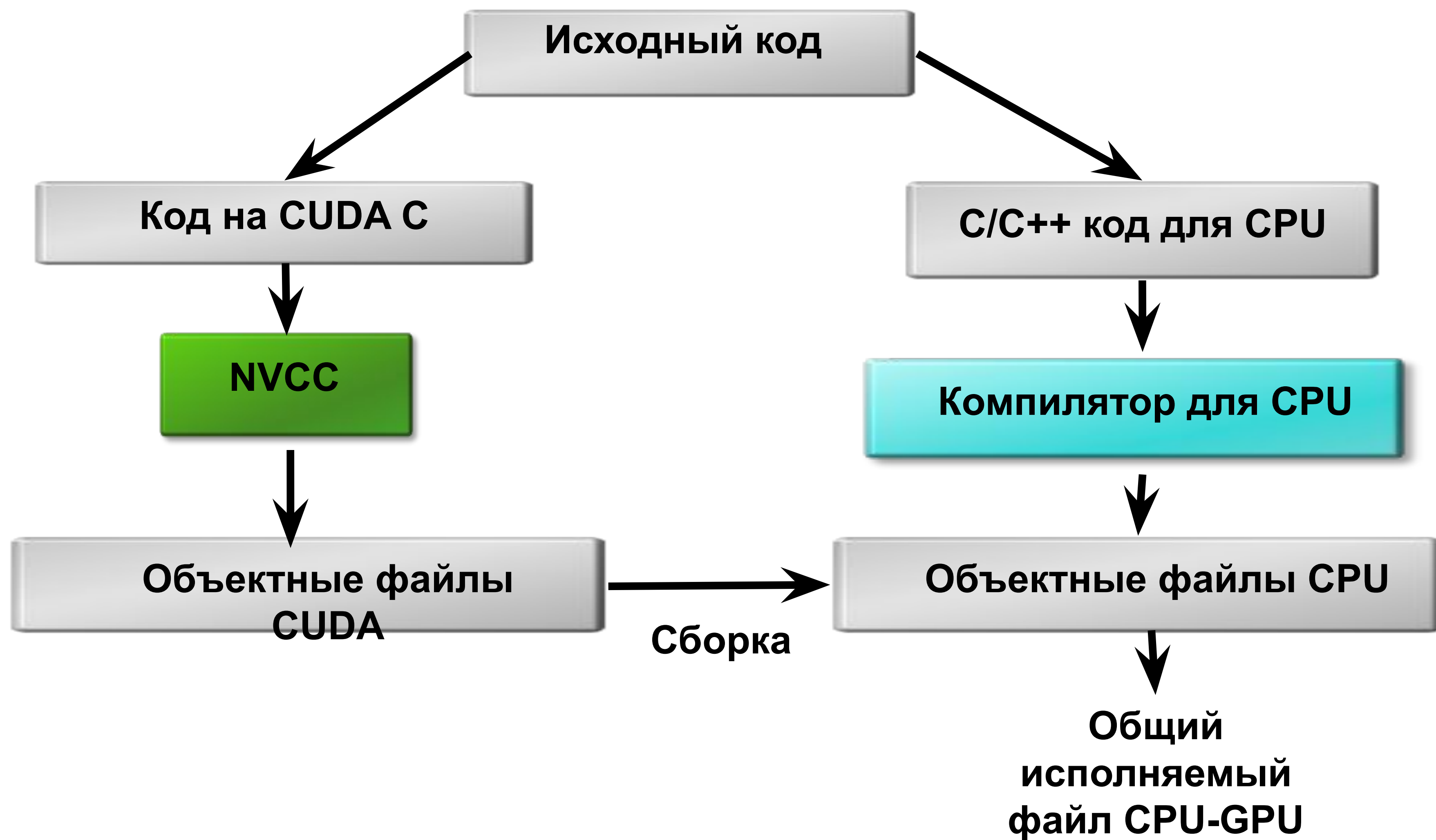
CUDA “Hello, World!”

```
const int block_size = 256;
int num_blocks =
    (n + block_size - 1) / block_size;
vecAdd_kernel <<< num_blocks, block_size
  >>> (a_gpu, b_gpu, result_gpu, n);
cudaMemcpy(result, result_gpu, n *
  sizeof(float), cudaMemcpyDeviceToHost);
delete [] a; delete [] b; delete [] result;
cudaFree(a_gpu); cudaFree(b_gpu);
  cudaFree(result_gpu);
return 0;
}
```

Компиляция и сборка

- Компилятор nvcc.
- Build rules для Microsoft Visual Studio.
- В CUDA до 4.0 поддерживались версии MSVS 2005 и 2008. В CUDA 4.0 добавлена поддержка MSVS 2010.

Компиляция и сборка



Материалы

- Линев А.В., Боголепов Д.К., Бастраков С.И. «Технологии параллельного программирования для процессоров новых архитектур» / Учебник.
- NVIDIA CUDA C Programming Guide v. 7.5.
- А.В. Боресков, А.А. Харламов «Основы работы с технологией CUDA» и материалы курса по CUDA в МГУ: https://sites.google.com/site/cudacsmsusu/file-cabinet
- Д. Сандерс, Э. Кэндрот «Технология CUDA в примерах: введение в программирование графических процессоров» (пер. с англ.).