NVIDIA CUDA™

NVIDIA CUDA C
Programming Guide

Version 4.2

4/16/2012

Chapter 2.
Programming Model

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C. An extensive description of CUDA C is given in Chapter 3.

Full code for the vector addition example used in this chapter and the next can be found in the *vectorAdd* SDK code sample.

2.1 Kernels

CUDA C extends C 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 is defined using the **`__global__`** declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<<…>>>** *execution configuration* syntax (see Appendix B.18). Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in **`threadIdx`** variable.

As an illustration, 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.

2.2 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 $(x + y D_x + z D_x D_y)$.

As an example, the following code 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.

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 2-1. 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.

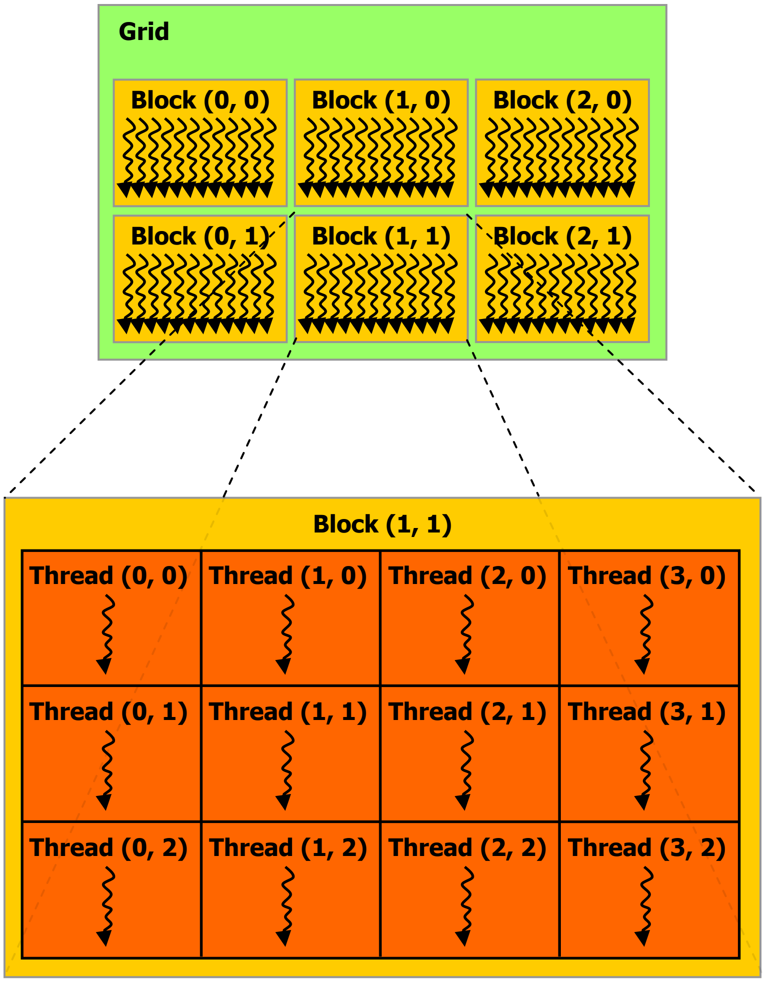


Figure 2-1. Grid of Thread Blocks

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 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 (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 1-4, 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. Section 3.2.3 gives an example of using shared memory.

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.

2.3 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2-2. 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 (see Sections 5.3.2.1, 5.3.2.4, and 5.3.2.5). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Section 3.2.10).

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

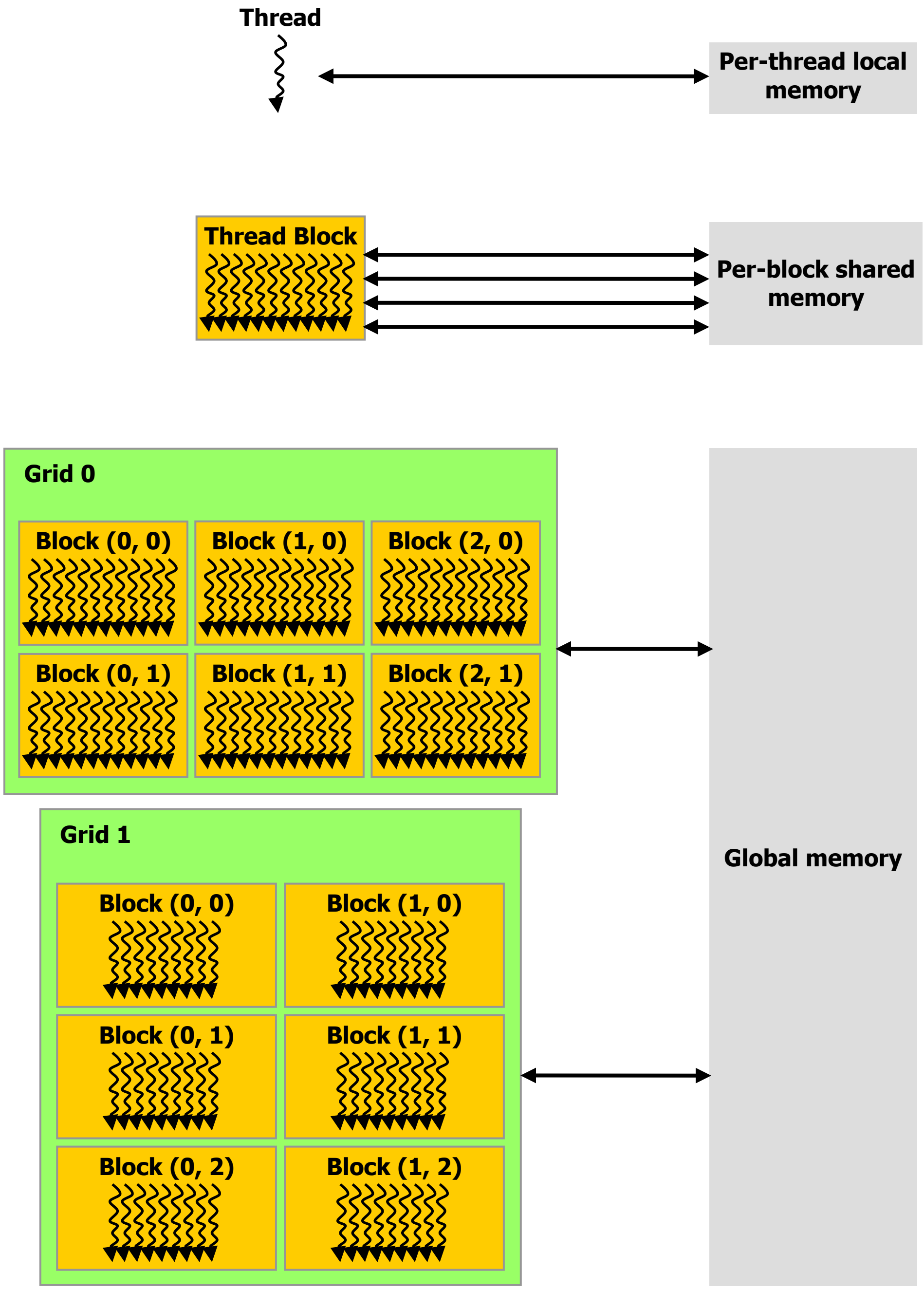
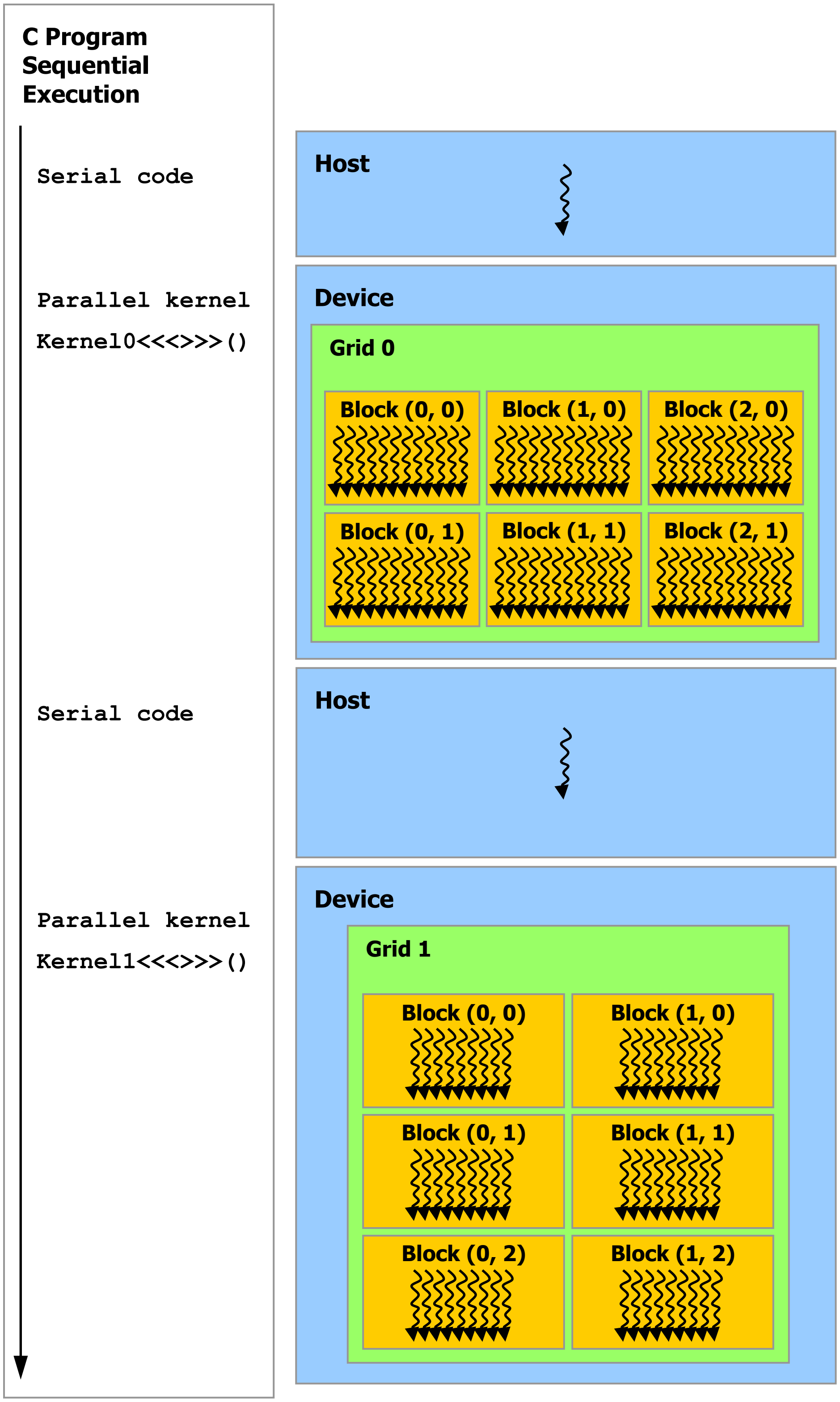


Figure 2-2. Memory Hierarchy

2.4 Heterogeneous Programming

As illustrated by Figure 2-3, 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 (described in Chapter 3). This includes device memory allocation and deallocation as well as data transfer between host and device memory.



Serial code executes on the host while parallel code executes on the device.

Figure 2-3. Heterogeneous Programming

2.5 Compute Capability

The *compute capability* of a device is defined by a major revision number and a minor revision number.

Devices with the same major revision number are of the same core architecture. The major revision number is 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.

Appendix A lists of all CUDA-enabled devices along with their compute capability. Appendix F gives the technical specifications of each compute capability.