Arquitecturas de Alto Desempenho

CUDA Programming

António Rui Borges

Summary

- *CUDA*
- *CUDA programming model*
- *Suggested reading*

CUDA - 1

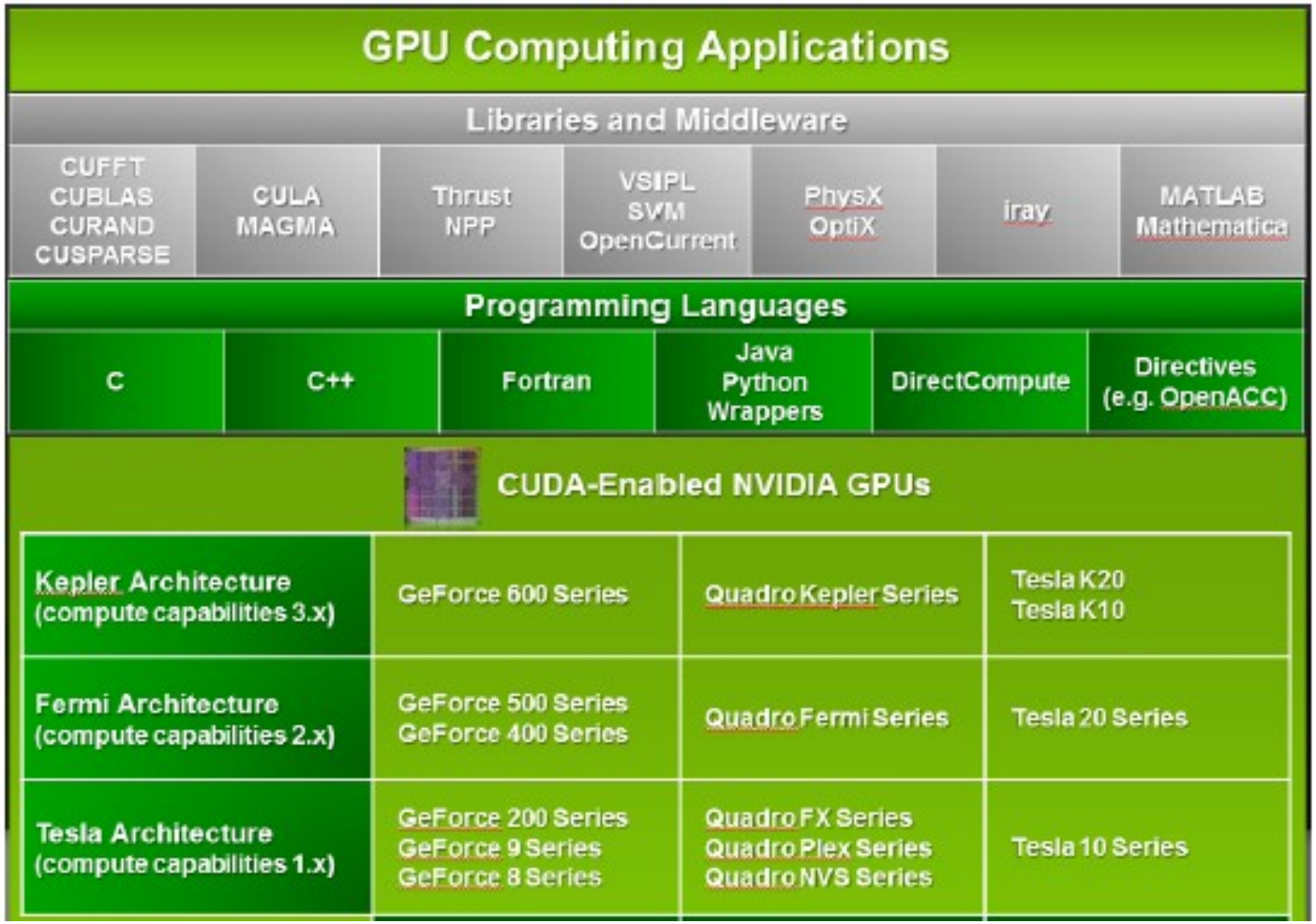
CUDA is a general purpose parallel computing platform and programming model that leverages the parallel computer engine in NVIDIA GPUs to solve many computation-intensive problems in a more efficient way.

The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces and extensions to industry-standard programming languages as C, C++, Fortran, Java and Python.

CUDA C is an extension of standard ANSI C with a handful of language extensions to enable heterogenous programming and also straightforward APIs to manage devices, memory and other tasks.

CUDA platform for heterogenous computing

Source: CUDA C Programming Guide - Nvidia



CUDA - 3

CUDA provides two API levels for managing the GPU device and organizing threads

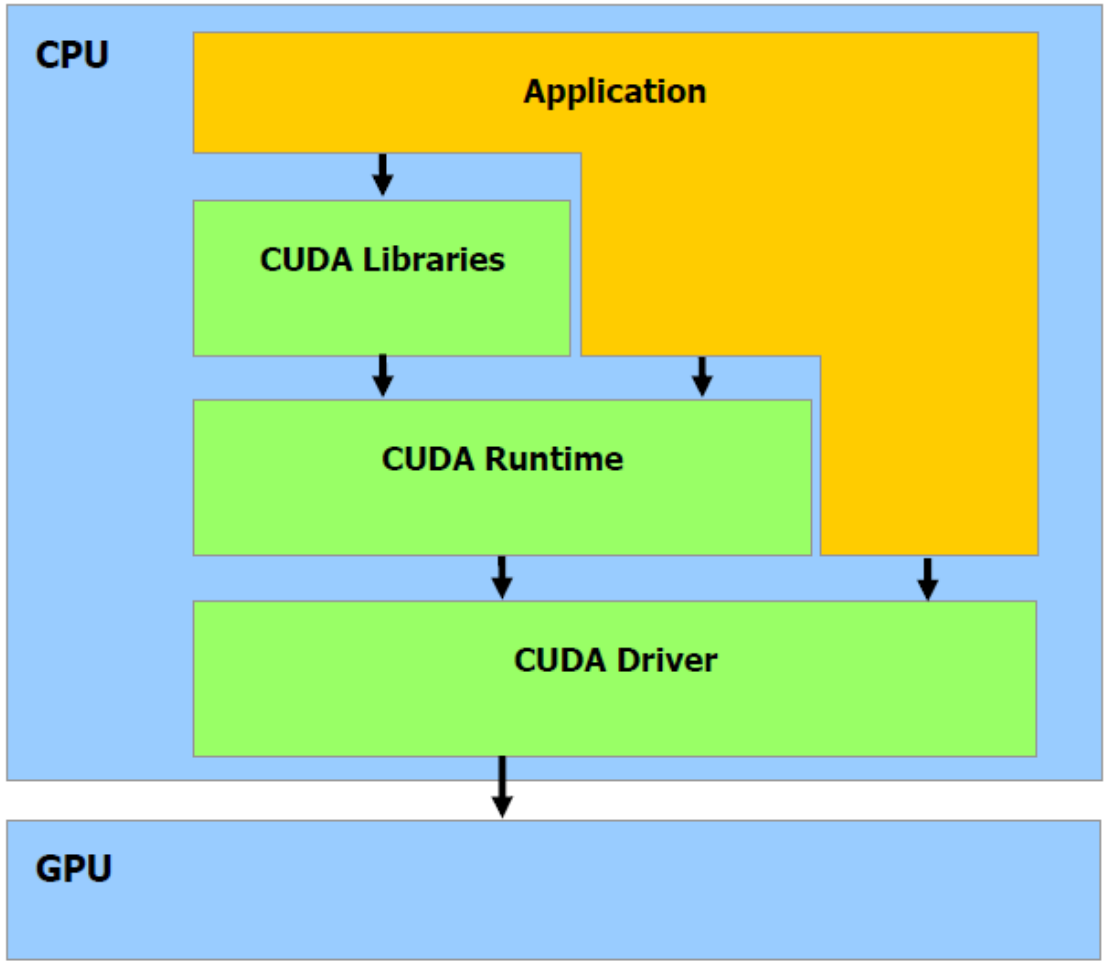
- CUDA driver API (http://docs.nvidia.com/cuda/cuda-driver-api/index.html)
- CUDA runtime API (http://docs.nvidia.com/cuda/cuda-runtime-api/index.html).

The *driver* API is a low-level API and is relatively hard to program. However, it gives the programmer more control over how the GPU device is used. The *runtime* API, on the other hand, is a high-level API implemented on top of the driver API. Each function of the runtime API is typically broken down into one or more basic operations issued to the driver API.

There is no noticeable difference between them. How programmers organize the threads and how memory is used have a much more pronounced effect on performance. They are nevertheless mutually exclusive: only one of them can be used in the source code, no mix of function calls from both is allowed.

CUDA software interface

Source: Professional CUDA C Programming



CUDA - 5

NVIDIA's CUDA `nvcc` compiler separates the device code from the host code during the compilation process. The *host code* is standard C code and is further compiled with a C compiler. The *device code* is written using CUDA C extended with keywords for labeling data parallel functions, called *kernels*. The device code is further compiled by `nvcc`. During the link stage, CUDA runtime or driver libraries are added for kernel procedure calls and explicit GPU manipulation.

The CUDA toolkit includes the compiler, math libraries and tools for debugging and optimizing the performance of applications.

CUDA - 6

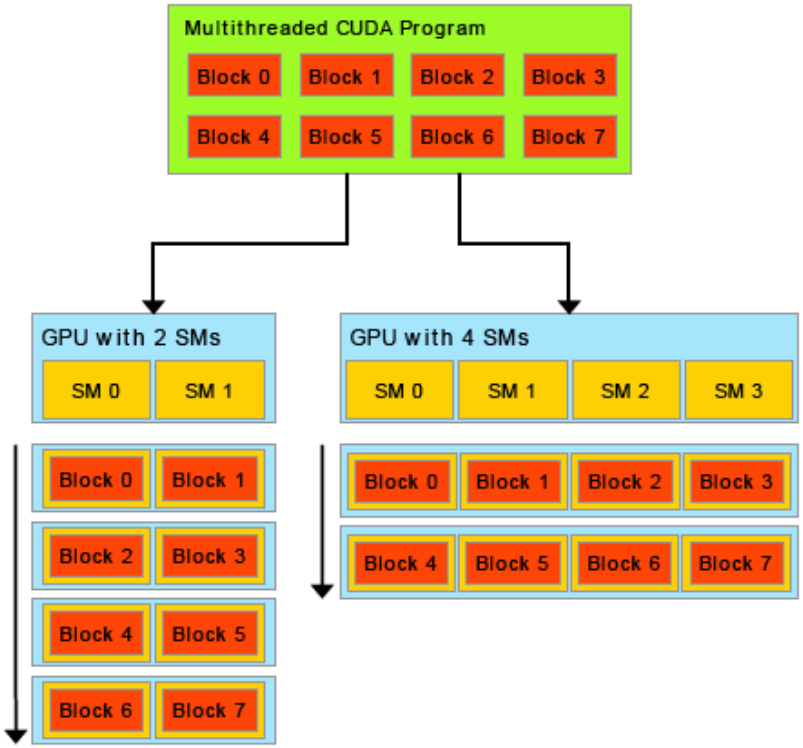
At its core, CUDA has three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

The above mentioned abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They let the programmer to partition the problem into a set of subproblems, which can be solved independently in parallel by blocks of threads, and each subproblem in turn into still finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressiveness by allowing threads to cooperate when solving each subproblem and at the same time enables automatic availability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors and only the runtime system needs to know the physical multiprocessor count.

CUDA - 7

A GPU is built around an array of *streaming multiprocessors*, organized as a MIMD topology of SIMD processors. A multithreaded program is partitioned into blocks of threads that execute independently from one another, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

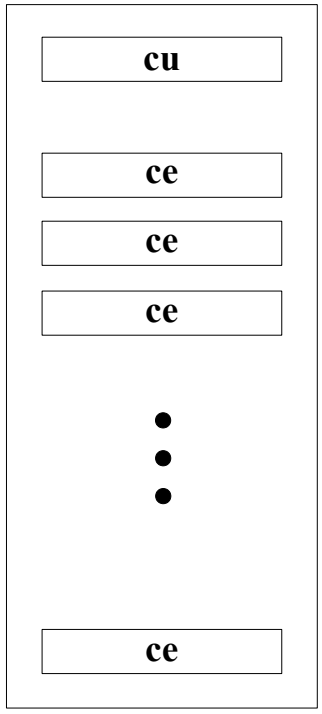


Automatic scalability

Source: CUDA C Programming Guide - Nvidia

CUDA - 8

A *streaming multiprocessor* is essentially a SIMD processor: the same instruction will be executed in parallel in multiple *computing elements* or *cores*, each having its own register bank and functional units. In this context, a *thread* may thought of as the operations carried out in sequence in each computing element.



Each *block of threads* is executed in the same streaming multiprocessor and is divided into chunks of 32 threads, called *warps*, which are processed in parallel.

CUDA - 9

A typical CUDA program structure consists of five main steps

- allocate GPU memory
- copy data from CPU memory to GPU memory
- invoke CUDA kernel to perform a program-specific computation
- copy data back from GPU memory to CPU memory
- destroy GPU memories.

Hello world

```
#include "../common/common.h"
#include <stdio.h>

/* A simple introduction to programming in CUDA. This program
 * prints "Hello World from GPU! from 10 CUDA threads running
 * on the GPU.
 */

__global__ void helloFromGPU()
{
    printf("Hello World from GPU!\n");
}

int main(int argc, char **argv)
{
    printf("Hello World from CPU!\n");

    helloFromGPU<<<1, 10>>>();
    CHECK(cudaDeviceReset());
    return 0;
}
```

CUDA programming model - 1

CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times concurrently by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the `__global__` declaration specifier. Upon calling, the number of CUDA threads that will be instantiated is specified using a `<<<...>>>` execution configuration clause which describes a grid of thread blocks running concurrently.

Each thread that executes the kernel is given a unique thread Id within the thread block it belongs to. The thread Id is accessible within the kernel through the built-in `threadIdx` variable. Each thread within a thread block executes an instance of the kernel, has its own registers and private memory, inputs data and output results.

CUDA programming model - 2

For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block*. This provides a natural way to invoke a computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread Id relate to each other in a straightforward way. For a 1-dimensional block, they are the same; for a 2-dimensional block of size (Dx, Dy), the thread Id of a thread of index (x, y) is (x + y Dx); for a 3-dimensional block of size (Dx, Dy, Dz), the thread Id of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

CUDA programming model - 3

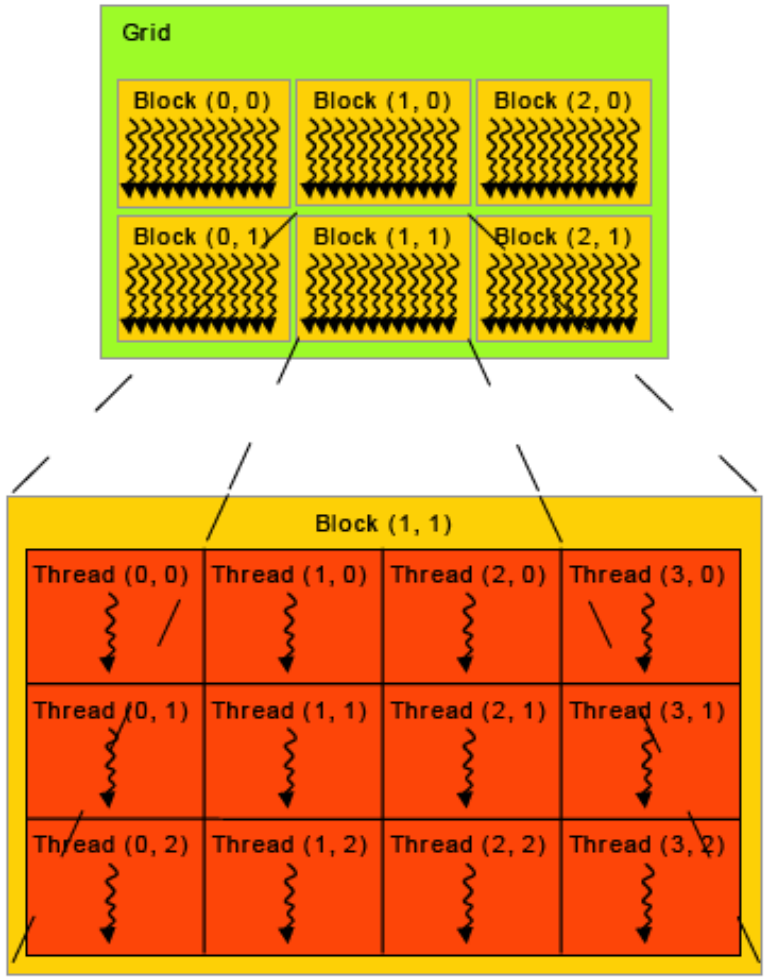
Blocks are organized into a 1-dimensional, 2-dimensional, or 3-dimensional *grid* of thread blocks. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type **`int`** or `dim3`. Each block within the grid can be identified by a 1-dimensional, 2-dimensional, or 3-dimensional index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

CUDA programming model - 4

Grid of thread blocks

Source: CUDA C Programming Guide - Nvidia



CUDA programming model - 5

Thread blocks are required to execute independently. It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled randomly across any number of cores, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L2 cache in a conventional CPU) and `__syncthreads()` is expected to be lightweight.

CUDA programming model - 6

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.

All threads have access to the same *global memory*. There are also two additional read-only memory spaces accessible by all threads: the *constant* and *texture* memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages.

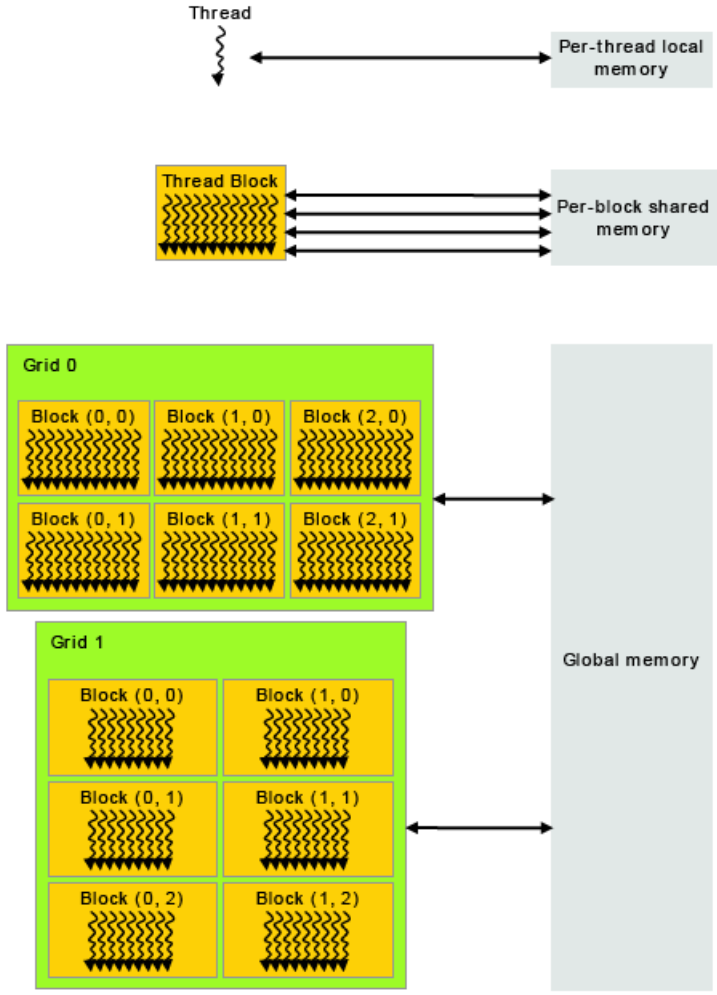
Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as *host memory* and *device memory*, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.

CUDA programming model - 7

Memory hierarchy

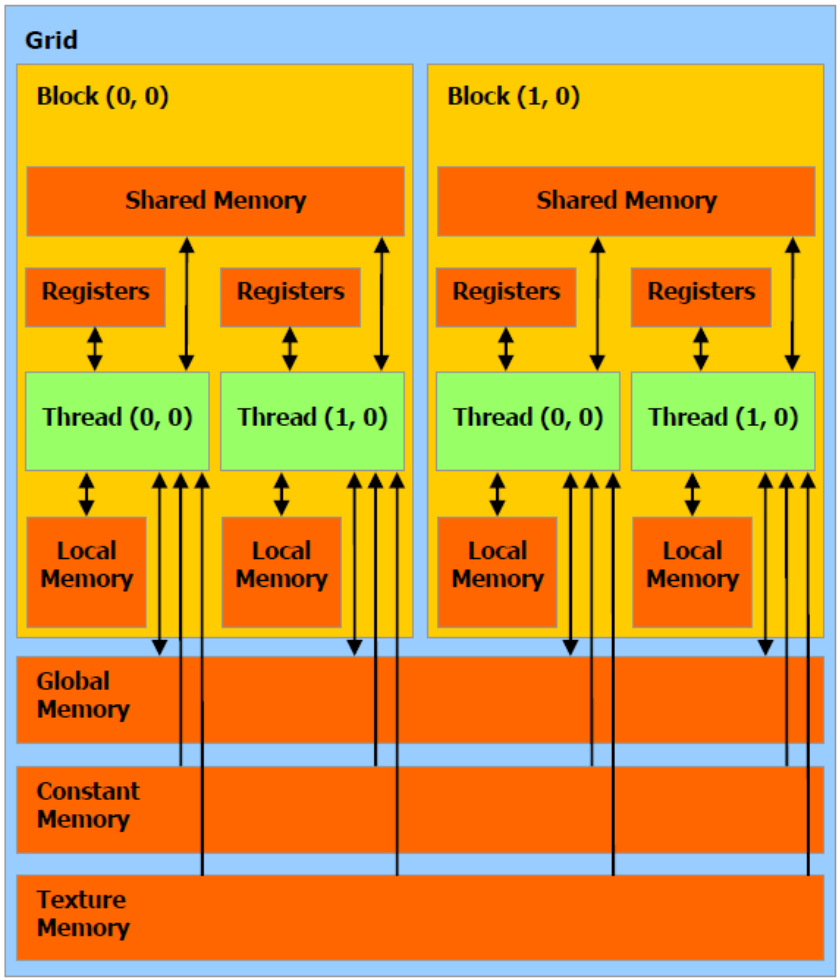
Source: CUDA C Programming Guide - Nvidia



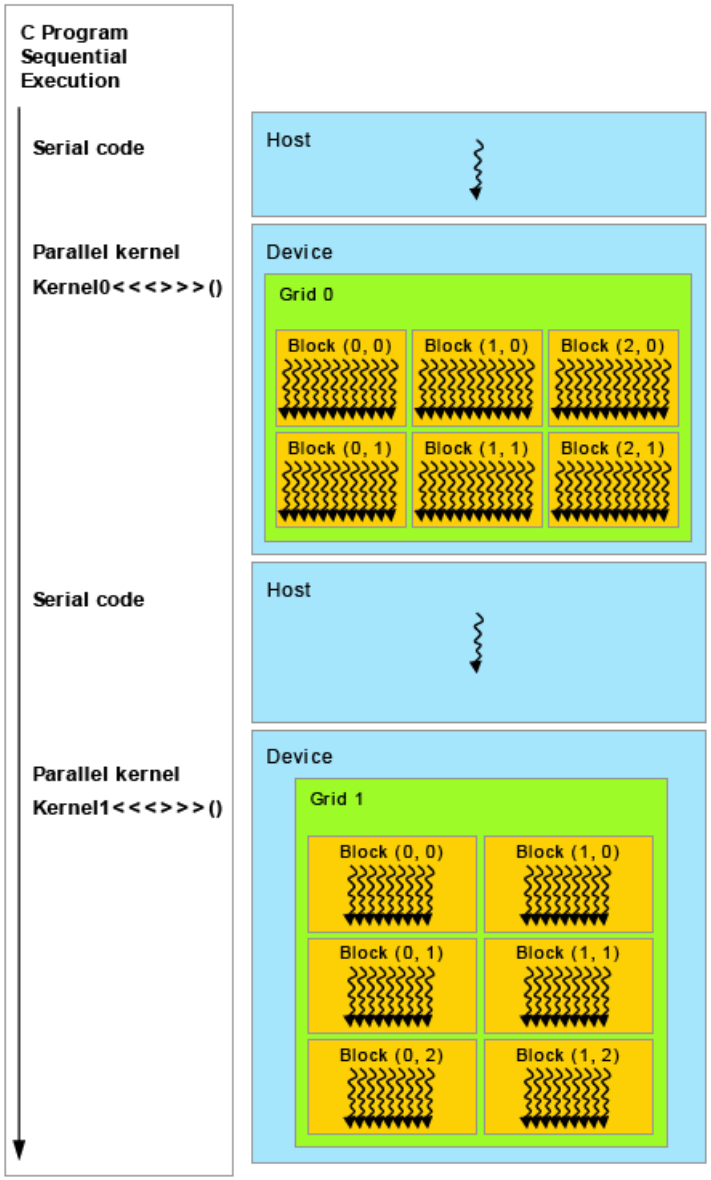
CUDA programming model - 8

Memory hierarchy

Source: CUDA C Programming Guide - Nvidia



CUDA programming model - 9



CPU – GPU interaction

Source: CUDA C Programming Guide - Nvidia

CUDA programming model – 10

Function type qualifiers

Qualifier	Execution	Callable	Observation
`__global__`	on the device	from the host	**void** return type
`__device__`	on the device	from the device	
`__host__`	on the host	from the host	may be omitted

The `__device__` and the `__host__` qualifiers can be used together, in which case the function is compiled for both the host and the device.

CUDA programming model – 11

CUDA kernels are functions with restrictions.

The following restrictions apply to all kernels

- they only access device memory
- they must have a **void** return type
- there is no support for a variable number of parameters
- there is no support for static variables
- there is no support for function pointers
- they exhibit an asynchronous behavior.

Suggested reading

- *CUDA C++ Programming Guide*, NVIDIA, 2021
- *Professional CUDA Programming*, Cheng J., Grossman M., McKercher T., John Wiley & Sons, Inc, 2014
- *Programming Massively Parallel Processors: A Hands-on Approach*, Kirk D. B., Hwu W. W., Morgan Kaufmann, 2017