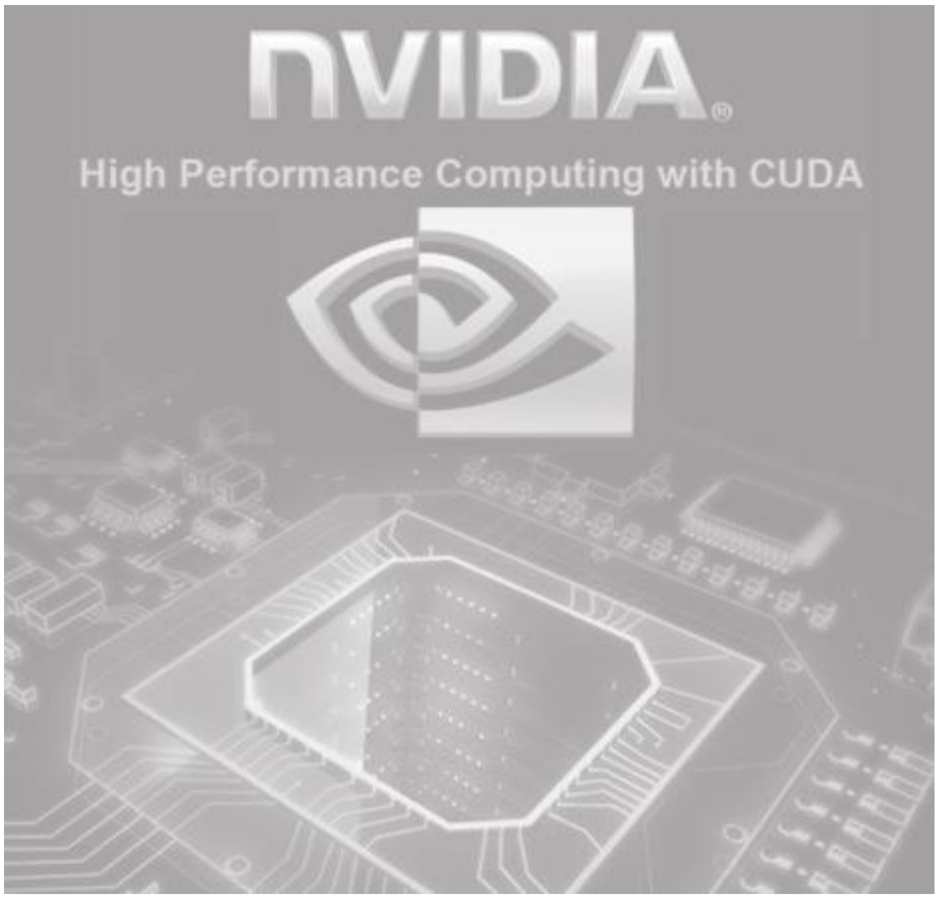
Chapter 2.

CUDA: *A Massively Parallel Architecture*



2. Introduction

With increasing demand for faster and more efficient parallel algorithms, researchers spend a great deal of time to invent new computer architectures. These parallel architectures, in contrast to traditional Von Neumann architecture, are carefully designed to support parallelism inherently. Many of these architectures are special purpose that is dedicated to solve certain problem domains. However, recent trend is to develop general-purpose architectures. Parallel processing on GPU has been new craze over past decade.

2.1. GPGPU

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit (GPU) has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth. Thus the GPU is switched from the conventional graphic processing to general purpose parallel computing. Hence, it is attributed as the general purpose graphic processing unit (GPGPU).

2.2. The Secret Behind GPU's Power

Figure 2.1 shows the capability of GPU compared to CPU in context of floating-point operations [1].

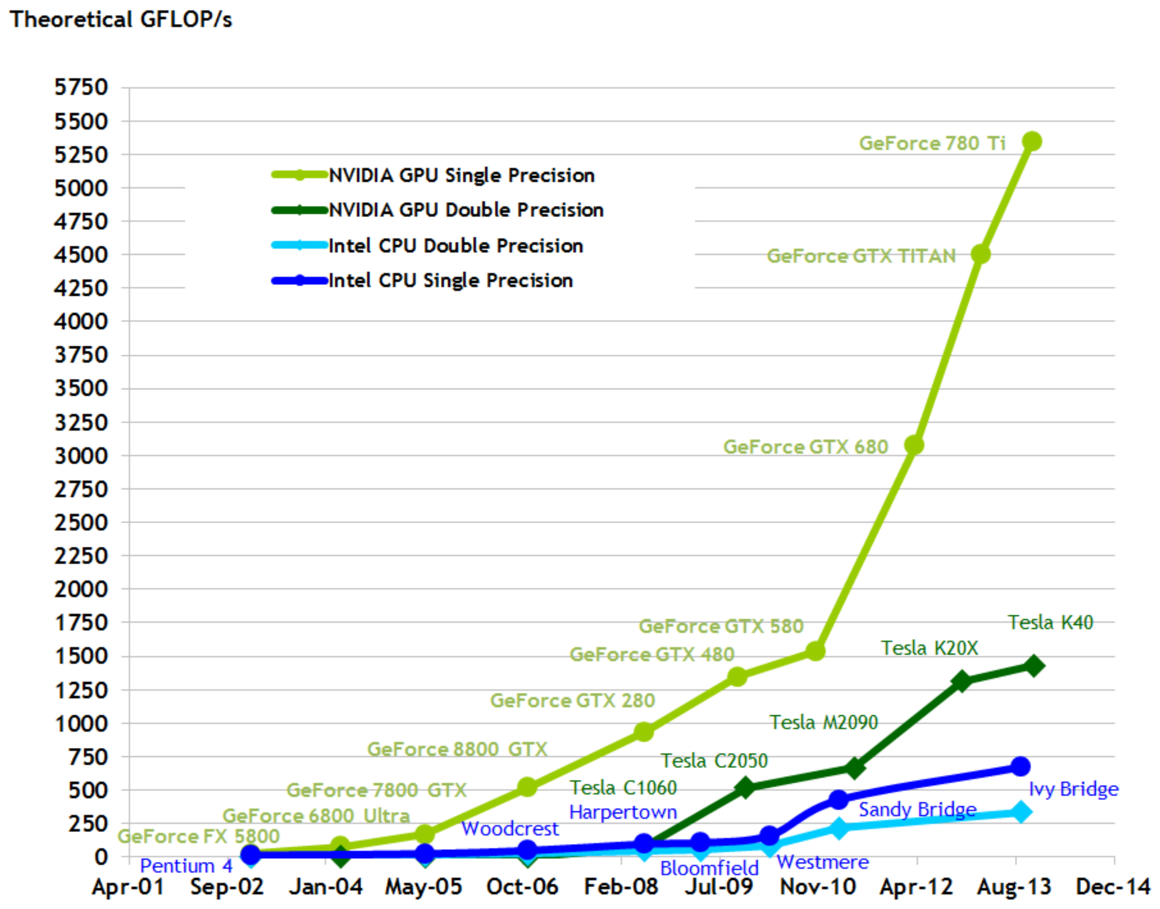


Figure 2.1: Floating-Point Operations per Second for the CPU and GPU

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 2.2.

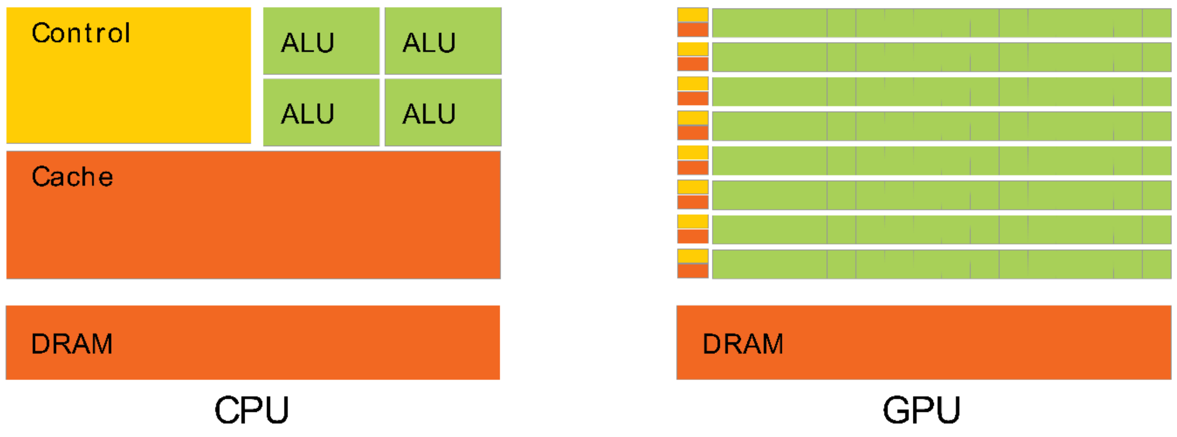


Figure 2.2: The GPU Devotes More Transistors to Data Processing

More specifically, the GPU is especially well suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

2.3. The CUDA Model

In November 2006, NVIDIA introduced CUDA®, 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. The acronym CUDA stands for Compute Unified Device Architecture.

CUDA comes with a software environment that allows developers to use C/C++ as a high level programming language. In addition to this, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenACC.

2.4. Scalability and CUDA

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are 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.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. 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 as shown by Figure 2.3, and only the runtime system needs to know the physical multiprocessor* count.

This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs [2].

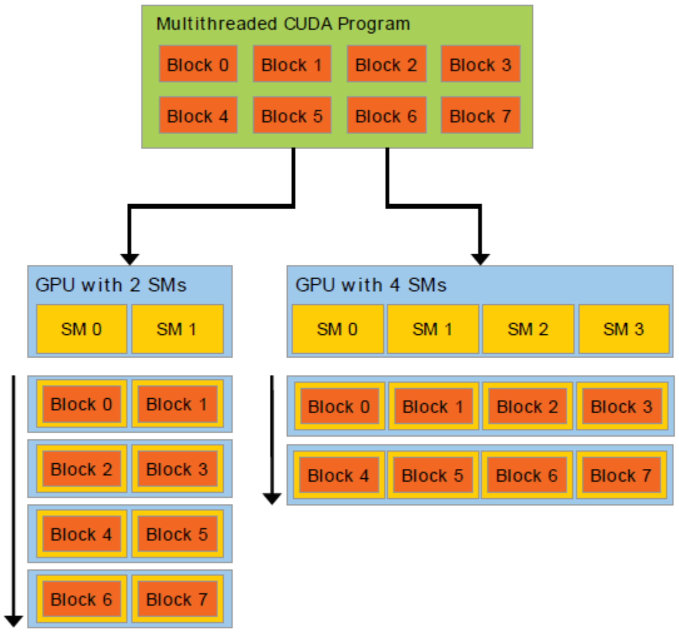


Figure 2.3: Automatic Scalability

* A GPU is built around an array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads, which will automatically assigned available multiprocessors to dynamically adopt the device specific hardware.

2.5. Programming Model

Here we introduce the main concepts behind the CUDA programming model by outlining how they are exposed in C/C++. An extensive description of CUDA C is given in the next section.

We consider a vector addition example and a matrix addition example for better description of CUDA programming.

2.5.1. Kernels

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

A kernel function is defined using the `__global__` declaration specifier and the number of CUDA threads, N, that will execute the kernel for a particular kernel call is specified using `<<<...>>>`, the new *execution configuration* syntax. Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in **threadIdx** variable.

The following sample code adds two vectors A and B of size N and stores the result into vector C:

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Here, each of the N threads that execute **`vecAdd()`** performs one pair-wise addition.

2.5.2. Thread Hierarchy

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 thread block. This provides a natural way to invoke 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 one-dimensional block, they are the same; for a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is ($x + y D_x$); for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is ($x + y D_x + z D_x D_y$).

Consider this code which adds two matrices A and B of size NxN and stores the result into matrix C:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

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. For details, refer to appendix A.

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.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure 2.4. 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.

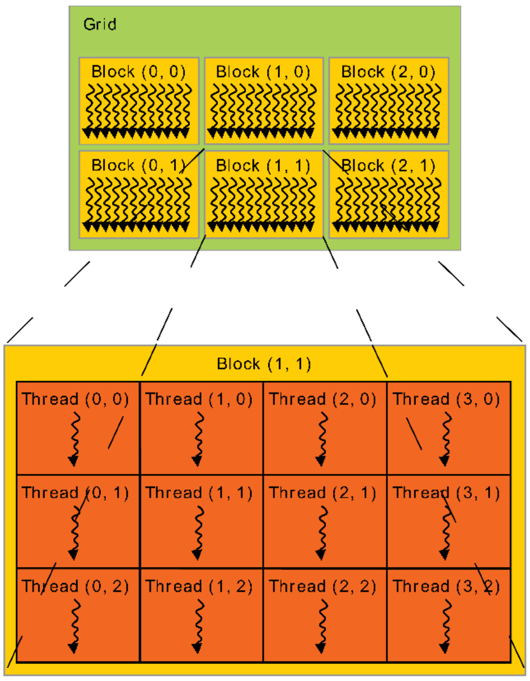


Figure 2.4: Grid of Thread Blocks

The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type int or dim3. Two-dimensional blocks or grids can be specified in the execution configuration as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-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.

Extending the previous `MatAdd()` example to handle multiple blocks, the code becomes as follows.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A thread block size of 16x16 (i.e., 256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

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 in any order across any number of cores as illustrated by Figure 3, 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. The `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed. This barrier synchronization is effective only within a block.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and `__syncthreads()` is expected to be lightweight.

2.5.3. Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2.5. 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.

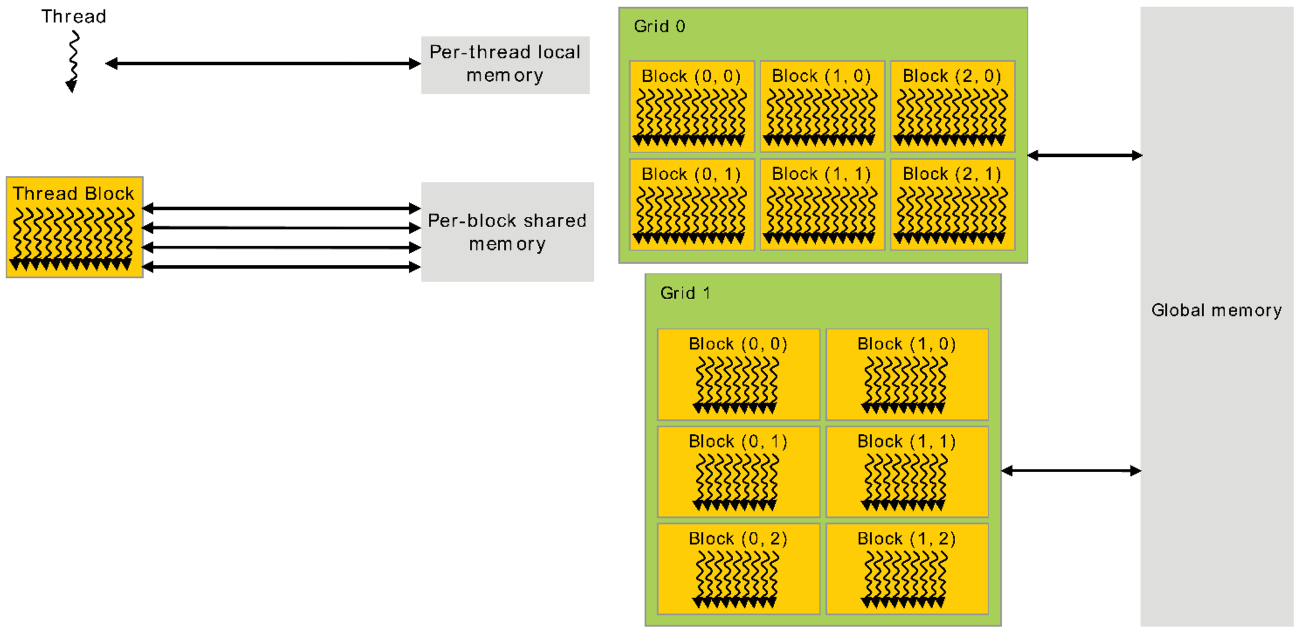


Figure 2.5: Memory Hierarchy

2.5.4. Heterogeneous Programming

As illustrated by Figure 2.6, the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

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.

Note: Serial code executes on the host (CPU) while parallel code executes on the device (GPU).

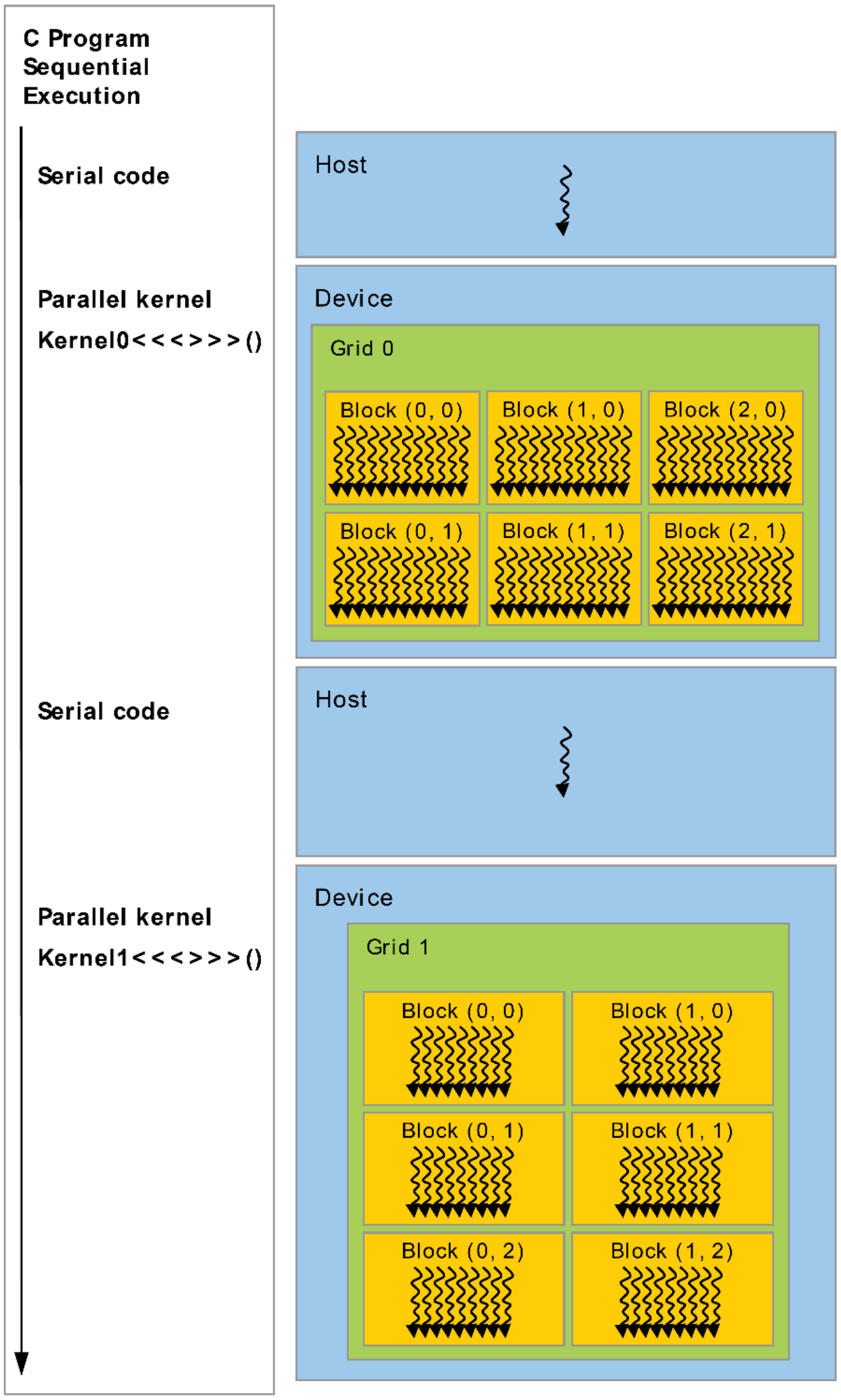


Figure 2.6: Heterogeneous Programming

2.6. Compute Capability

The compute capability of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

The compute capability version comprises a major and a minor version number (x.y). Devices with the same major revision number are of the same core architecture. The major revision number is 5 for devices based on the Maxwell architecture, 3 for devices based on the Kepler architecture, 2 for devices based on the Fermi architecture, and 1 for devices based on the Tesla architecture.

The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features. The online CUDA-Enabled GPUs list [2] gives the technical specifications of each compute capability.

Note: The compute capability version of a particular GPU should not be confused with the CUDA version (e.g., CUDA 5.5, CUDA 6, CUDA 6.5, CUDA 7.0), which is the version of the CUDA software platform. The CUDA platform is used by application developers to create applications that run on many generations of GPU architectures, including future GPU architectures yet to be invented. While new versions of the CUDA platform often add native support for a new GPU architecture by supporting the compute capability version of that architecture, new versions of the CUDA platform typically also include software features that are independent of hardware generation.

References

[1] NVIDIA CUDA C programming guide available at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] List of CUDA-enabled GPUs available at https://developer.nvidia.com/cuda-gpus