Chapter 2: Brief Introduction to CUDA

In November 2006, NVIDIA® introduced CUDA® (Compute Unified Device Architecture), a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

Many Python developers can skip this chapter if they are using a high-level library interface. The concepts presented here will come in handy if you ever have to write your own kernel functions or map your algorithm efficiently into GPU architecture.

CUDA Architecture

There are typically three main steps required to execute a function (a.k.a. kernel) on a GPU in a scientific code:

1. Copy the input data from the CPU memory to the GPU memory
2. Load and execute the GPU kernel on the GPU
3. Copy the results from the GPU memory to CPU memory.

As explained in the last chapter, the architecture of the CPU and GPU are different, requiring a different programming paradigm.

CUDA Programming Model

CUDA operates on a heterogeneous computing model that involves both the CPU and GPU.

Kernels

The CUDA kernel is a global function that gets executed on GPU. The parallel portion of your applications is executed K times in parallel by K different CUDA threads, as opposed to only one time like regular sequential functions on the CPU.

Threads and Warps

In CUDA, the kernel is executed with the aid of threads. The thread is an abstract entity that represents the execution of the kernel. This represents the smallest unit of execution in CUDA and possesses a unique Thread ID.

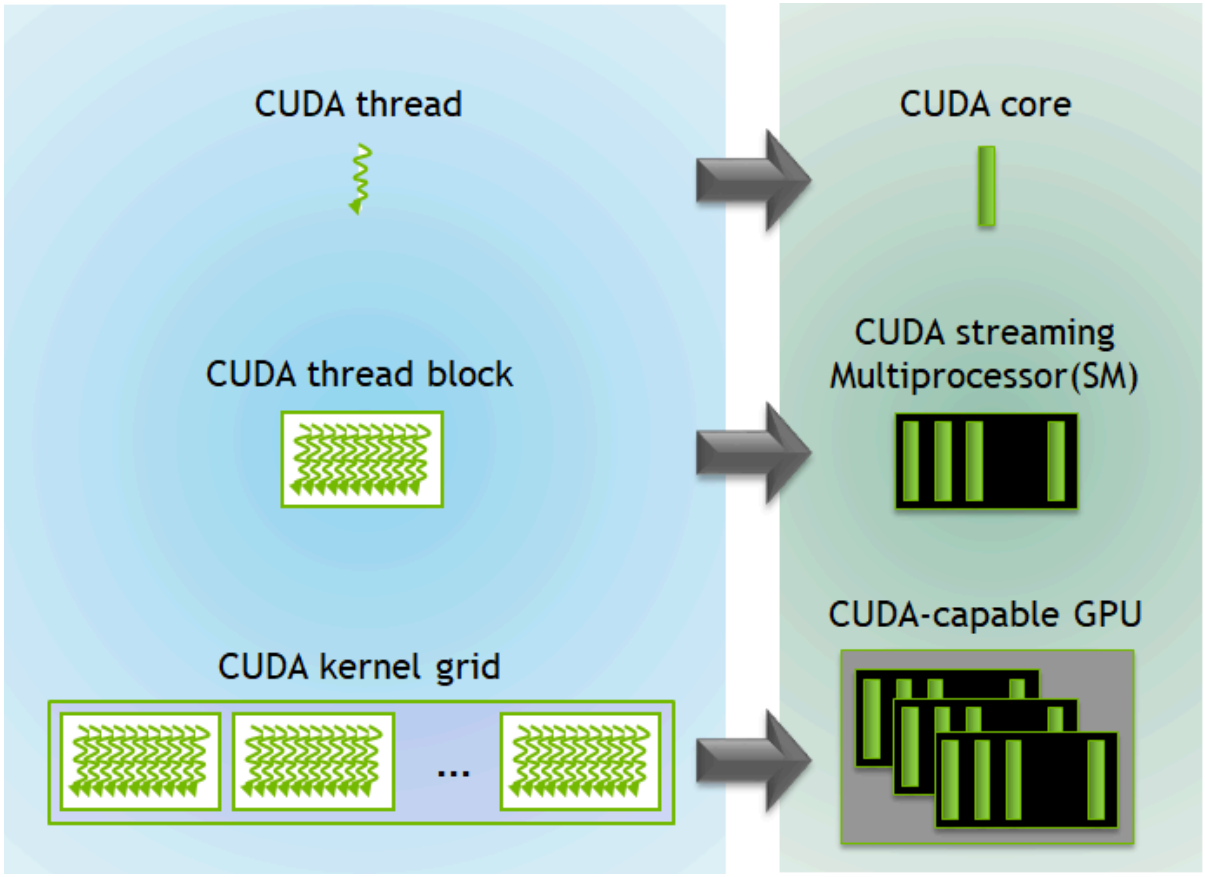
Each group of 32 consecutive threads is called a warp. A Warp is the primary unit of execution in an SM. Once a thread block is allocated to an SM, it will be further divided into a set of warps for execution. There is always a discrete number of warps per thread block and operations on a GPU are executed warp-wide.

Blocks

A group of threads is called a CUDA block. The group of threads can cooperate through shared memory and synchronize their memory access with execution. CUDA blocks are grouped into a grid. A kernel is executed as a grid of blocks of threads. The array of blocks execute the same kernel, scaling the parallel execution to all threads.

Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism). One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time.

GPU Kernel Execution



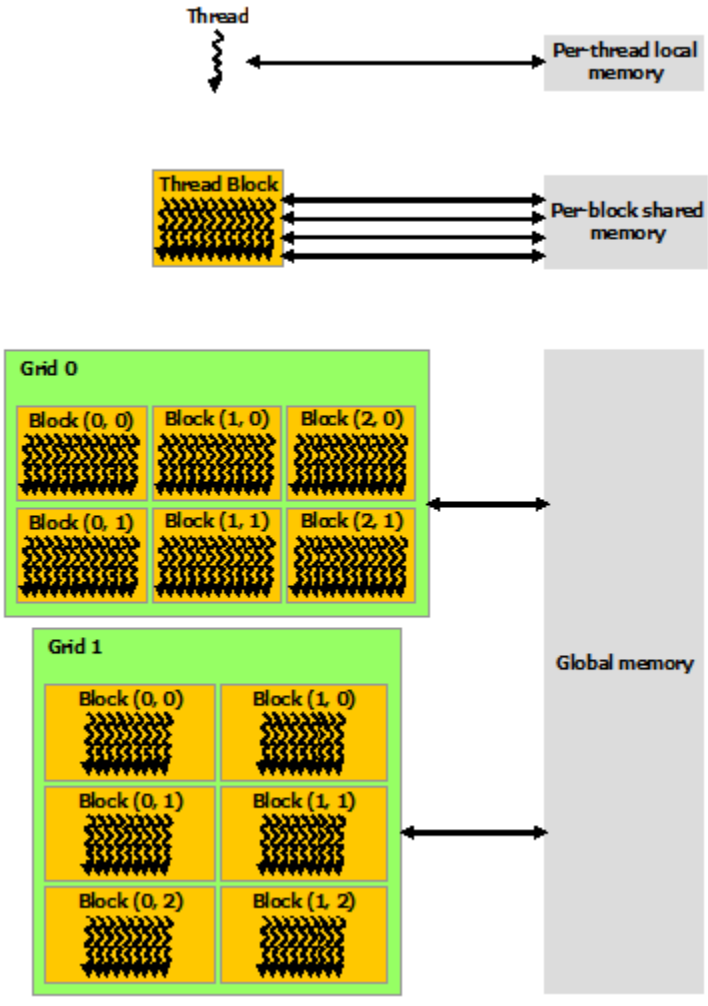
The hierarchical structure of these components not only aid in the management of thread execution, but also reflects the physical architecture of CUDA-enabled GPUs. Threads are executed concurrently on multiple Streaming Multiprocessors (SMs).

Memory Management

Understanding the CUDA memory hierarchy is crucial for optimizing the performance of your CUDA programs. GPUs feature several kinds of memory, each with its own scope, and cache. Using memory types in a way that efficiently takes advantage of shared memory can reduce latency and increase throughput.

Major Types include:

- Global memory - Can be accessed by all threads.
- Shared memory / L1 Cache - Shared among threads within the same block.
- Local - Private to each thread, but part of global memory.
- Registers - Private to each thread.



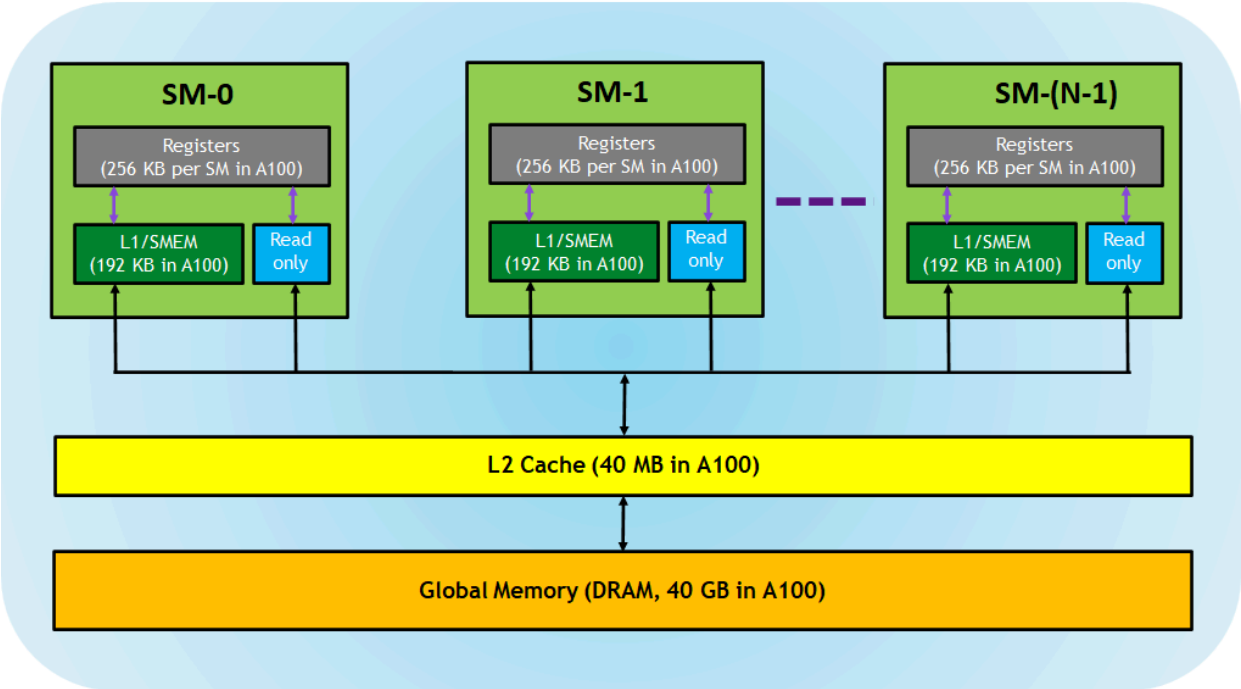
There are other types of memory to be aware of, but may be out of scope for crafting accelerated Python applications with high-level libraries.

- Constant memory - Cached read-only memory with access by all threads. In the case of an NVIDIA GPU, the shared memory, the L1 cache and the Constant memory cache are within the streaming multiprocessor block. Hence they are faster than the L2 cache, and GPU RAM
- Textures - Read-only memory optimized for filtering, interpolation methods, and random access by all threads.

The memory type will impact performance. For example, global memory is large but may increase latency. Local memory is private to each thread, but part of global memory so may also have long access times. Shared memory is faster, but limited in size.

For optimal performance, data should reside in the memory type closest to where it is being processed and best matches its access pattern. Data frequently accessed or shared among threads should be moved to shared memory or stored in registers whenever possible. This will reduce the overhead associated with fetching data from global memory.

Memory Hierarchy



The architecture creates a pool of managed memory where each allocation from this memory pool is accessible on both the host and the device with the same address or pointer. The underlying system migrates data to both the host and device.

The ability to transfer data between the CPU and GPU efficiently is the key to achieving optimal performance in GPU-accelerated applications. Data transfers between the CPU and GPU

involve moving data across the PCIe bus, which can limit the speed of accessing local memory on both CPU and GPU.

Moving Forward the Pythonic Way

These memory allocation types are in this chapter for reference, but the details of kernel execution, thread execution, and memory management are generally hidden in many high-level CUDA Python packages. Much generated boilerplate code and smart default values lead to performant code without having to call into low-level CUDA functionality. Python developers who are not familiar with C++ will also appreciate the pythonic nature of the API's.

The general consensus is that developers should use the highest-level library available to them for their application, and then explore lower-level options if necessary. Most high-level packages provide an interface to low-level functionality, so you have this option available if the high-level library calls do not give you the performance you seek.

Resources

CUDA C++ Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/
CUDA Toolkit: https://developer.nvidia.com/cuda-toolkit
NVIDIA CUDA Developer Forums:
https://forums.developer.nvidia.com/c/accelerated-computing/cuda/206