

# PMC Lecture 12

Gianmaria Romano

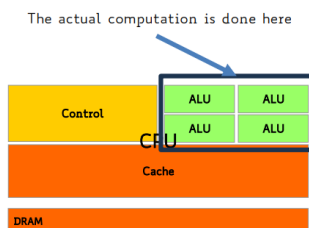
25 November 2025

## Chapter 6

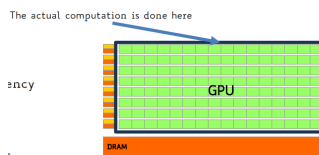
# Compute Unified Device Architecture

### 6.1 The main aspects of the CUDA interface

Due to higher clock frequency and complex control logic, CPUs are designed to be latency-oriented as they implement more complex hardware structures, such as caches and branch predictors, that allow to perform computations in the ALU with minimal delay.

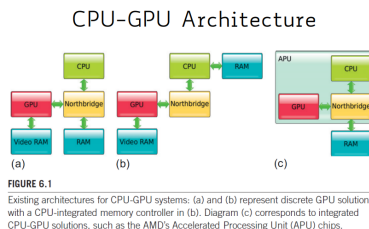


On the other hand, GPUs are designed to be throughput-oriented as they implement simpler control units to make more space for a large number of execution units that can be used in parallel.



For this reasons, CPUs are more suitable for sequential tasks, whereas GPUs

are convenient for parallel tasks.

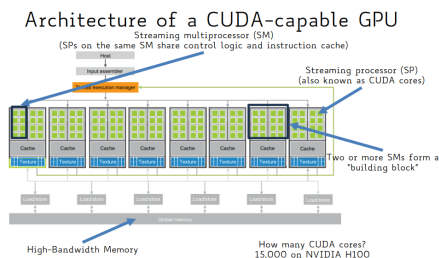


**N.B.:** Since GPU and host memories are disjoint, communication will always need some data transfers, although recent versions of CUDA make use of a unified virtual memory that uses common addresses.

## 6.2 The CUDA architecture

CUDA is a parallel programming platform that provides two API layers for performing parallel computations.

From a high-level perspective, the CUDA architecture consists of several building blocks containing various streaming processors, which are grouped in streaming multiprocessors that share access to a high-bandwidth memory space.



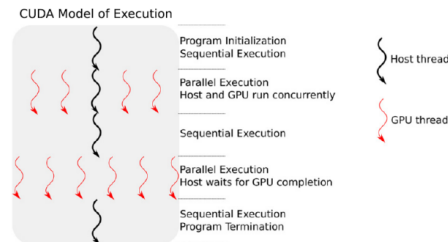
In particular, the CUDA architecture sees the GPU as a co-processor that, using its global memory, can run many threads in parallel to easily perform some tasks.

## 6.3 The structure of a CUDA program

Generally speaking, the first step of a CUDA program consists in allocating the necessary GPU memory for computations and transferring the required data from the host to the GPU memory.

Then, the program runs a CUDA kernel, which is a function that is executed on

the GPU, and, when done, it copies the output result from the GPU memory to the host memory.



**N.B.:** I/O operations are typically left to the host.

**Example:** Writing a CUDA program requires writing a kernel, which gets executed by all the threads, and specifying the grid/block organization for the threads.

For instance, the following code implements a function that prints "Hello, World!" in CUDA:

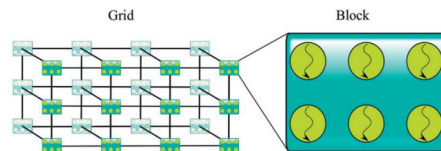
```

1  #include <stdio.h>
2  #include <cuda.h>
3
4  // The global decorator indicates that the function must be executed on the GPU.
5  // In particular, since kernels are declared as void, the computed result should be copied
6  __global__ void hello() {
7      printf("Hello, world!");
8  }
9
10 int main() {
11     hello<<1,10>>>(); // Run hello() using one block containing 10 threads.
12     cudaDeviceSynchronize(); // Block until the CUDA kernel terminates.
13     return 1;
14 }

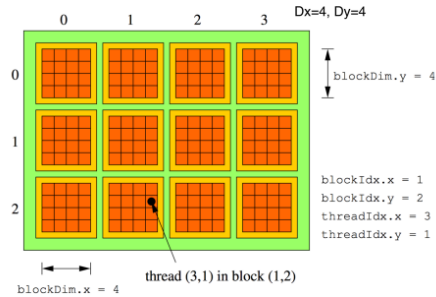
```

### 6.3.1 Thread organization in a CUDA program

In the context of CUDA programming, threads are organized in blocks, which are arranged in a grid that, depending on the input datatype, can be organized across one, two or three dimensions.



In particular, each thread is able to determine its (global) position within a grid using some intrinsic variables providing size or position information.



### 6.3.2 Function decorators for CUDA programs

A decorator indicates whether a function should be compiled to be performed on the host or on the GPU.

- **Global:** this decorator can be called by the host or by the GPU to specify where the function should be executed.
- **Device:** this decorator indicates that the function runs on the GPU and can only be called from the GPU.
- **Host:** this decorator indicates that the function can only run on the host. Generally speaking, however, this decorator is only used with the device decorator to indicate function that can run on both the host and the GPU.

## 6.4 Thread scheduling in CUDA programs

When dealing with CUDA programs, each thread runs on a specific streaming processor within a single streaming multiprocessor, which hosts all the threads contained a given block.

In particular, a streaming multiprocessor will (concurrently) run a block on a single instruction and, when done, it will start running a new block.

### 6.4.1 Scheduling using wraps

Threads within a block are normally grouped in contiguous warps, which typically contain 32 threads.

However, since threads within the same warp must always execute the same assembly task, a warp might experience a divergence when, typically upon a selective branch, some threads within the same warp try to perform different tasks.

In this cases, the program must sequentially analyse each possible branch while stalling the threads that do not enter the current path, often leading to a lower usage of the resources.

**Example:** A warp divergence can happen if the code contains a branch based

on whether the current thread's ID is an odd number or an even number.

