PMC Lecture 12

Gianmaria Romano

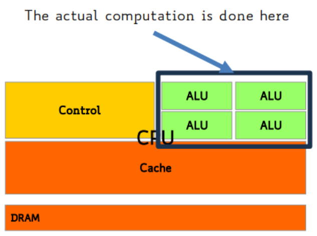
25 November 2025

Chapter 6

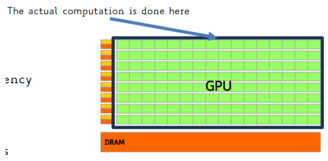
Compute Unified Device Architecture

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 hands, 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.

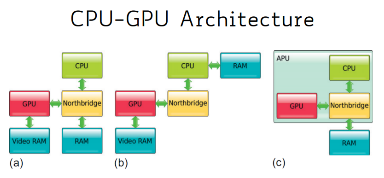


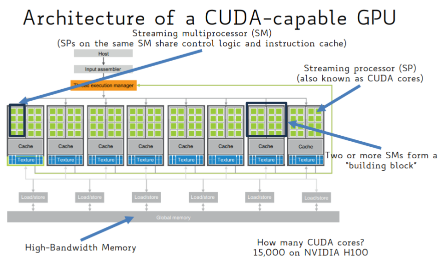
FIGURE 6.1

Existing architectures for CPU-GPU systems: (a) and (b) represent discrete GPU solutions, with a CPU-integrated memory controller in (b). Diagram (c) corresponds to integrated CPU-GPU solutions, such as the AMD's Accelerated Processing Unit (APU) chips.

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 APIs 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.

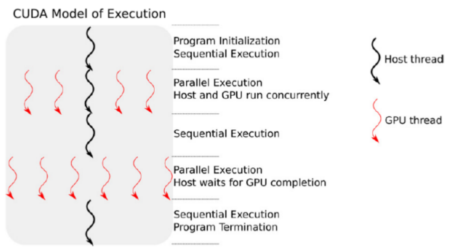

Architecture of a CUDA-capable GPU
Streaming multiprocessor (SM)
(SPs on the same SM share control logic and instruction cache)
Host
Input assembler
Thread execution manager
Streaming processor (SP)
(also known as CUDA cores)
Cache
Texture
Two or more SMs form a "building block"
Load/store
Global memory
High-Bandwidth Memory
How many CUDA cores?
15,000 on NVIDIA H100

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.

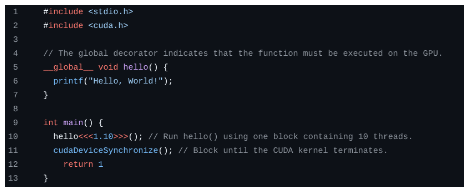
6.3 The structure of a CUDA program

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.



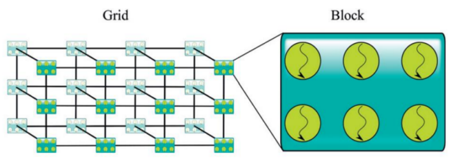
N.B.: I/O operations are typically left to the host.
Example: Writing a CUDA program requires writing a kernel, which will be executed by all the threads, and specifying the grid/block organization for the threads.
For instance, the following code implements a function that prints "Hello, World!" in CUDA:



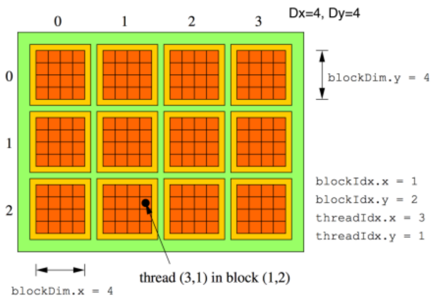
```
#include <stdio.h>
#include <cuda.h>

// The global decorator indicates that the function must be executed on the GPU.
__global__ void hello() {
  printf("Hello, World!");
}

int main() {
  hello<<<1.10>>>(); // Run hello() using one block containing 10 threads.
  cudaDeviceSynchronize(); // Block until the CUDA kernel terminates.
    return 1
}
```

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 over one, two or three dimensions.



In particular, each thread can determine its position within a grid using some intrinsic variables, which provide the size of each block/grid across each dimensions or, in a more specific way, the position of a thread in a block or the position of a block in the grid: these variables also allow the thread to recover its global position within the entire thread group.



Keep in mind, however, that the maximum size of a block/grid is limited by the capability of the GPU's architecture, which defines the features and instructions that are supported by the device.

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 the the function can only run on the host, although this decorator is mostly used when device functions are presented as well.

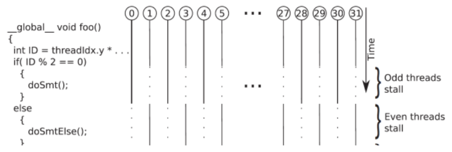
6.4 Thread scheduling in CUDA programs

When dealing with CUDA programs, each thread runs on a specific streaming processor within a single streaming multiprocessor, which hosts all the threads within a given block.
In particular, when a thread block is fully executed, the multiprocessor will run the next one, although, generally speaking, not all threads need to run concurrently.

6.4.1 Scheduling using warps

Threads within a block are normally grouped in warps, typically containing 32 threads organized contiguously.
However, since threads within the same warp must always execute the same assembly instruction, a warp might experience a divergence when, typically upon a branching code, threads within the same block try to perform different tasks.
In these cases, the program sequentially must analyse each possible branch,

stalling the threads that are not performing that branch.



```
__global__ void foo()
{
 int ID = threadIdx.y * . . .
 if( ID % 2 == 0)
   {
     doSmt();
   }
 else
   {
     doSmtElse();
   }
```