Нижегородский государственный университет им. Н.И. Лобачевского

Факультет Вычислительной математики и кибернетики

Введение в CUDA. Язык CUDA C

Горшков А.В., Бастраков С.И.

ВМК ННГУ

anton.v.gorshkov@gmail.com

Содержание

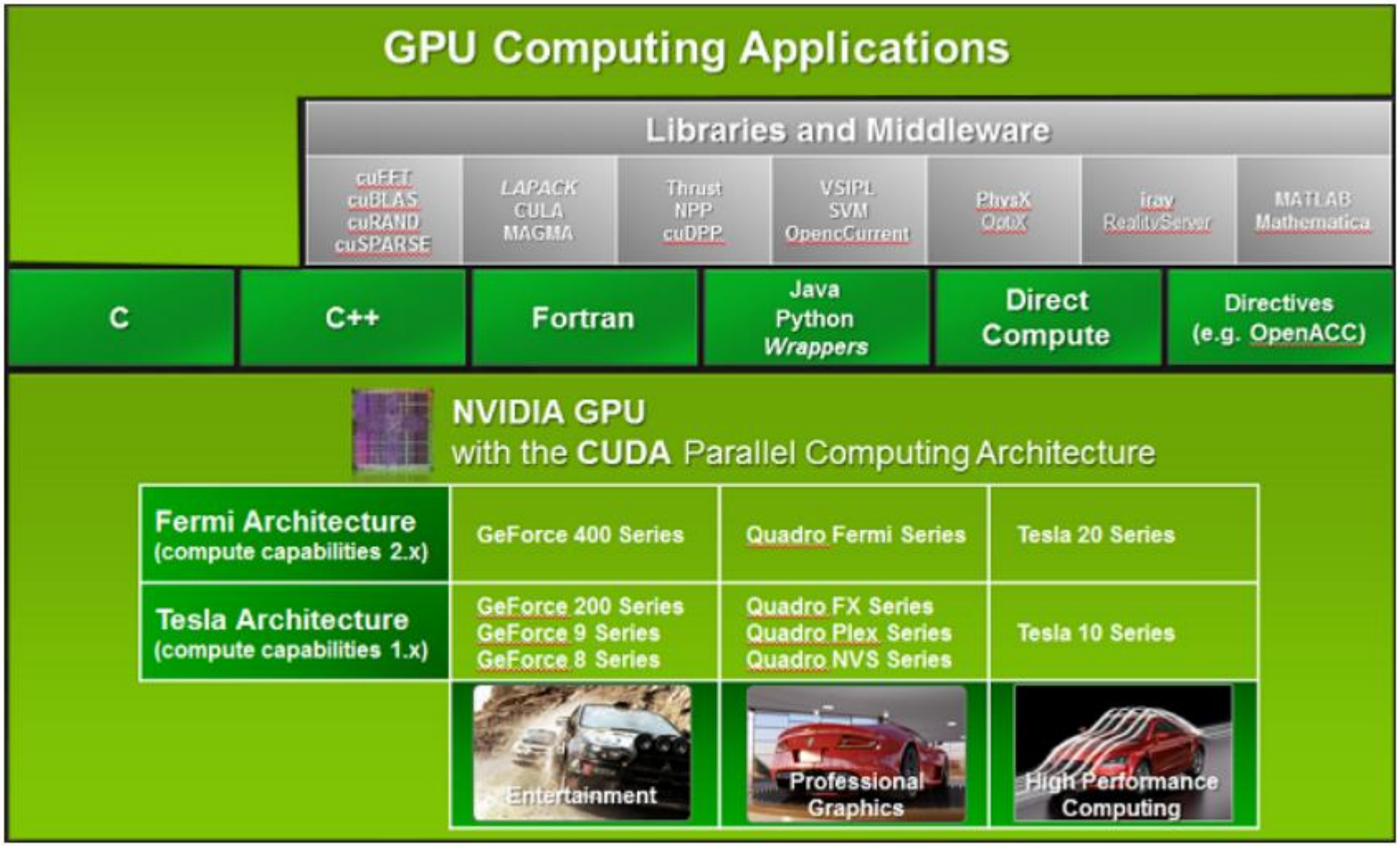
- Введение в CUDA
- Модель вычислений
- Язык CUDA C

Введение в CUDA

NVIDIA CUDA…

- **CUDA** – *Compute Unified Device Architecture*.
- Программно-аппаратная платформа для параллельных вычислений от NVIDIA.
- Раскрывает потенциал GPU для вычислений общего назначения.
- Традиционно не поддерживалась другими производителями. С выходом последних версий это стало возможным.
  - В частности, PGI CUDA-x86 http://www.pgroup.com/resources/cuda-x86.htm

Множество интерфейсов программирования



[NVIDIA CUDA C Programming Guide v. 5.0]

Комплект поставки CUDA

- CUDA driver.
- CUDA Toolkit.
  - компилятор;
  - профилировщик;
  - оптимизированные библиотеки;
  - документация.
- GPU Computing SDK.
- Поддержка основных операционных систем.

Модель вычислений

Основные понятия

- Программа с использованием GPU состоит из 2 частей:
  - Код на C/C++ (в т.ч. функция main) исполняется как обычно на CPU.
  - На GPU исполняются специальные функции – **ядра** (*kernel*) и функции, вызываемые внутри них.
- Ядро является потоковой функцией – большое количество **потоков** (*threads*) параллельно исполняют тело ядра.
- Ядро вызывается со стороны CPU, при этом указывается количество потоков, которое будет его исполнять.
- Модель вычислений на GPU во многом напоминает **потоковую модель вычислений**.
- Замечание: в русскоязычной литературе GPU-потоки часто называются нитями. Мы будем придерживаться термина «поток».

Простейшие примеры ядер

- Сложение векторов: в ядре складывается пара элементов, количество потоков равно длине векторов.
- Вычисление матричного произведения по определению: в ядре вычисляется один элемент результата, количество потоков равно количеству элементов результирующей матрицы.
- Численное решение ДУ с помощью явной разностной схемы: в ядре пересчитывается одно сеточное значение.
- *Общее свойство этих примеров – соответствие количества потоков количеству работы – является простейшим случаем, в общем случае оно необязательно.*

Параллелизм по данным

- Все приведенные примеры являются ситуациями с *независимостью по данным.*
- Это самый простой случай для распараллеливания (не только на CUDA): любое подмножество данных может обрабатываться параллельно и независимо.
- **Параллелизм по данным** – все вычислительные элементы выполняют одинаковые действия над разными данными.
- При последовательной реализации на C++ возникает цикл с независимыми итерациями.

Простейшее преобразование цикла с независимыми итерациями в ядро

- Цикл с независимыми итерациями на C++:

```
for (int i = 0; i < n; i++)
    some_computations ( i );
```

- Тело цикла превращается в тело ядра.
- Количество потоков равно количеству итераций цикла.
- Необходимо добавить код, определяющий данные, с которыми нужно работать потоку.
- Тело ядра, ядро выполняется *n* потоками:

```
int i = …; // уникальный индекс потока
some_computations ( i );
```

- Самый простой, но не всегда самый эффективный способ.

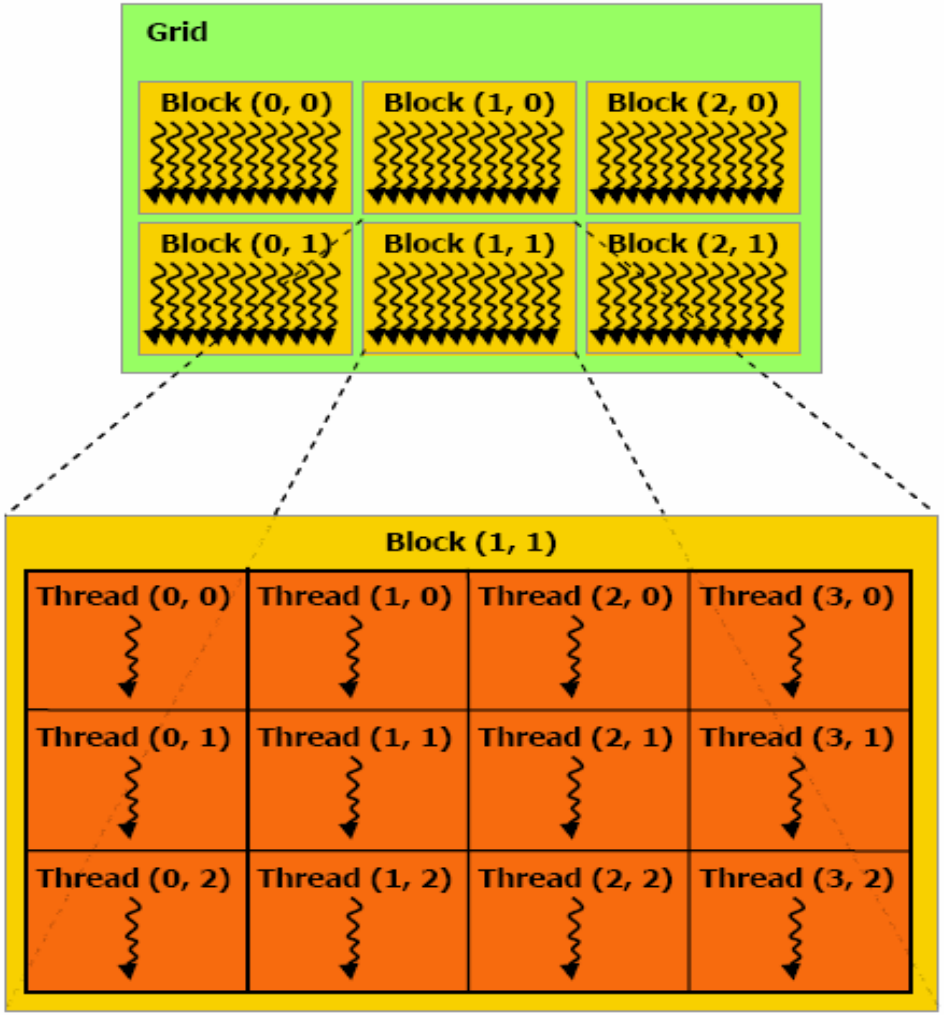
Иерархия потоков

- Для соответствия логической модели вычислений архитектуре используется иерархия потоков.
- Потоки группируются в **блоки потоков** (*thread blocks*).
- Все блоки потоков (исполняющих одно ядро) имеют **одинаковый размер**.
- Блоки объединяются в **решетку/сетку блоков** (*grid*).
- Ядро выполняется на решетке блоков. Размер блока и количество блоков в решетке задаются при вызове ядра.

Идентификаторы

- Каждый поток и блок потоков имеют **идентификаторы.**
- С их помощью каждый поток и блок могут и должны определить, с какими данными они будут работать.
- Идентификаторы являются трехмерными. Многомерная индексация может упрощать декомпозицию задачи.
- Пример: ядро выполняет умножение матриц, каждый поток вычисляет один элемент результирующей матрицы. Удобно использовать двумерные индексы, например, x-компоненту для номеров строк и y-компоненту для столбцов.

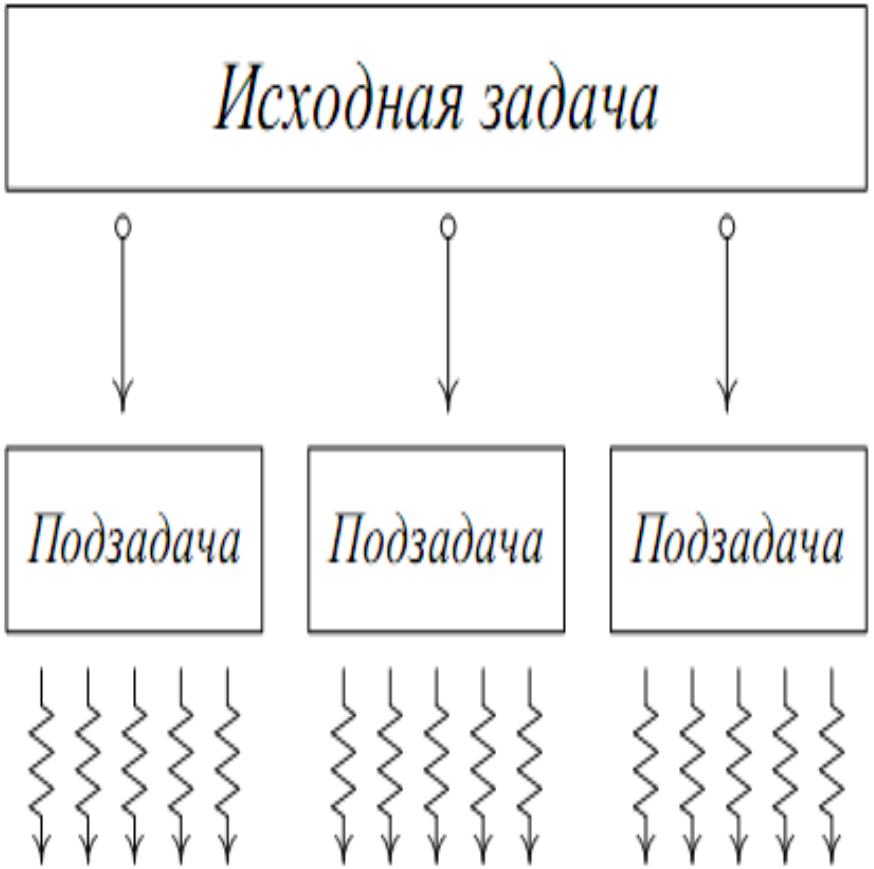
Идентификаторы



[NVIDIA CUDA C Programming Guide v. 4.0]

Типичная схема организации вычислений

- Крупнозернистый параллелизм на уровне блоков.
- Мелкозернистый параллелизм на уровне потоков внутри блока.
- Исходная задача разбивается на независимые подзадачи, каждая из которых решается параллельно одним блоком.



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

Язык CUDA C

CUDA C

- CUDA C – расширение C/C++, включающее
  - квалификаторы функций;
  - квалификаторы типов памяти;
  - встроенные переменные.
- Содержит элементы C++. Полная поддержка C++ для современных устройств в последней версии CUDA.
- В данной презентации далеко не полный обзор, лишь необходимый минимум.
- Терминология:
  - **хост** (*host*) = CPU;
  - **устройство** (*device*) = GPU;
  - **ядро** (*kernel*) – функция, параллельно выполняемая потоками на GPU.

Квалификаторы функций

Квалификатор	*Выполняется на:*	*Вызывается с:*
__host__	хост	хост
__global__	устройство	хост
__device__	устройство	устройство

- **__host__** (по умолчанию) – функция, вызываемая с хоста и выполняемая на нем; все обычные функции на C++ попадают в эту категорию.
- **__global__** – функция, вызываемая с хоста и выполняемая потоками на устройстве (ядро), всегда возвращает void.
- **__device__** – функция, вызываемая (одним потоком) и выполняемая на устройстве.

Пример синтаксиса (без глубокого смысла)

```
__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);
}
```

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.
- Все они являются 3-мерными векторами, доступ к компонентам через .x, .y, .z.
- $(0, 0, 0) \leq \text{blockIdx} < \text{gridDim}$, $(0, 0, 0) \leq \text{threadIdx} < \text{blockDim}$.
- 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.

Вызов ядер

- Вызов ядра – асинхронная операция (выход из функции осуществляется до ее реального завершения)
- Чтобы гарантировать завершение всех расчетов на GPU, используется cudaThreadSynchronize()
- Пример:

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

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.

Сложение векторов: device…

```
#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];
}
```

Сложение векторов: host…

```
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));
```

Сложение векторов: host…

```
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);
```

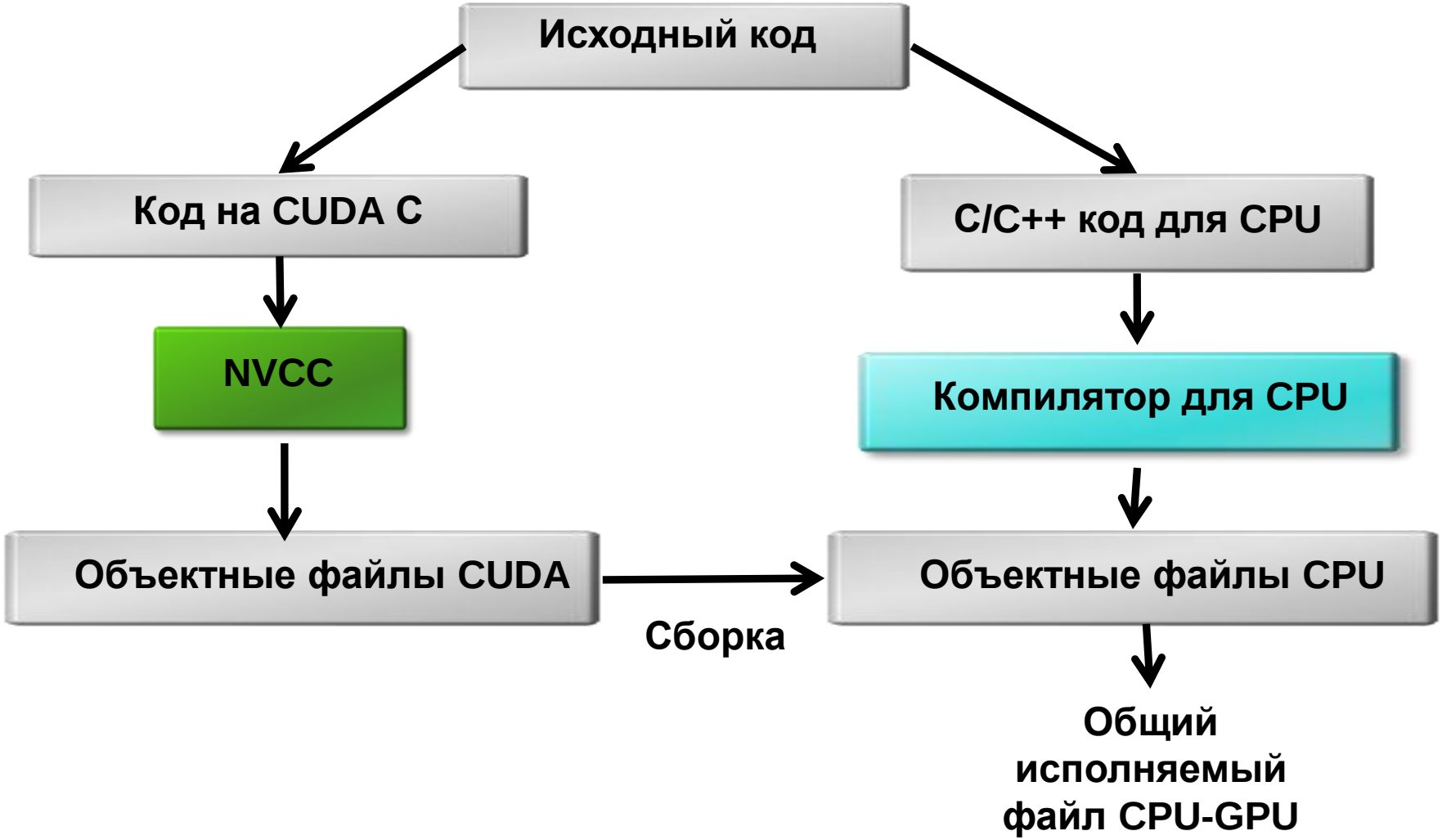
Сложение векторов: host…

```
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. 4.2.
- 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.).