

$$\frac{\text{CUDA}}{\text{CPU} + \text{GPU}}$$

CUDA

- Compute Unified Device Architecture
- NVIDIA
- C/C++
- Python
- Matlab

Установка CUDA

- GPU с поддержкой CUDA
- Драйвер GPU
- Пакет CUDA SDK
- Path - путь к компилятору cl.exe

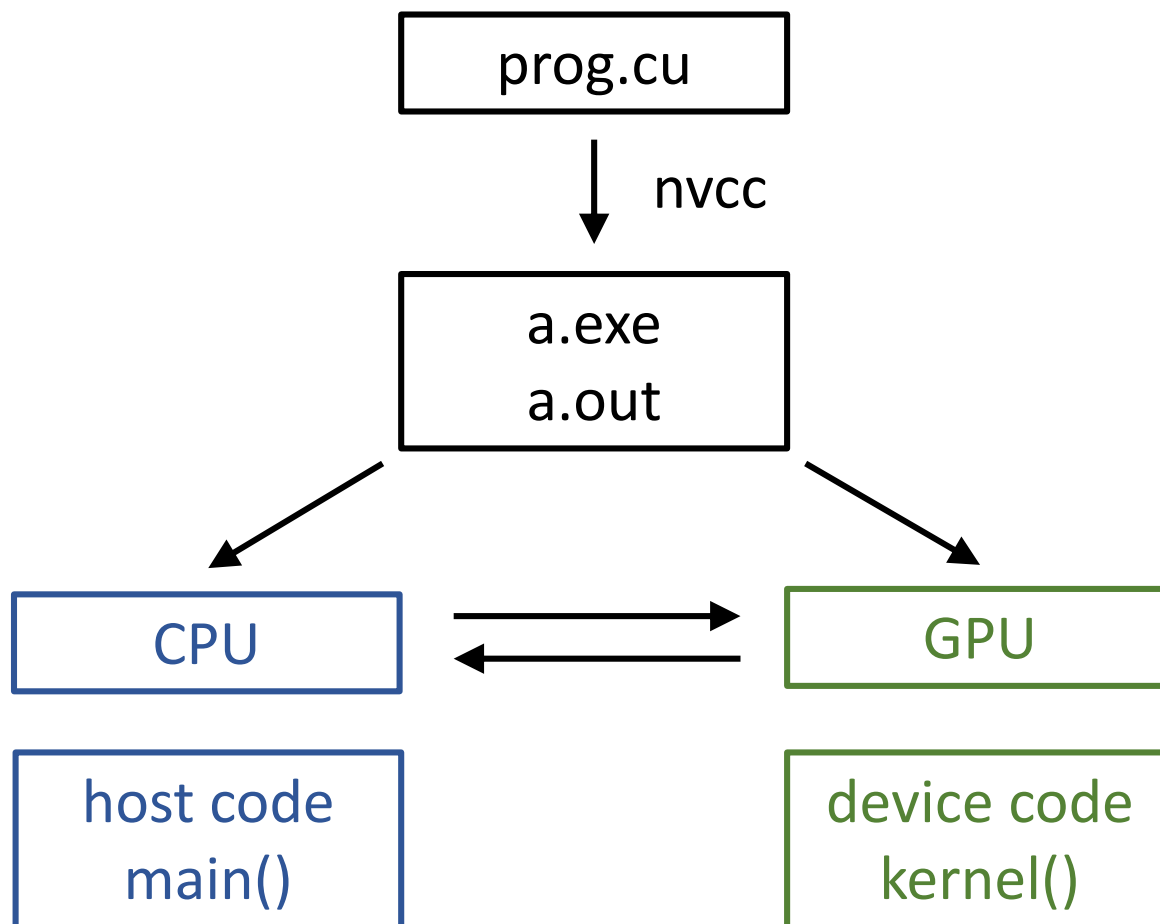
Основные понятия

- Host
 - Центральный процессор CPU
- Host memory
 - Оперативная память центрального процессора
- Device
 - Графический процессор GPU
 - Нумерация GPU: 0, 1, 2, ...
- Device memory
 - Оперативная память графического процессора
- Компилятор NVCC

Host

- Хозяин
- Принимающий (гостей), принимающая сторона
- Организовать, проводить
- ИТ
- Хостинг сайта
- Hosts – IP / host name
- Host OS / Guest OS
- Host - CPU

Программа CUDA



nvcc

- Device code
- Nvidia CUDA Compiler
- Host code
- Standard C++ host compiler
 - gcc and g++ (Linux)
 - cl.exe (Windows)

hello.c

```
#include <stdio.h>
```

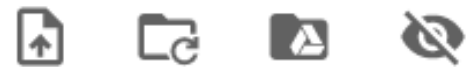
```
int main()
```

```
{
```

```
    printf("Hello, World!\n");
```

```
}
```


hello.c – gcc – a.out



..
sample_data
a.out
hello.c

✓
0s

[18] !cat hello.c

```
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
}
```

✓
0s

[20] !gcc hello.c
!./a.out

Hello, World!

Colab

- colab.research.google.com
- Google account

hello.cu (no device code)

```
#include <stdio.h>
```

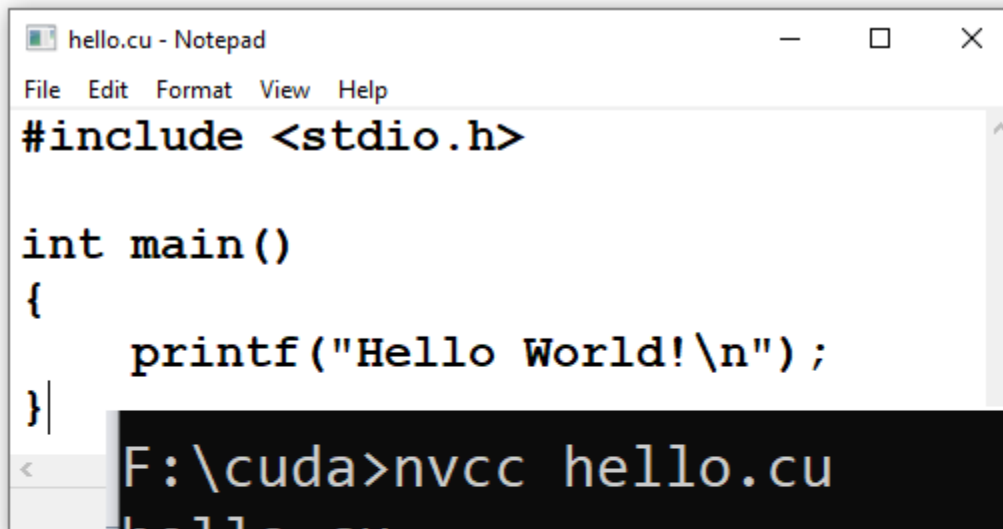
```
int main(void)
```

```
{
```

```
    printf("Hello, World!\n");
```

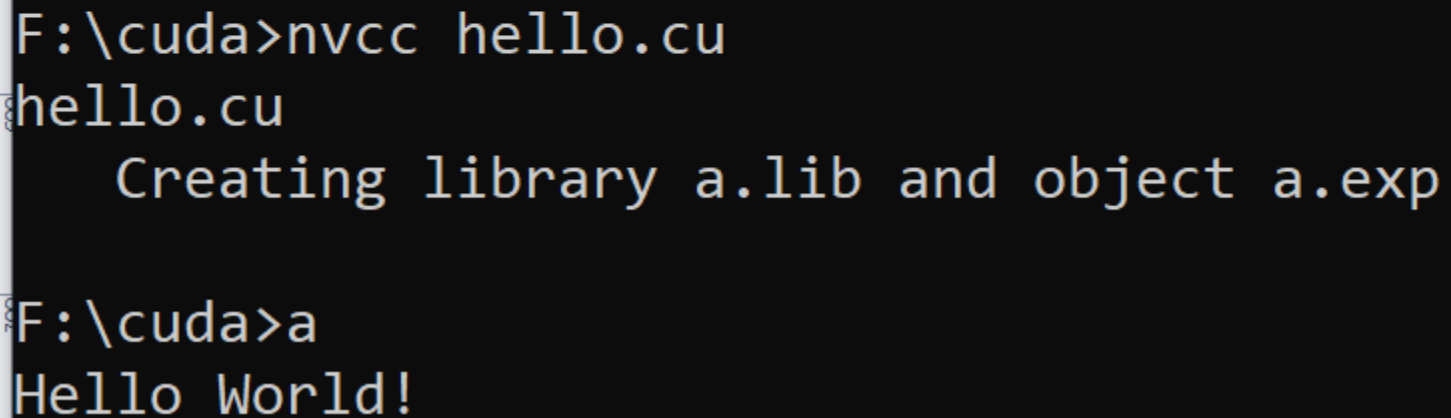
```
}
```

hello.cu → nvcc → a.exe



```
hello.cu - Notepad
File Edit Format View Help
#include <stdio.h>

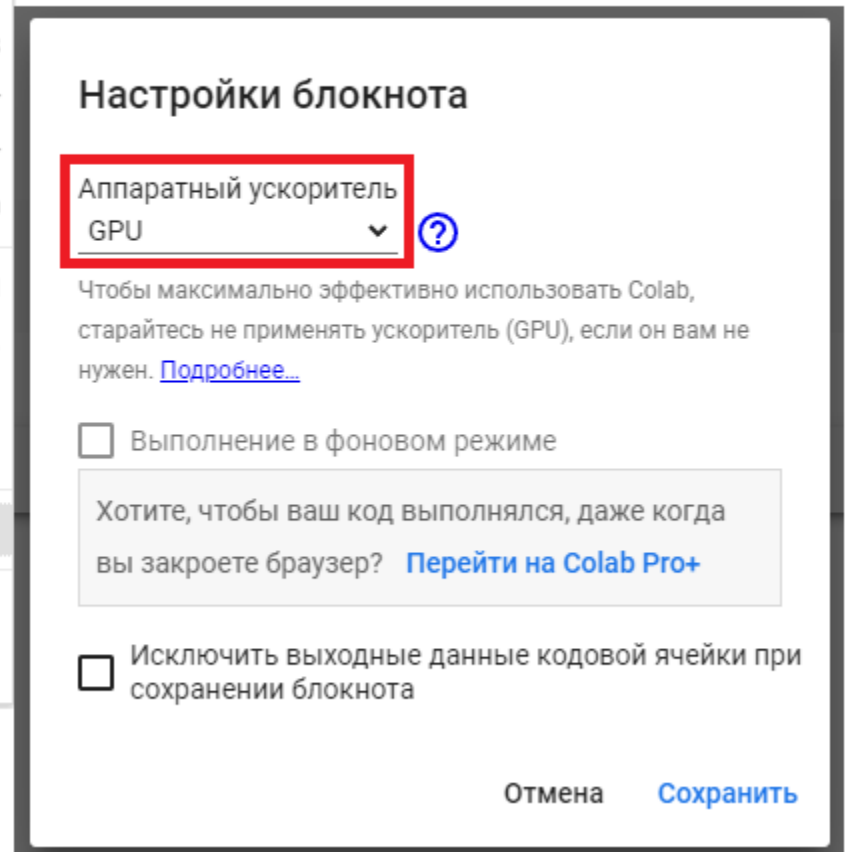
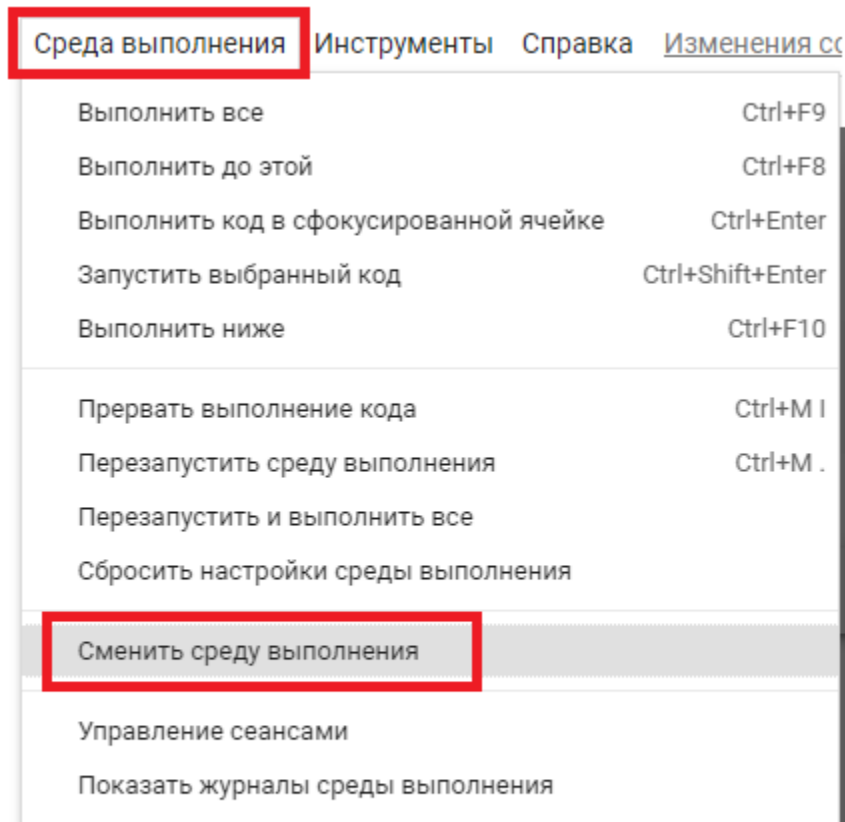
int main()
{
    printf("Hello World!\n");
}
```



```
F:\cuda>nvcc hello.cu
hello.cu
        Creating library a.lib and object a.exp

F:\cuda>a
Hello World!
```

Google Colab – Runtime – GPU



hello.cu – nvcc – a.out

The screenshot displays the CUDA-labs.ipynb Jupyter Notebook interface. The top navigation bar includes the CO logo, the notebook name 'CUDA-labs.ipynb', and a star icon. Below this is a menu with options: 'Файл', 'Изменить', 'Вид', 'Вставка', 'Среда выполнения', 'Инструменты', and 'Справка'.

The left sidebar, titled 'Файлы', shows a file explorer. It contains a search bar, icons for file operations, and a tree view. The tree view shows a folder named 'sample_data' which contains a file named 'hello.cu'. The 'hello.cu' file is highlighted with a red rectangular box.

The main area of the notebook shows two code cells. The first cell, labeled '[1]', contains the command `!cat hello.cu`. The second cell, labeled '[2]', contains the commands `!nvcc hello.cu` and `!./a.out`. The output of the second cell, 'Hello World!', is displayed below the code and is also highlighted with a red rectangular box.

dev.cu

```
#include <stdio.h>
```

```
int main()
```

```
{
```

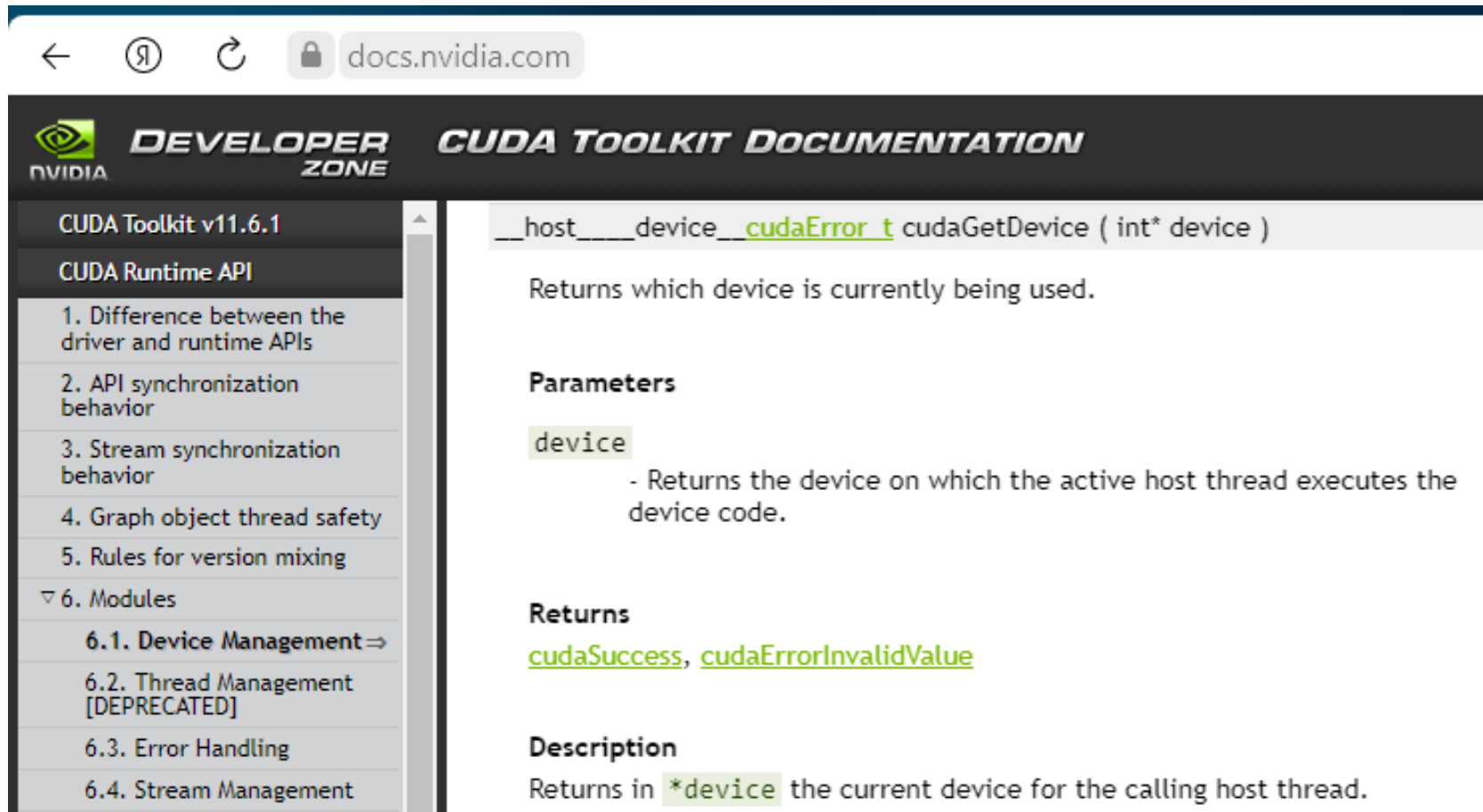
```
    int dev;
```

```
    cudaGetDevice ( &dev );
```

```
    printf( "Current Device: %d\n", dev );
```

```
}
```

cudaGetDevice



The screenshot shows a web browser with the address bar displaying 'docs.nvidia.com'. The page header includes the NVIDIA logo, 'DEVELOPER ZONE', and 'CUDA TOOLKIT DOCUMENTATION'. A left sidebar lists the 'CUDA Toolkit v11.6.1' and 'CUDA Runtime API' sections, with '6.1. Device Management' selected. The main content area displays the function signature: `__host__ __device__ cudaError_t cudaGetDevice (int* device)`. Below the signature, it states 'Returns which device is currently being used.' The 'Parameters' section lists `device` as a parameter that returns the device on which the active host thread executes the device code. The 'Returns' section lists [cudaSuccess](#) and [cudaErrorInvalidValue](#). The 'Description' section states that the function returns in `*device` the current device for the calling host thread.

← ⓘ ↻ 🔒 docs.nvidia.com

NVIDIA DEVELOPER ZONE CUDA TOOLKIT DOCUMENTATION

CUDA Toolkit v11.6.1

CUDA Runtime API

- 1. Difference between the driver and runtime APIs
- 2. API synchronization behavior
- 3. Stream synchronization behavior
- 4. Graph object thread safety
- 5. Rules for version mixing
- ▼ 6. Modules
 - 6.1. Device Management →**
 - 6.2. Thread Management [DEPRECATED]
 - 6.3. Error Handling
 - 6.4. Stream Management

`__host__ __device__ cudaError_t cudaGetDevice (int* device)`

Returns which device is currently being used.

Parameters

`device`

- Returns the device on which the active host thread executes the device code.

Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

Description

Returns in `*device` the current device for the calling host thread.

NVCC

- Nvidia CUDA Compiler

```
F:\cuda>nvcc dev1.cu
dev1.cu
    Creating library a.lib and object a.exp

F:\cuda>a
Current Device: 0
```

Output file name

```
F:\cuda>nvcc dev1.cu -o dev1.exe
dev1.cu
    Creating library dev1.lib and object dev1.exp

F:\cuda>dir dev1.* /b
dev1.cu
dev1.exe
dev1.exp
dev1.lib

F:\cuda>dev1
Current Device: 0
```

dev1.cu → nvcc → a.out

The screenshot displays the CUDA-labs.ipynb JupyterLab interface. On the left, the 'Файлы' (Files) sidebar shows a directory structure with 'sample_data' and 'dev1.cu' (highlighted with a red box). The main area shows two code cells. Cell [1] contains a terminal command to view the contents of 'dev1.cu', which is a C++ program that prints the current CUDA device. Cell [2] contains terminal commands to compile 'dev1.cu' using 'nvcc' and run the resulting 'a.out' executable, which outputs 'Current Device: 0'.

CO CUDA-labs.ipynb ☆

Файл Изменить Вид Вставка Среда выполнения Инструменты Справка [Изменения сохранены](#)

Файлы

sample_data

dev1.cu

+ Код + Текст

[1] 1 !cat dev1.cu

```
#include <stdio.h>

int main()
{
    int dev;
    cudaGetDevice ( &dev );
    printf( "Current Device: %d\n", dev );
}
```

[2] 1 !nvcc dev1.cu
2 !./a.out

Current Device: 0

Виды функций

Функция	Вызов	Выполнение
__host__	CPU	CPU
__global__	CPU	GPU
__device__	GPU	GPU

device code + host code

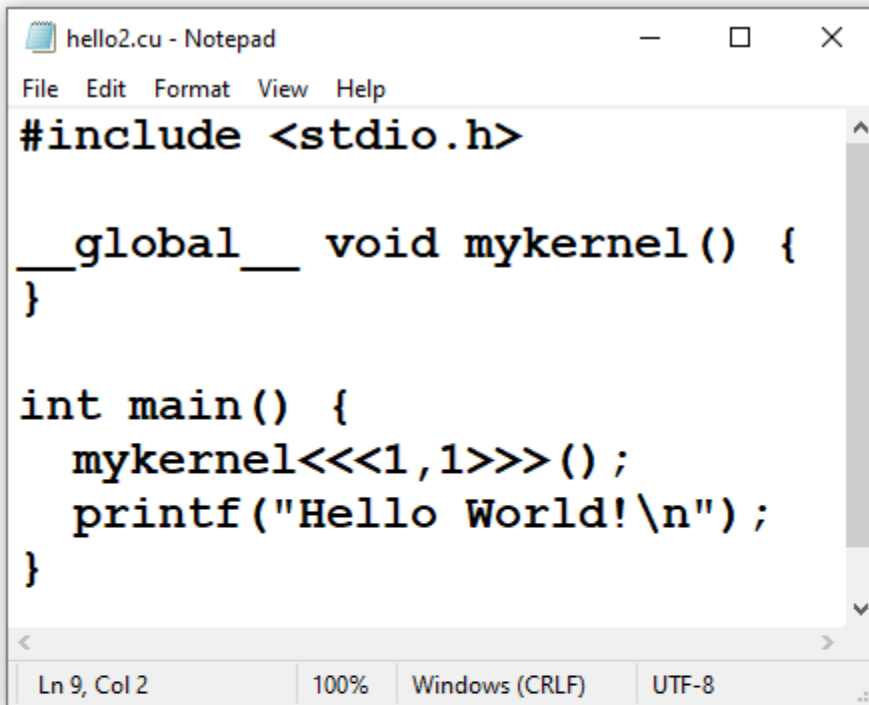
```
#include <stdio.h>
```

```
__global__ void mykernel() {  
}
```

```
int main() {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
}
```



kernel launch



```
hello2.cu - Notepad
File Edit Format View Help
#include <stdio.h>

__global__ void mykernel() {
}

int main() {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
}
```

Ln 9, Col 2 100% Windows (CRLF) UTF-8

```
1 !nvcc hello2.cu
2 !./a.out
```

Hello World!

```
F:\cuda>nvcc hello2.cu
hello2.cu
      Creating library a.lib and object a.exp

F:\cuda>a
Hello World!
```

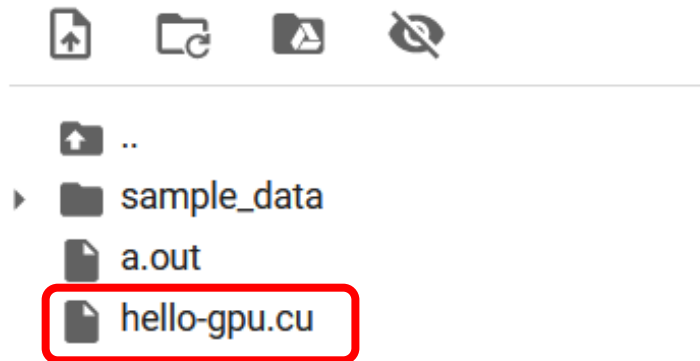
CPU + GPU

```
#include <stdio.h>
```

```
__global__ void mykernel() {  
    printf("Hello, GPU!\n");  
}
```

```
int main() {  
    mykernel<<<1,1>>>();  
    printf("Hello, CPU!\n");  
}
```

CPU + GPU: Результаты



✓
0s

!cat hello-gpu.cu

```
#include <stdio.h>

__global__ void mykernel() {
    printf("Hello, GPU!\n");
}

int main() {
    mykernel<<<1,1>>>();
    printf("Hello, CPU!\n");
}
```

✓
1s

[13] !nvcc hello-gpu.cu
!./a.out

Hello, CPU!

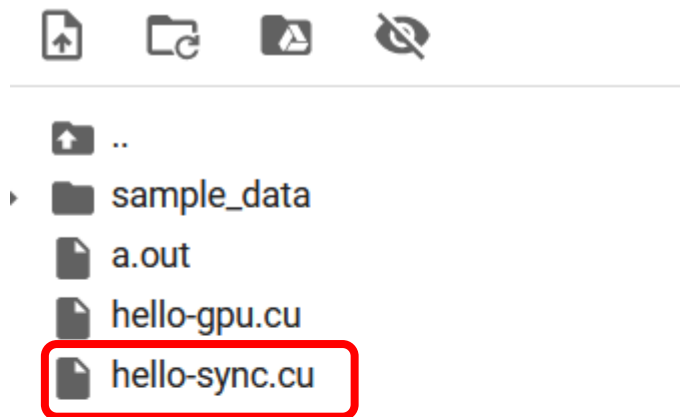
Синхронизация

```
#include <stdio.h>
```

```
__global__ void mykernel() {  
    printf("Hello, GPU!\n");  
}
```

```
int main() {  
    mykernel<<<1,1>>>();  
    printf("Hello, CPU!\n");  
    cudaDeviceSynchronize();  
}
```

Синхронизация: Результаты



```
✓ [14] !cat hello-sync.cu
1s

#include <stdio.h>

__global__ void mykernel() {
    printf("Hello, GPU!\n");
}

int main() {
    mykernel<<<1,1>>>();
    printf("Hello, CPU!\n");
    cudaDeviceSynchronize();
}
```

```
✓ [15] !nvcc hello-sync.cu
0s
!./a.out

Hello, CPU!
Hello, GPU!
```

GPU: 5 потоков

```
#include <stdio.h>
```

```
__global__ void mykernel() {  
    printf("Hello, GPU!\n");  
}
```

```
int main() {  
    mykernel<<<1,5>>>();  
    printf("Hello, CPU!\n");  
    cudaDeviceSynchronize();  
}
```



- ..
- sample_data
- a.out
- hello-gpu.cu
- hello-sync-5.cu
- hello-sync.cu

✓
0s

[16] !cat hello-sync-5.cu

```
#include <stdio.h>

__global__ void mykernel() {
    printf("Hello, GPU!\n");
}

int main() {
    mykernel<<<1,5>>>();
    printf("Hello, CPU!\n");
    cudaDeviceSynchronize();
}
```

✓
1s

[17] !nvcc hello-sync-5.cu
!./a.out

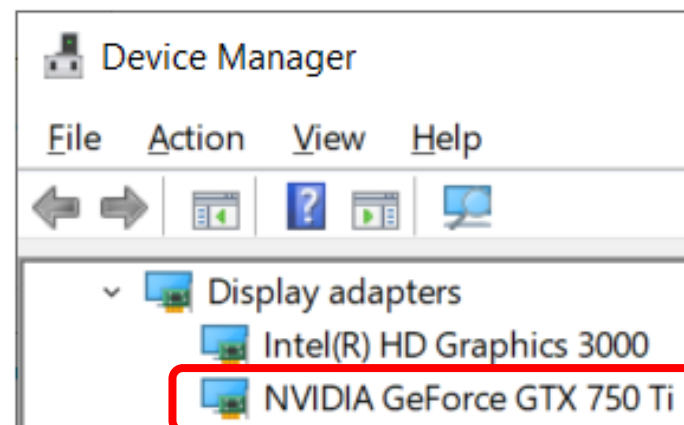
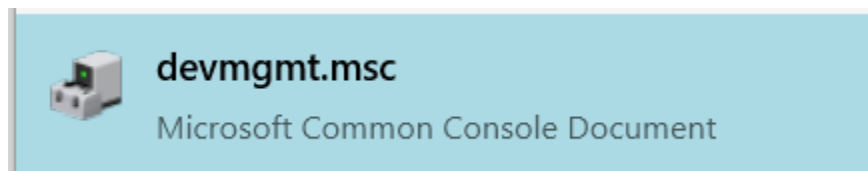
```
Hello, CPU!
Hello, GPU!
Hello, GPU!
Hello, GPU!
Hello, GPU!
Hello, GPU!
```

Задание

- Википедия
 - CUDA (RU/EN)
- Версия / Архитектура / Видеокарты
 - 1.0 / Tesla / 8800, 870
 - 5.0 / Maxwell / 750 Ti / M10
 - 8.0 / Ampere / 3060, A100
- Version features and specifications
 - Compute capability (version)
 - Data Types
 - Tensor Core

Модель видеокарты

- Диспетчер устройств / Device Manager
 - [Win + X] – Device Manager
 - Start – Settings – Devices - Printers – Device manager
 - [Win + R] – devmgmt.msc
 - Search – Device Manager



Параметры видеокарты

- [nvidia.com](https://www.nvidia.com) – Поиск – Спецификации
- Wiki – CUDA – GPUs – compute capability – 5.x (Maxwell) – CUDA SDK 6.5 ... 11.8

GeForce GTX 750 Ti

Спецификации GPU	
Ядер CUDA	640
Базовая тактовая частота	1020 MHz
Тактовая частота с ускорением	1085 MHz
Спецификации памяти	
Быстродействие памяти (Gbps)	5,4
Объем памяти	2048 MB
Интерфейс памяти	128-bit GDDR5
Максимальная полоса пропускания памяти	86,4 GB/sec
Возможности	
Программное окружение	CUDA

Версия компилятора

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver  
Copyright (c) 2005-2021 NVIDIA Corporation  
Built on Sun_Feb_14_21:12:58_PST_2021  
Cuda compilation tools, release 11.2, V11.2.152  
Build cuda_11.2.r11.2/compiler.29618528_0
```


Параметры, ключи, опции...

- Один минус / черточка / дефис
 - `ls -l`
 - `ls -a`
 - `ls -la`
 -
- Двойной минус / черточка / дефис
 - `nvcc --version`
 - `nvcc --help`

System Management Interface

```
!nvidia-smi
```

Sun Nov 6 09:10:58 2022

NVIDIA-SMI 460.32.03			Driver Version: 460.32.03			CUDA Version: 11.2		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC		
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.	
0	Tesla T4	off	00000000:00:04.0	off			0	
N/A	36C	P8	9W / 70W	0MiB / 15109MiB	0%	Default	N/A	

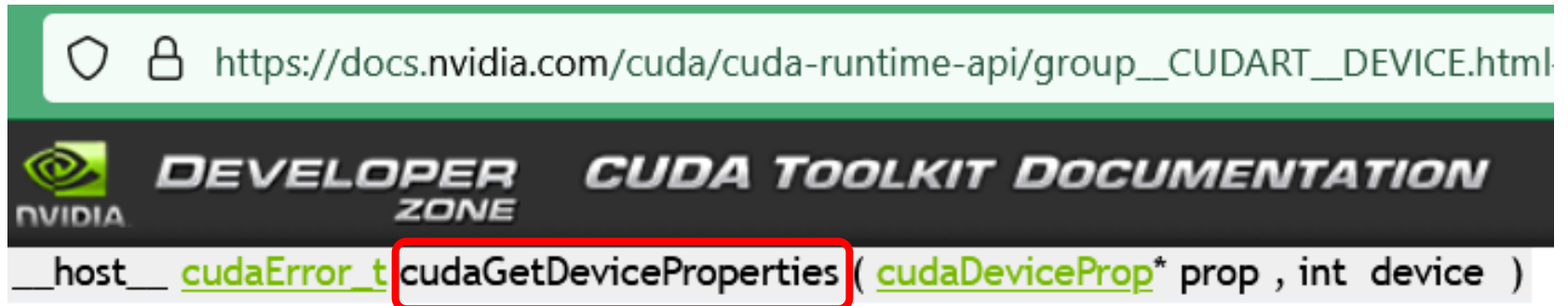
```
!nvidia-smi --list-gpus
```

```
GPU 0: Tesla T4 (UUID: GPU-53534127-5127-60bb-6b1f-a6af33046c2d)
```

Спецификации видеокарты

- NVIDIA Tesla T4
 - Версия 7.5 / Архитектура Turing
 - Ядра NVIDIA CUDA 2560
 - Производительность операций с одинарной точностью (FP32) 8,1 Терафлопс
 - Операции INT8 130 тера-операций в секунду (TOPS)
 - Объем видеопамати 16 ГБ GDDR6
 - Пропускная способность памяти 320+ Гбит/с
 - Энергопотребление 70 Вт
- Цена?

cudaGetDeviceProperties



Returns information about the compute-device.

Parameters

prop

- Properties for the specified device

device

- Device number to get properties for

Параметры видеокарты

```
#include <stdio.h>
```

```
int main(int argc, char **argv) {  
    cudaDeviceProp props;  
    cudaGetDeviceProperties(&props, 0);  
    printf("GPU %d: %s\n", 0, props.name);  
    printf("Compute capability: %d.%d\n",  
props.major, props.minor);  
}
```

Формат вывода

- **`printf("%f", x)`**
- **`%d`** - decimal
- **`%x` `%h`** - hexadecimal
- **`%o`** - octal
- **`....`** - binary?

Результаты

```
!cat comp_cap.cu
```

```
#include <stdio.h>

int main(int argc, char **argv) {
    cudaDeviceProp props;
    cudaGetDeviceProperties(&props, 0);
    printf("GPU %d: %s\n", 0, props.name);
    printf("Compute capability: %d.%d\n", props.major, props.minor);
}
```

```
!nvcc comp_cap.cu
```

```
!./a.out
```

```
GPU 0: Tesla T4
Compute capability: 7.5
```

Динамические переменные

Calls	C function	CUDA API
Memory	Host	Device
Выделение памяти	malloc()	cudaMalloc()
Копирование данных	memcpy()	cudaMemcpy()
Освобождение памяти	free()	cudaFree()

cudaMemcpy



Copies data between host and device.

Parameters

`dst`

- Destination memory address

`src`

- Source memory address

`count`

- Size in bytes to copy

`kind`

- Type of transfer

```
!cat add.cu
```

```
#include <stdio.h>

__global__ void add(int *x) {
    *x += 10;
}

int main(void) {
    int cpu_a, *gpu_a, size = sizeof(int);
    cpu_a = 2;
    cudaMalloc((void **)&gpu_a, size);
    cudaMemcpy(gpu_a, &cpu_a, size, cudaMemcpyHostToDevice);
    add<<<1,1>>>(gpu_a);
    cudaMemcpy(&cpu_a, gpu_a, size, cudaMemcpyDeviceToHost);
    cudaFree(gpu_a);
    printf("a = %d\n", cpu_a);
}
```

```
!nvcc add.cu
```

```
!./a.out
```

```
a = 12
```

Ссылки и указатели

Grid / Cluster / Block / Thread

- Grid of Thread Blocks
 - Сетка блоков потоков
 - Thread Block \leq 1024 Threads
- **myKernel<<<numBlcks , threadsPerBlck>>>()**
 - 1D, 2D, 3D сетки и блоки
- Warp: group of threads
 - Группа параллельных потоков
- Grid of Thread Block Clusters
 - Сетка кластеров блоков потоков

Глобальный индекс потока

- Одномерная модель

- `threadIndex =`

`blockIdx.x * blockDim.x +
threadIdx.x`

```
%%writefile idx.cu
#include <stdio.h>
__global__ void myKernel() {
int t = threadIdx.x;
int b = blockIdx.x;
int d = blockDim.x;
int g = d * b + t;
printf("Global %d Thread %d Block %d\n",g,t,b) ;
}
```

```
int main() {
// 2 блока по 2 потока
myKernel<<<2, 2>>>();
cudaDeviceSynchronize();
return 0;
}
```

```
Global 2 Thread 0 Block 1
Global 3 Thread 1 Block 1
Global 0 Thread 0 Block 0
Global 1 Thread 1 Block 0
```

SIMT / SIMD

- Flynn's Taxonomy of computer architectures
 - Классификация Флинна
 - Numbers of instruction and data streams
- SIMD (Single Instruction, Multiple Data)
 - Один набор инструкций на множестве данных
 - Same operation across many data elements
 - Single / scalar thread issues vector instructions applied to data vectors
- SIMT (Single Instruction, Multiple Threads)
 - Один набор инструкций на множестве потоков
 - Multiple threads issue common instructions to arbitrary data
 - Более гибкое выполнение потоков
 - Собственная локальная память
 - Разные пути выполнения / divergent control flow paths
 - Может работать с разными данными

Synchronization Primitives

- Atomics
 - atomic objects and operations
- Latches
 - single-phase asynchronous thread coordination mechanism
- Barriers
 - multi-phase asynchronous thread coordination mechanism
- Semaphores
 - constraining concurrent access / mutual exclusion
- Pipelines
 - Blocks current thread until prior pipeline stage complete

Атомарные операции

- Атомарная сумма **atomicAdd()**
- Безопасно добавляет значение к переменной в глобальной или общей памяти
- Предотвращает ситуацию гонки
- **atomicAdd(&totalSum, partialSum) ;**

Синхронизация потоков

- Взаимодействие потоков
 - Внутри блока - разделяемая память
 - Между блоками — глобальная память
- Барьерная синхронизация
 - `__syncthreads()`
 - `__threadfence_block()`
 - `__threadfence()`
 - `__threadfence_system()`

Структура GPU

- GPU
 - Видеокарта
 - Global Memory - Глобальная память / видеопамять
- Streaming Multiprocessors (SM)
 - потоковые мультипроцессоры
- Scalar Processors
 - Скалярные процессоры (ядра)
 - Shared Memory - Разделяемая память
 - Registers - Регистры
 - Warp

Виды памяти

- Регистровая память (register) компилятор
- Локальная память (local memory)
- Разделяемая память (shared memory) между потоками одного блока
 - `__shared__`
- Глобальная память (global memory)
 - `__global__` / `cudaMallocXXX` / `__device__`
- Константная память (constant memory) запись с хоста, чтение с GPU
 - `__constant__`
- Текстурная память (texture memory)

Общий шаг цикла

- Шаг с учетом размера сетки и блока
- **`stride = blockDim.x * gridDim.x;`**

Материалы

CUDA Tutorial

<https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/>

Тумаков 2017 Технология программирования CUDA

<https://e.lanbook.com/book/130543>

Минязев 2021 Параллельное программирование (MPI, OpenMP, CUDA)

<https://e.lanbook.com/book/264890>

Тоуманнен 2020 Программирование GPU при помощи Python и CUDA

<https://e.lanbook.com/book/179469>

Елесина 2020 Основы работы с технологией параллельных вычислений CUDA

<https://e.lanbook.com/book/220436>

Паттерсон 2018 Глубокое обучение с точки зрения практика

<https://e.lanbook.com/book/116122>

Боресков 2010 Основы работы с технологией CUDA

<https://e.lanbook.com/book/1260>

Малявко 2015 Параллельное программирование на основе OpenMP, MPI, CUDA

<https://e.lanbook.com/book/118245>