Introduction to GPU Computing

Beyond the CPU

- Increasement of single-core CPU performance is slow
- 10 years ago, the solution was *multicore* and increased parallelism
- Applications need increased performance
  - Big data, brain simulation, scientific simulation, ...

- So, if you're going to write parallel code, are there faster, viable alternatives to the CPU?

What is GPU?

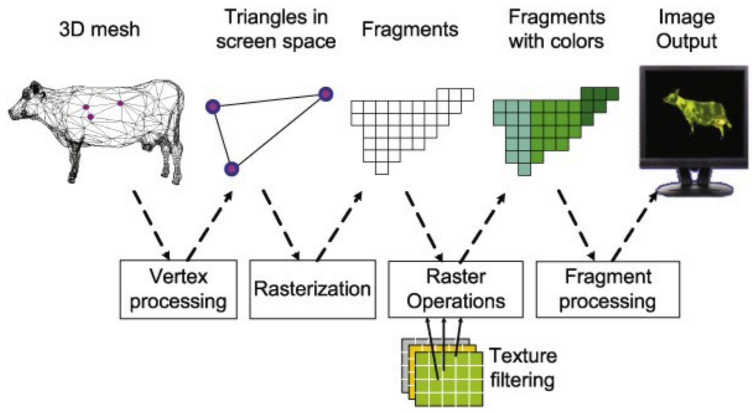
Graphics Processing Unit (GPU), is an electronic circuit that,

- Used to through rapid memory manipulation and massive parallel data processing, accelerates the building of images intended for output to a display.
- Right now, GPUs are used in almost all customer end personal computers, game consoles, professional workstations and even the cell phone you are holding.

The rise of the GPU - 1990s

Before GPU was introduced, CPU did all the graphic processing tasks.

- The CPU has always been slow for Graphics Processing
  - Visualization
  - Games



- Graphics processing is inherently parallel and there is a lot of parallelism – O(*pixels*)
- GPUs were built to do graphics processing *only*
- Initially, hardwired logic replicated to provide parallelism
  - Little to no programmability

GPUs – a decade ago

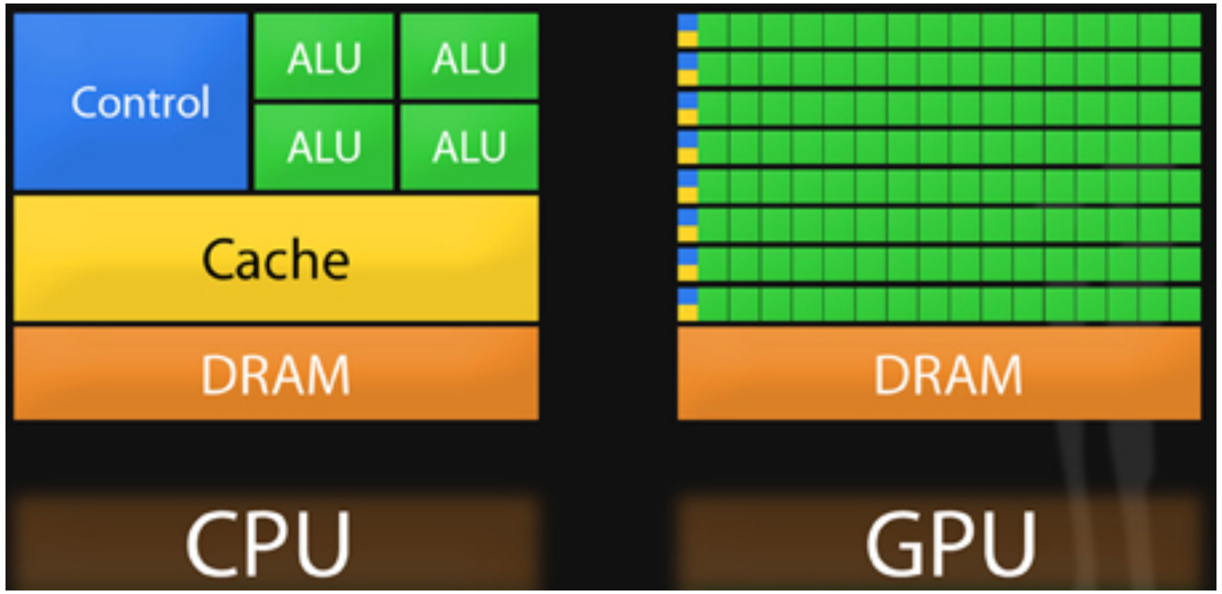
- Like CPUs, GPUs benefited from Moore's Law
- Evolved from fixed-function hardwired logic to flexible, programmable ALUs
- Around 2004, GPUs were programmable “enough” to do some non-graphics computations
  - Severely limited by graphics programming model (shader programming)
- In 2006, GPUs became “fully” programmable
  - NVIDIA releases “CUDA” language to write non-graphics programs that will run on GPUs

GPGPU

After two decades of development, GPUs eventually outpaced CPUs as they had more transistors, ran faster and were capable of doing parallel computation more efficiently.
GPUs became so complex that they are basically computers in themselves, with their own memory, buses and processors. Therefore, sometimes GPU is like an extra brain (supplemental processors) to the computer system.

As GPU harnessed more and more horsepower, GPU manufactures, such as NVIDIA and ATI/AMD, found a way to use GPUs for more general purposes, not just for graphics or videos. This gave birth to CUDA structure and CUDA C Programming Language, NVIDIA's response on facilitating the development of General Purpose Graphics Processing Unit (GPGPU).

What is the difference between GPU and CPU?



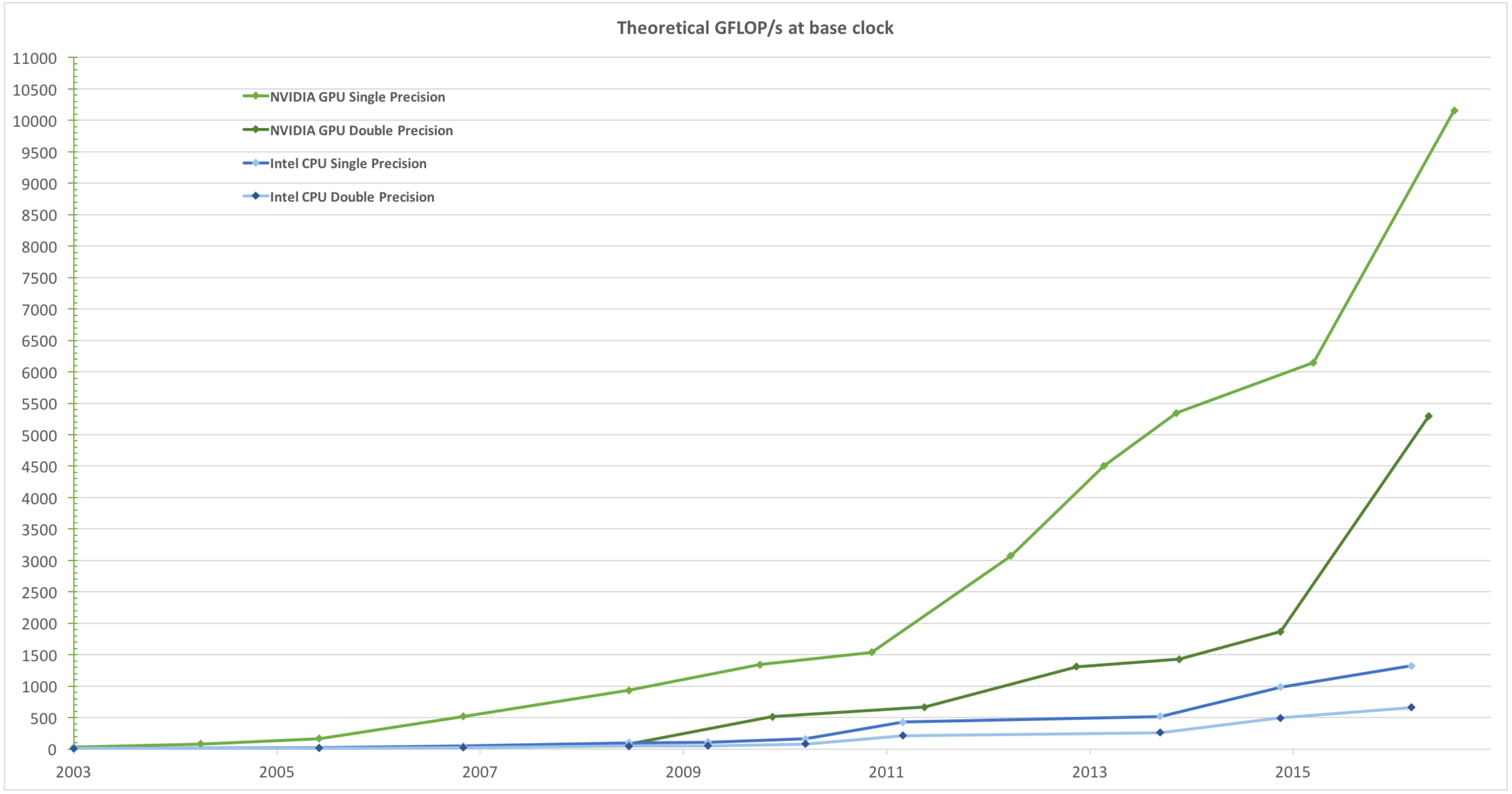
Cache is designed for data caching;
Control is designed for flow control;
ALU (Arithmetic Logic Unit) is designed for data processing.

What is the difference between GPU and CPU?

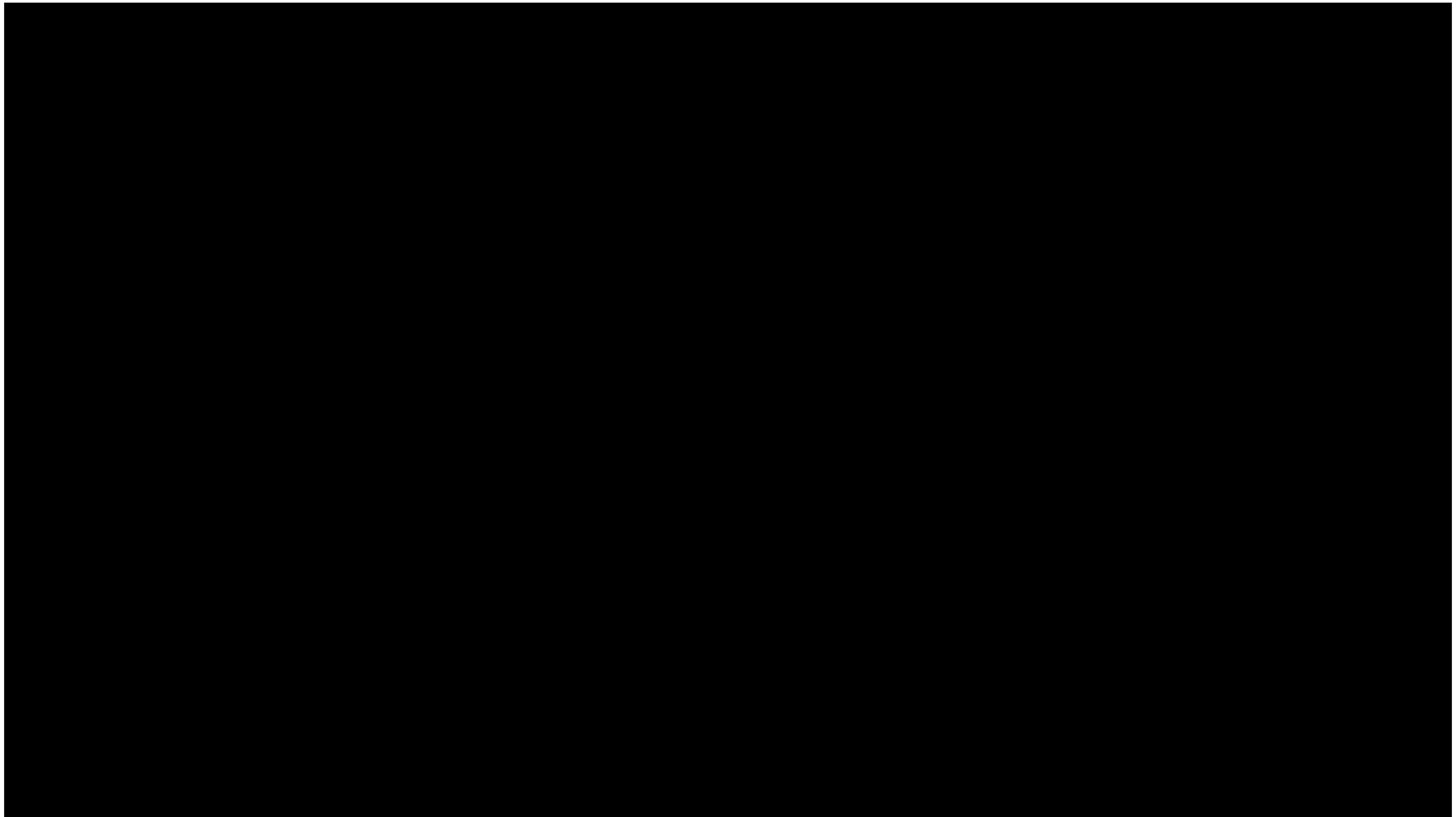
Major difference: GPU has a highly parallel structure which makes it more effective than CPU if used on data that can be partitioned and processed in parallel. To be more specific, GPU is highly optimized to perform advanced calculations such as floating-point arithmetic, matrix arithmetic and so on.

The reason behind the difference of computation capability between CPU and GPU is that GPU is specialized for compute-intensive and highly parallel computations, which is exactly what you need to render graphics. The design of GPU is more of data processing than data caching and flow control. If a problem can be processed in parallel, it usually means two things: first, same problem is executed for each element, which requires less sophisticated flow control; second, dataset is massive and problem has high arithmetic intensity, which reduces the need for low latency memory.

What is the difference between GPU and CPU?



What is the difference between GPU and CPU?



What is the advantage of using GPU for computation?

Nowadays, most of the scientific researches require massive data processing. What scientist usually do right now is to have all the data being processed on supercomputing clusters. Although most universities have constructed their own parallel computing clusters, researchers still need to compete for time to use the shared resources that not only cost millions to build and maintain, but also consume hundreds of kilowatts of power.

Different from traditional supercomputers that are built with many CPU cores, supercomputers with a GPU structure can achieve same level of performance with less cost and lower power consumption. Personal Supercomputer (PSC) based on NVIDIA's Tesla companion processors, was first introduced in 2008. The first generation four-GPU Tesla personal supercomputer have 4 Teraflops of parallel supercomputing performance, more than enough for most small researches. All it takes is 4 Tesla C1060 GPUs with 960 cores and two Intel Xeon processors. Moreover, Personal supercomputer is also very energy efficient as it can even run off a 110 volt wall circuit. Although supercomputer with GPUs cannot match the performance of the top ranking supercomputers that cost millions even billions, it is more than enough for researchers to perform daily research related computations in subjects like bioscience, life science, physics and geology.

Using a GPU

- You must retarget code for the GPU
- The working set must fit in GPU RAM
- You must copy data to/from GPU RAM
- Data accesses should be streaming
- Lots of parallelism preferred (throughput, not latency)
- High arithmetic intensity (FLOPs/byte) preferred

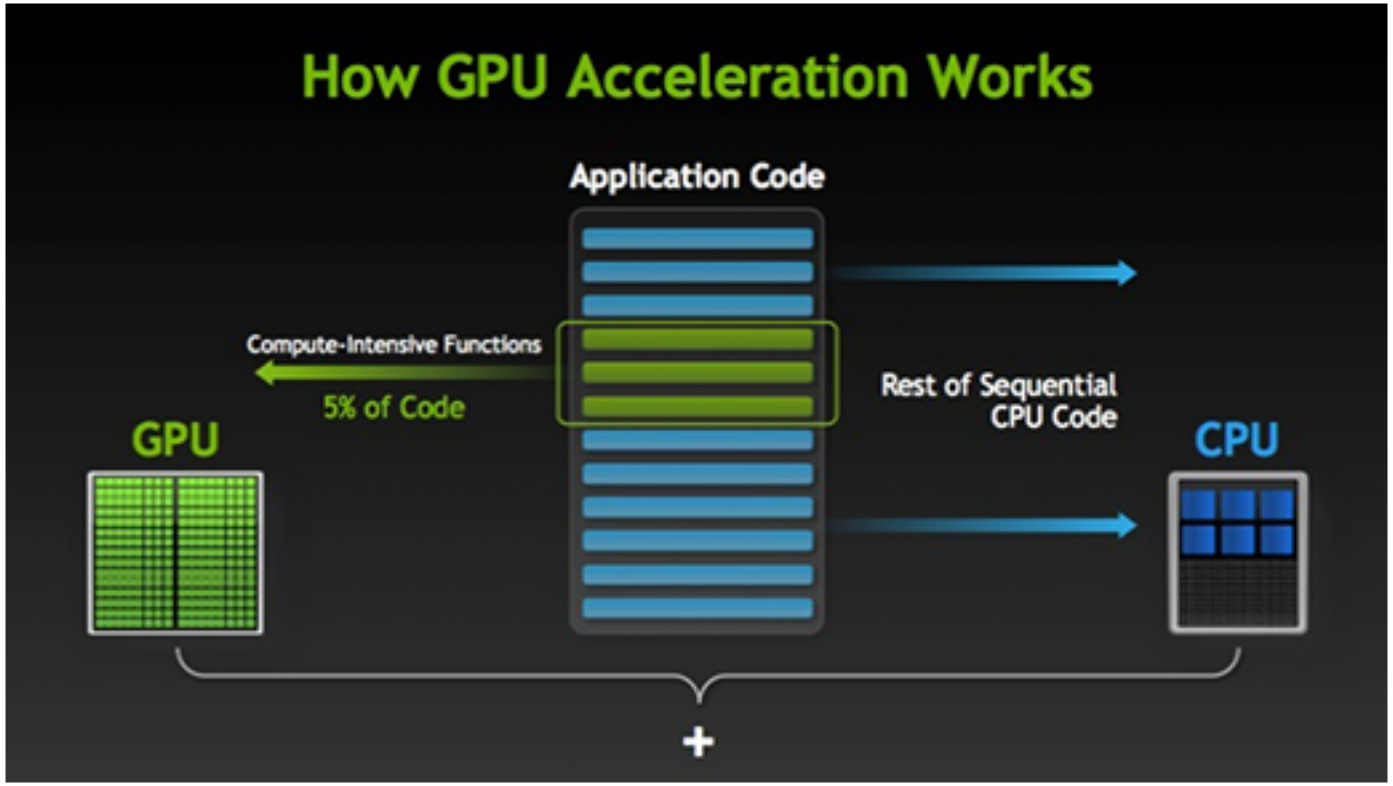
Programming

When computer scientists first attempted to use GPUs for scientific computing, the scientific codes had to be mapped onto the matrix operations for manipulating triangles. This was incredibly difficult to do and took a lot of time and dedication.

However, there are now high-level languages (such as CUDA and OpenCL) that target the GPUs directly, so GPU programming is rapidly becoming mainstream in the scientific community.

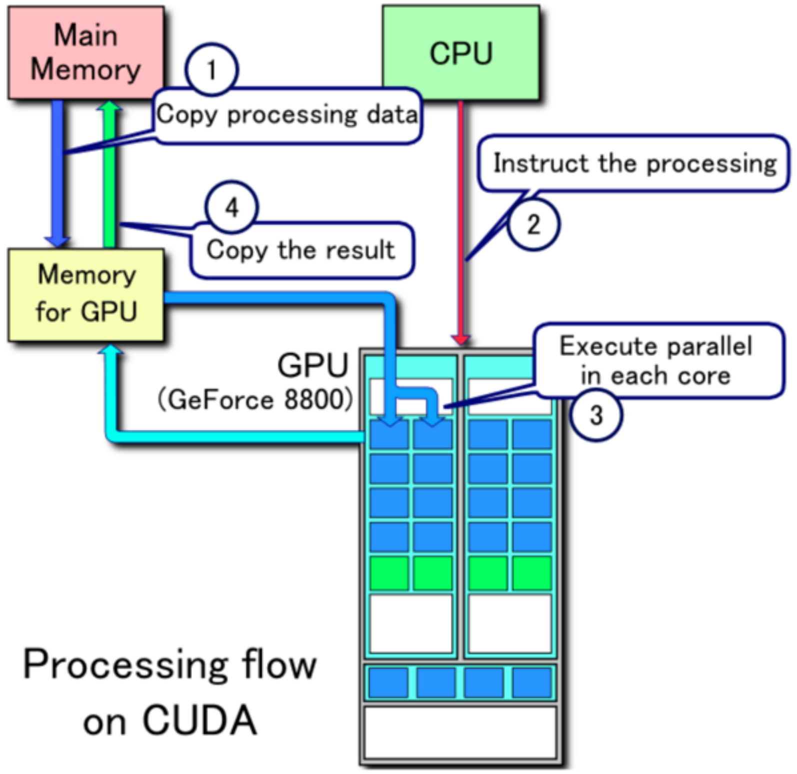
Programming

A GPU program comprises two parts: a *host* part the runs on the CPU and one or more *kernels* that run on the GPU. Typically, the CPU portion of the program is used to set up the parameters and data for the computation, while the kernel portion performs the actual computation. In some cases, the CPU portion may comprise a parallel program that performs message passing operations using MPI.


How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

Programming

A GPU program comprises two parts: a *host* part the runs on the CPU and one or more *kernels* that run on the GPU. Typically, the CPU portion of the program is used to set up the parameters and data for the computation, while the kernel portion performs the actual computation. In some cases, the CPU portion may comprise a parallel program that performs message passing operations using MPI.



GPU programming languages

CUDA
Initially released by Nvidia in 2007, CUDA (Compute Unified Device Architecture) is the dominant proprietary framework today. Developers can call CUDA from programming languages such as C, C++, Fortran, or Python without any skills in graphics programming.

OpenCL
Initially released by the Khronos Group in 2009, OpenCL(Open Computing Language) is the most popular open, royalty-free standard for cross-platform, parallel programming.

OpenCL has so far been implemented by Altera, AMD, Apple, ARM, Creative, IBM, Imagination, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS, and it supports all popular operating systems across all major platforms, making it extremely versatile. OpenCL defines a C-like language for writing programs, but third-party APIs exist for other programming languages and platforms such as Python or Java.

OpenACC

OpenACC(*open accelerators*) is the youngest programming standard for parallel computing. It was initially released in 2015 by a group of companies comprising Cray, CAPS, Nvidia, and PGI (the Portland Group) to simplify parallel programming of heterogeneous CPU/GPU systems. OpenACC has a syntax like OpenMP.

CUDA

“Compute Unified Device Architecture”

First language to allow general-purpose programming for GPUs

- preceded by shader languages Promoted by NVIDIA for their GPUs

Not supported by any other accelerator

- though commercial CUDA-to-x86/64 compilers exist

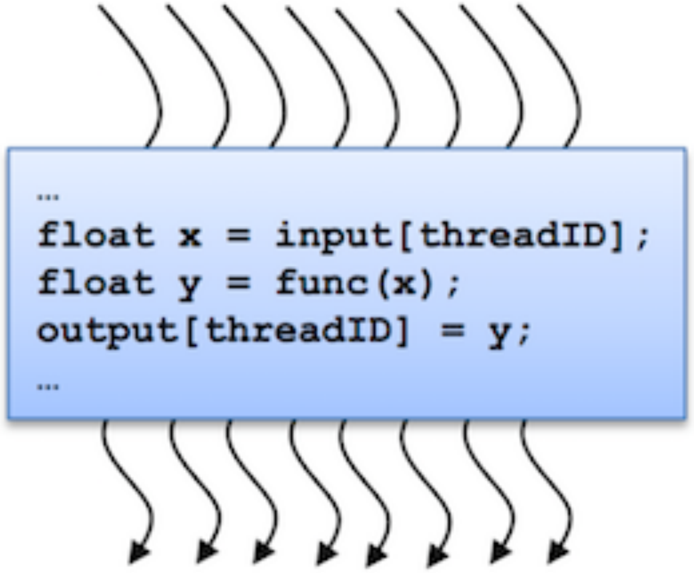
Vocabulary:

- a *CUDA kernel* is a GPU function launched by the host and executed on the device,
- the GPU and its memory are called the *device*,
- the CPU and its memory are called the *host*.

CUDA

A kernel is executed in parallel by an array of threads:

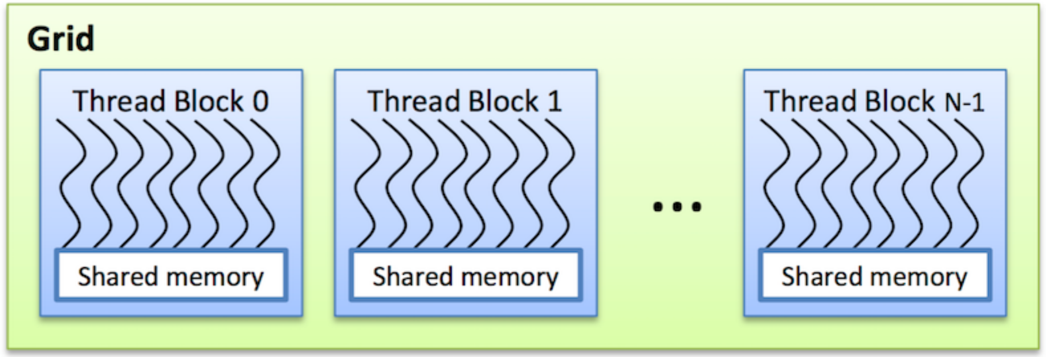
- All threads run the same code.
- Each thread has an ID that it uses to compute memory addresses and make control decisions.



```
...
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...
```

Threads are arranged as a grid of thread blocks:

- Different kernels can have different grid/block configuration
- Threads from the same block have access to a shared memory and their execution can be synchronized

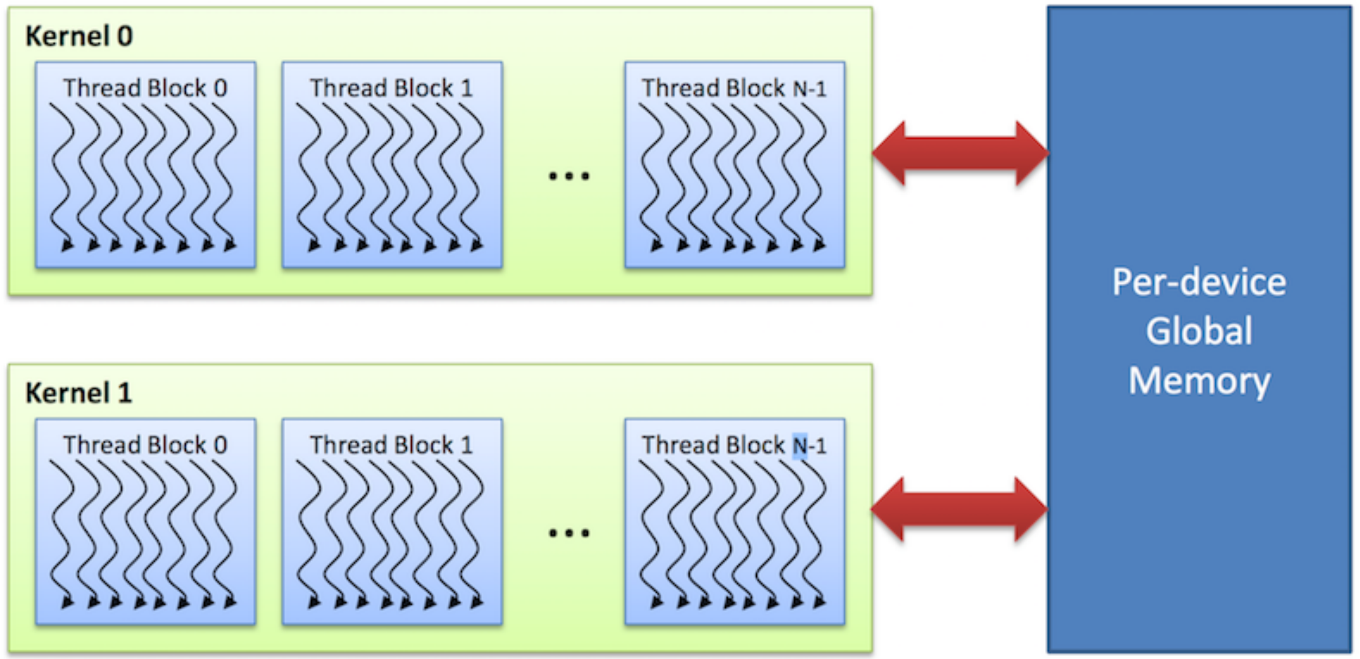


Memory Hierarchy

The CPU and GPU have separate *memory spaces*. This means that data that is processed by the GPU must be moved from the CPU to the GPU before the computation starts, and the results of the computation must be moved back to the CPU once processing has completed.

Global memory: accessible to all threads as well as the host (CPU)

- Global memory is allocated and deallocated by the host
- Used to initialize the data that the GPU will work on

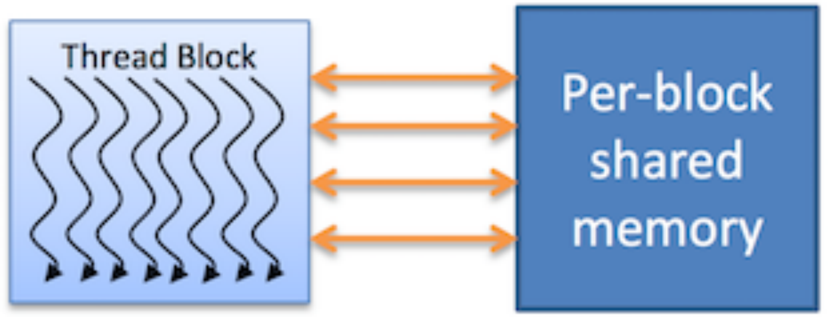


Memory Hierarchy

The CPU and GPU have separate *memory spaces*. This means that data that is processed by the GPU must be moved from the CPU to the GPU before the computation starts, and the results of the computation must be moved back to the CPU once processing has completed.

Shared memory: each *thread block* has its own shared memory

- Accessible only by threads within the block
- Much faster than local or global memory
- Requires special handling to get maximum performance
- Only exists for the lifetime of the block

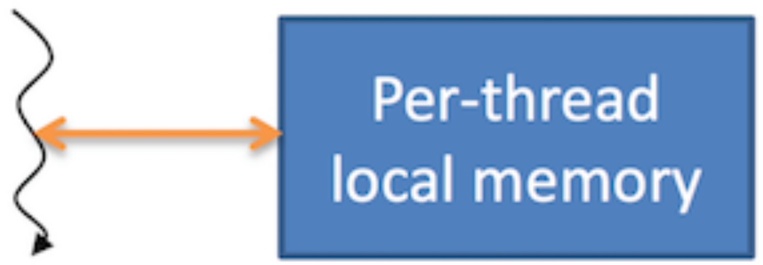


Memory Hierarchy

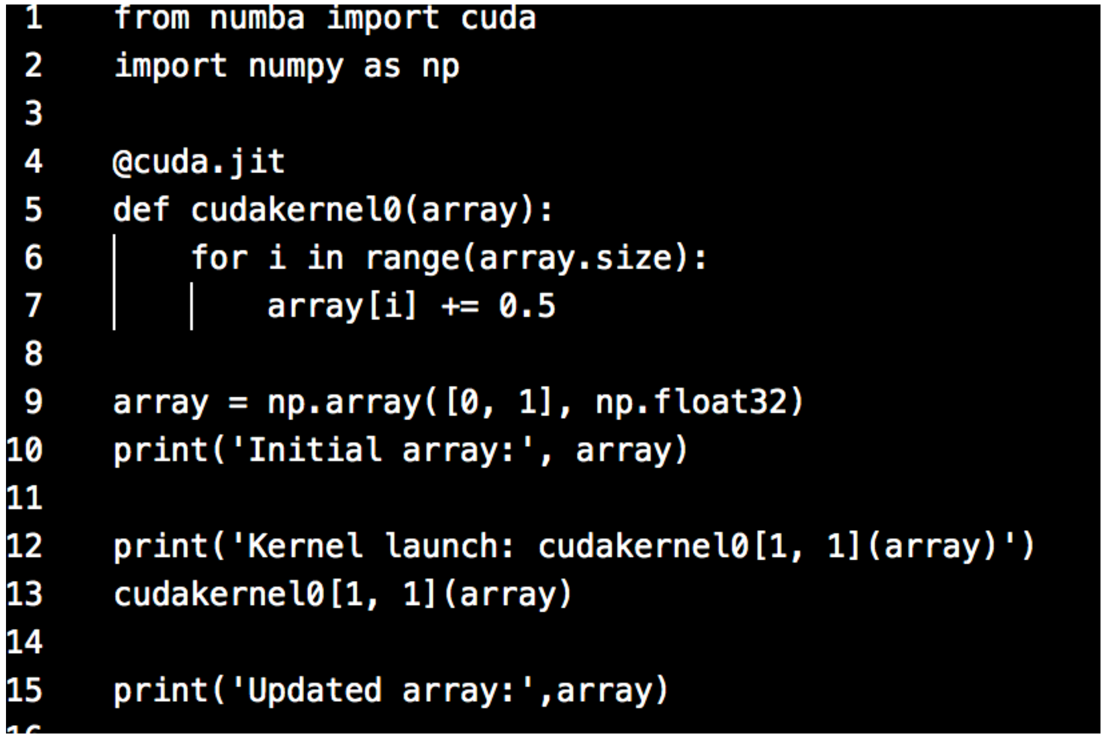
The CPU and GPU have separate *memory spaces*. This means that data that is processed by the GPU must be moved from the CPU to the GPU before the computation starts, and the results of the computation must be moved back to the CPU once processing has completed.

Local memory: each *thread* has its own private local memory

- Only exists for the lifetime of the thread
- Generally handled automatically by the compiler



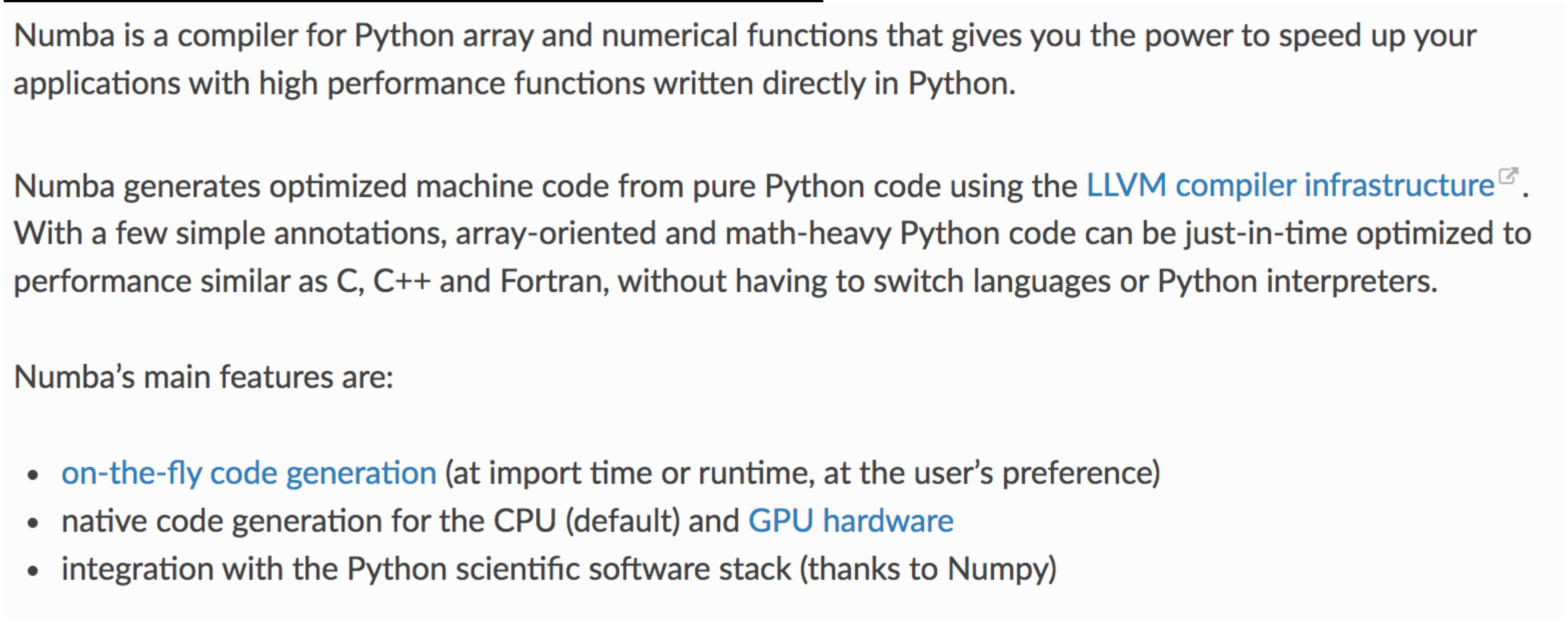
CUDA Example



```
from numba import cuda
import numpy as np

@cuda.jit
def cudakernel0(array):
    for i in range(array.size):
        array[i] += 0.5

array = np.array([0, 1], np.float32)
print('Initial array:', array)

print('Kernel launch: cudakernel0[1, 1](array)')
cudakernel0[1, 1](array)

print('Updated array:',array)
```

CUDA Example

```python
from numba import cuda
```



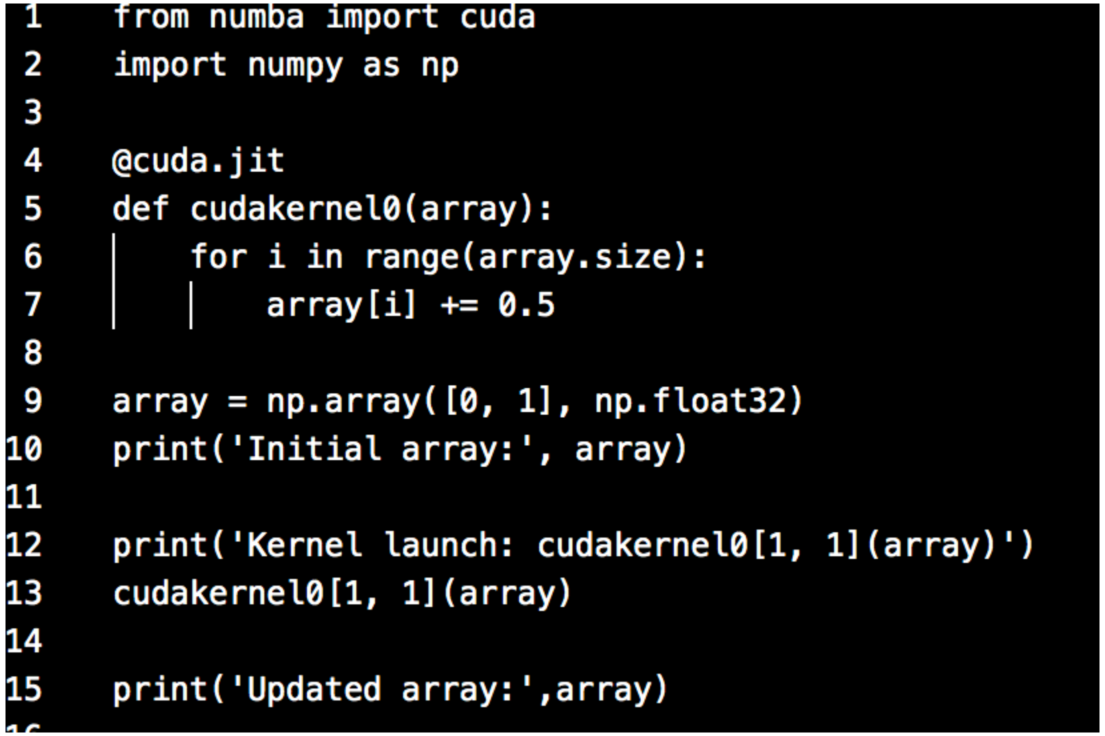
Numba is a compiler for Python array and numerical functions that gives you the power to speed up your applications with high performance functions written directly in Python.

Numba generates optimized machine code from pure Python code using the LLVM compiler infrastructure. With a few simple annotations, array-oriented and math-heavy Python code can be just-in-time optimized to performance similar as C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba's main features are:

- on-the-fly code generation (at import time or runtime, at the user's preference)
- native code generation for the CPU (default) and GPU hardware
- integration with the Python scientific software stack (thanks to Numpy)

CUDA Example



```python
from numba import cuda
import numpy as np

@cuda.jit
def cudakernel0(array):
    for i in range(array.size):
        array[i] += 0.5

array = np.array([0, 1], np.float32)
print('Initial array:', array)

print('Kernel launch: cudakernel0[1, 1](array)')
cudakernel0[1, 1](array)

print('Updated array:',array)
```

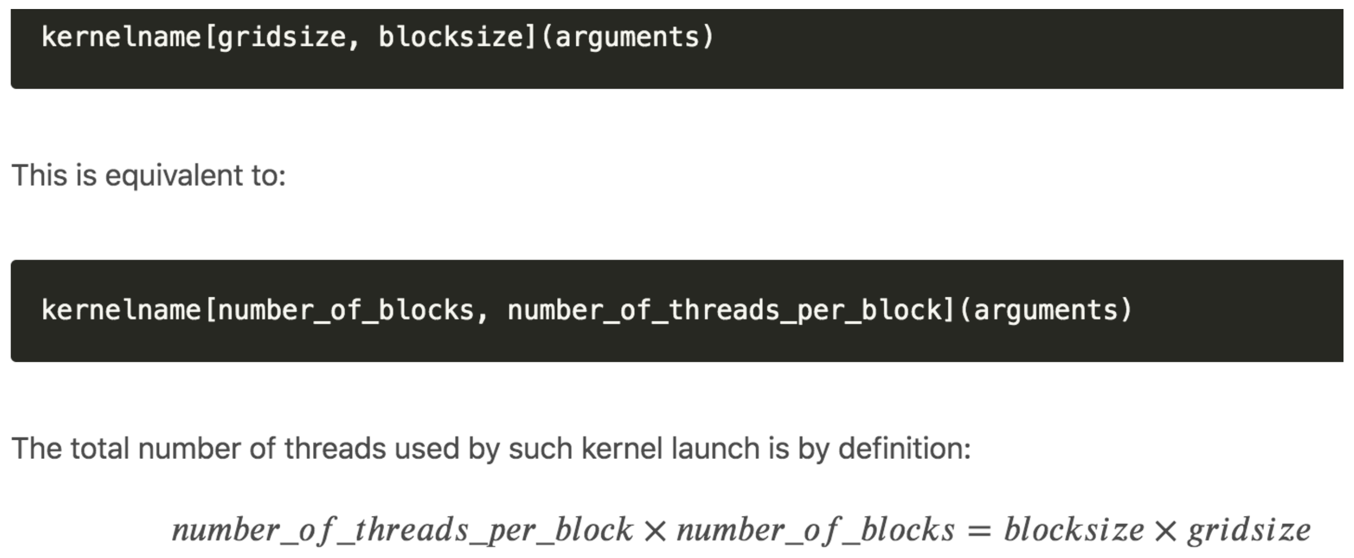
```
Initial array: [0. 1.]
Kernel launch: cudakernel0[1, 1](array)
Updated array: [0.5 1.5]
```

CUDA Example

On the GPU, the computations are executed in separate blocks, and each of these blocks uses many threads. The set of all the blocks is called a grid. Therefore, the grid size is equal to the number of blocks used during a computation.

Furthermore, each block uses the same number of threads. We call this number the block size.

To launch a kernel over multiple threads, we use the following syntax:



```
kernelname[gridsize, blocksize](arguments)
```

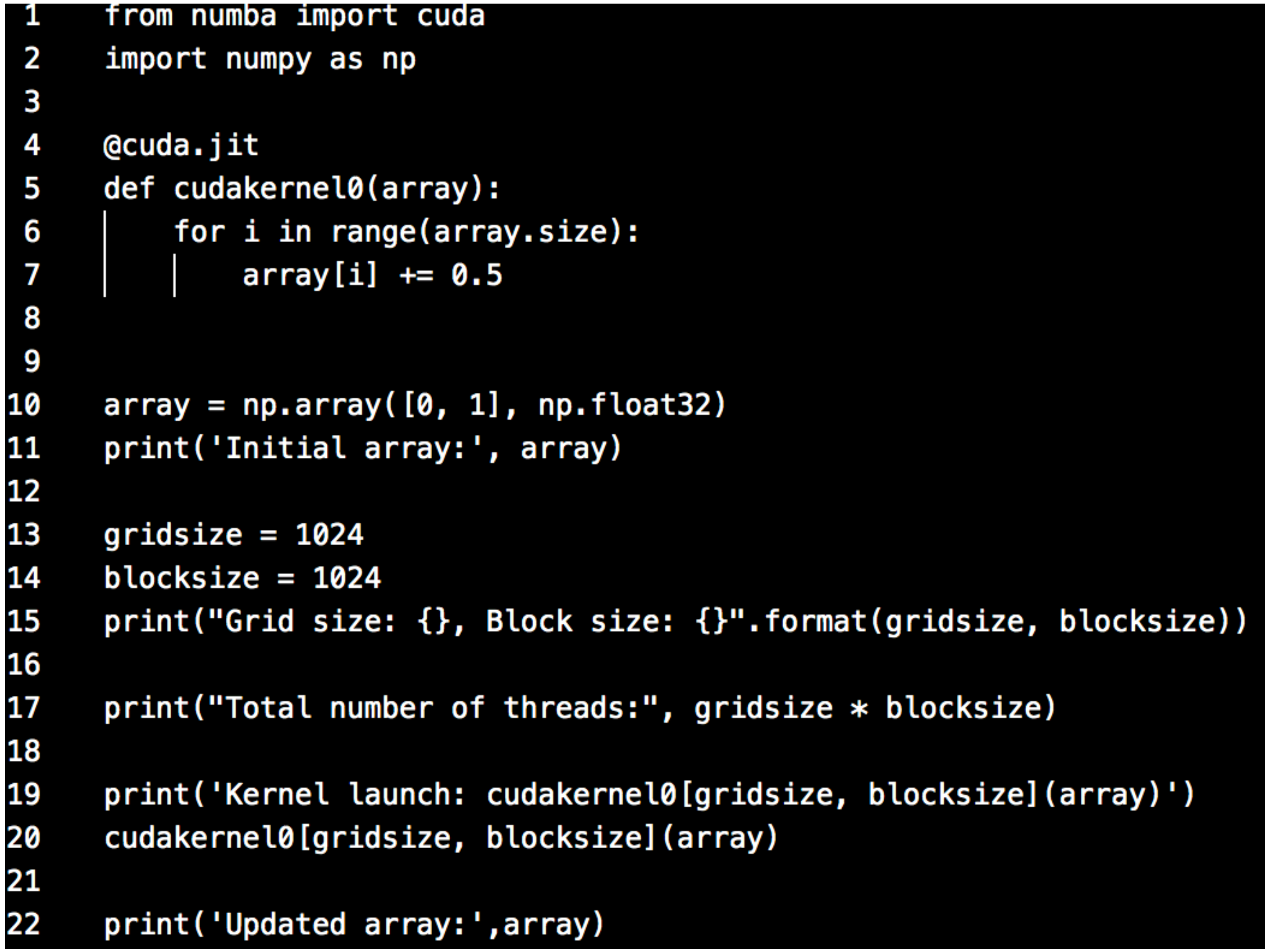
This is equivalent to:

```
kernelname[number_of_blocks, number_of_threads_per_block](arguments)
```

The total number of threads used by such kernel launch is by definition:

$$number\_of\_threads\_per\_block \times number\_of\_blocks = blocksize \times gridsize$$

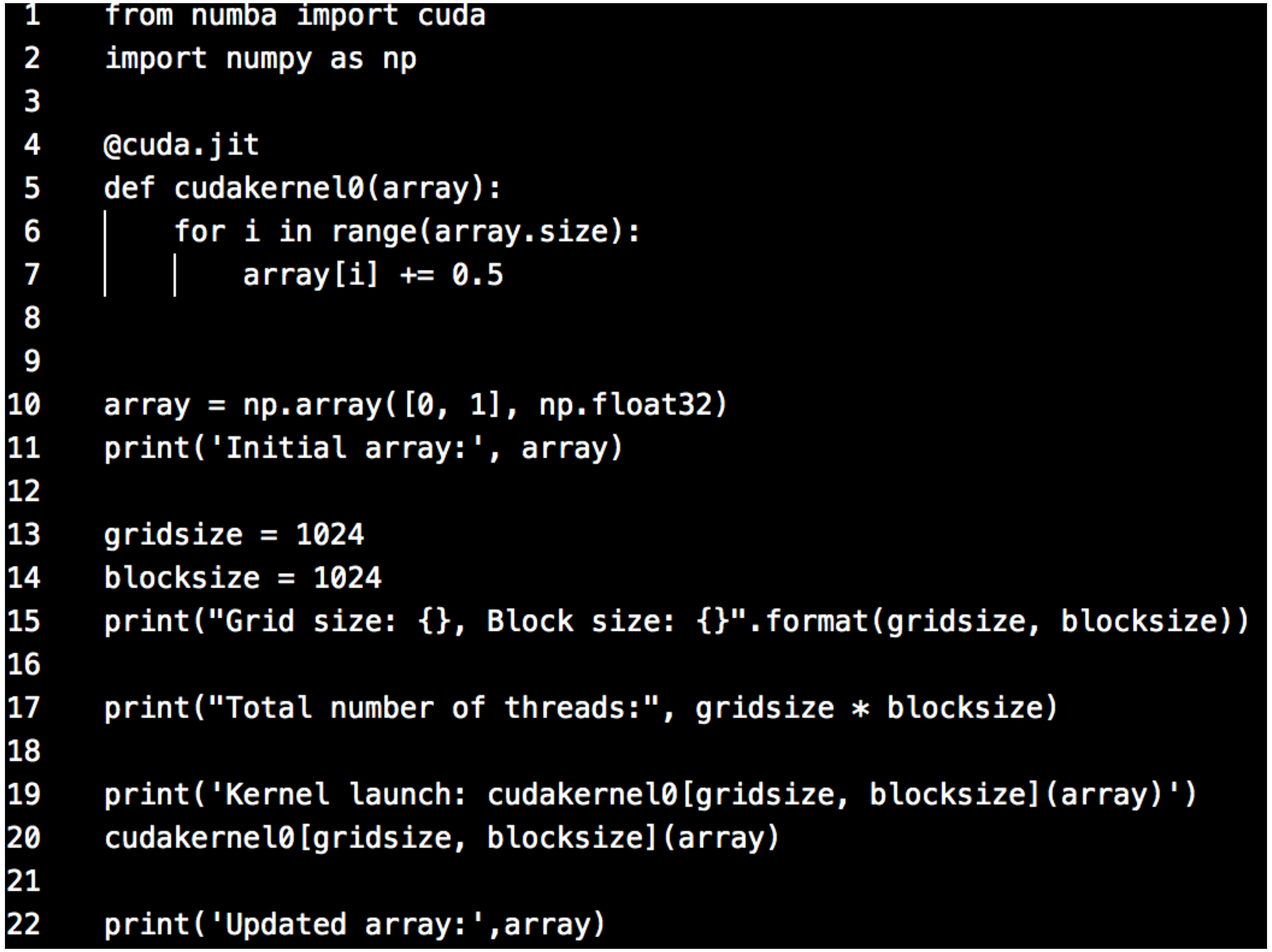
CUDA Example



```python
from numba import cuda
import numpy as np

@cuda.jit
def cudakernel0(array):
    for i in range(array.size):
        array[i] += 0.5


array = np.array([0, 1], np.float32)
print('Initial array:', array)

gridsize = 1024
blocksize = 1024
print("Grid size: {}, Block size: {}".format(gridsize, blocksize))

print("Total number of threads:", gridsize * blocksize)

print('Kernel launch: cudakernel0[gridsize, blocksize](array)')
cudakernel0[gridsize, blocksize](array)

print('Updated array:',array)
```

```
Initial array: [0. 1.]
Grid size: 1024, Block size: 1024
Total number of threads: 1048576
Kernel launch: cudakernel0[gridsize, blocksize](array)
Updated array: [56.5 56. ]
```

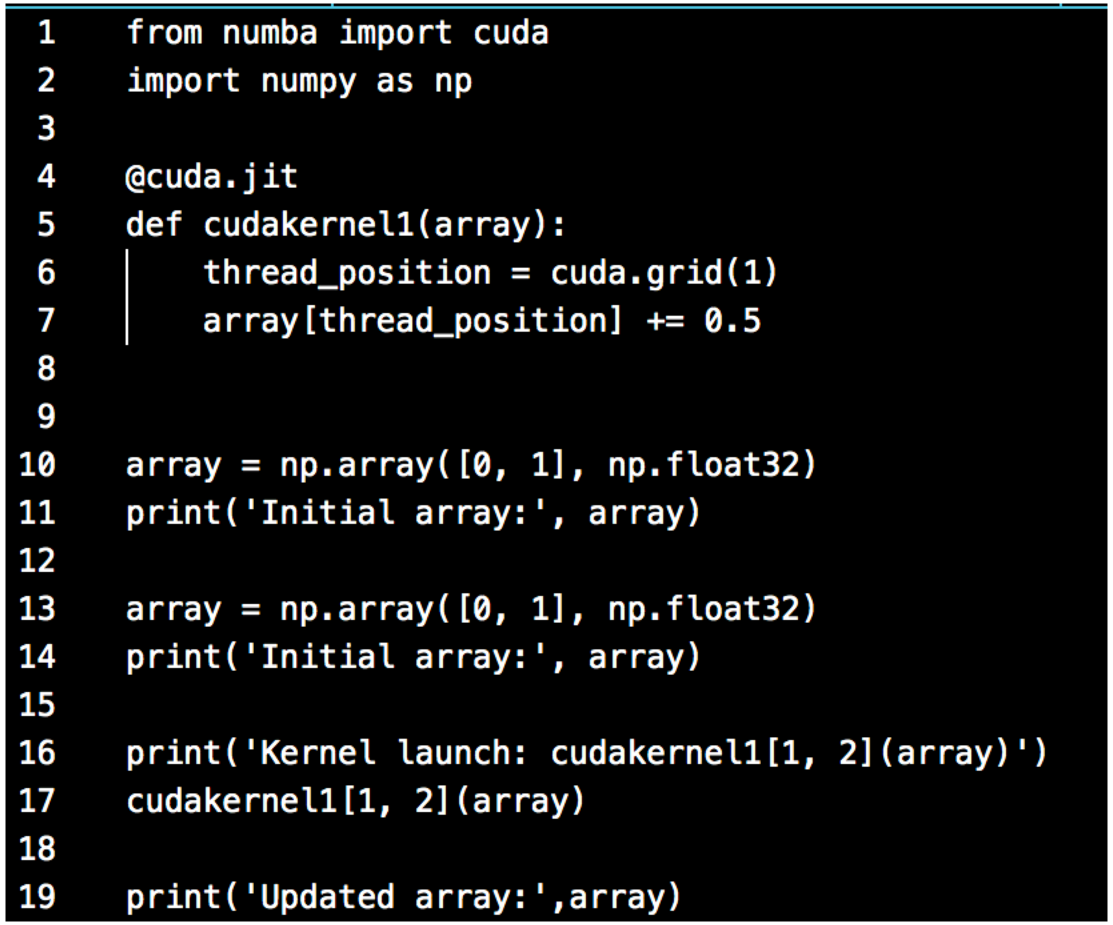
CUDA Example



```python
from numba import cuda
import numpy as np

@cuda.jit
def cudakernel0(array):
    for i in range(array.size):
        array[i] += 0.5


array = np.array([0, 1], np.float32)
print('Initial array:', array)

gridsize = 1024
blocksize = 1024
print("Grid size: {}, Block size: {}".format(gridsize, blocksize))

print("Total number of threads:", gridsize * blocksize)

print('Kernel launch: cudakernel0[gridsize, blocksize](array)')
cudakernel0[gridsize, blocksize](array)

print('Updated array:',array)
```

```
Initial array: [0. 1.]
Grid size: 1024, Block size: 1024
Total number of threads: 1048576
Kernel launch: cudakernel0[gridsize, blocksize](array)
Updated array: [56.5 56. ]
```

CUDA Example



```python
from numba import cuda
import numpy as np

@cuda.jit
def cudakernel1(array):
    thread_position = cuda.grid(1)
    array[thread_position] += 0.5


array = np.array([0, 1], np.float32)
print('Initial array:', array)

array = np.array([0, 1], np.float32)
print('Initial array:', array)

print('Kernel launch: cudakernel1[1, 2](array)')
cudakernel1[1, 2](array)

print('Updated array:',array)
```

```
Initial array: [0. 1.]
Kernel launch: cudakernel1[1, 2](array)
Updated array: [0.5 1.5]
```

CUDA Example

Note that if we launch this kernel with a grid of only one thread, only the first cell of the array will be updated:

```
array = np.array([0, 1], np.float32)
print('Initial array:', array)

print('Kernel launch: cudakernel1[1, 1](array)')
cudakernel1[1, 1](array)

print('Updated array:',array)
```

```
Initial array: [0. 1.]
Kernel launch: cudakernel1[1, 1](array)
Updated array: [0.5 1. ]
```

CUDA Example

Let's test this kernel with an array of size 2^{20}. We therefore need a total number of threads of $2^{20} = 1024 \times 1024$.

```
array = np.zeros(1024 * 1024, np.float32)
print('Initial array:', array)

print('Kernel launch: cudakernel1[1024, 1024](array)')
cudakernel1[1024, 1024](array)

print('Updated array:', array)

# Since it is a huge array, let's check that the result is correct:
print('The result is correct:', np.all(array == np.zeros(1024 * 1024, np.float32) + 0.5))
```

```
Initial array: [0. 0. 0. ... 0. 0. 0.]
Kernel launch: cudakernel1[1024, 1024](array)
Updated array: [0.5 0.5 0.5 ... 0.5 0.5 0.5]
The result is correct: True
```

> Note that there are some limitations on the number of threads depending on the hardware. For recent GPUs, the maximum number of threads by block is usually equal to 1024.