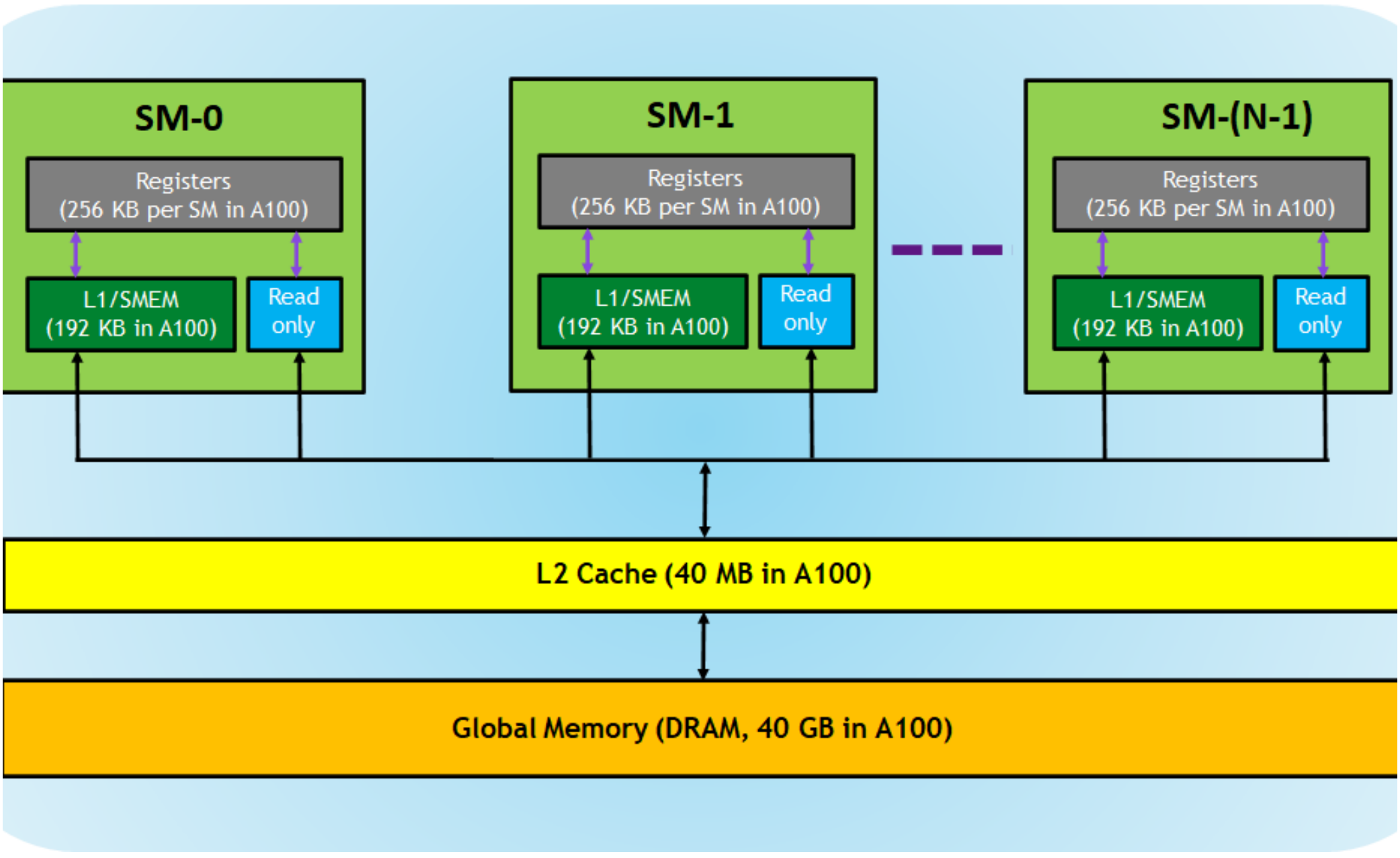
Technical Walkthrough

Jun 26, 2020

CUDA Refresher: The CUDA Programming Model

By Pradeep Gupta

Tags: CUDA, CUDA Refresher, Parallel Programming



This is the fourth post in the CUDA Refresher series, which has the goal of refreshing key concepts in CUDA, tools, and optimization for beginning or intermediate developers.

The CUDA programming model provides an abstraction of GPU architecture that acts as a bridge between an application and its possible implementation on GPU hardware. This post outlines the main concepts of the CUDA programming model by outlining how they are exposed in general-purpose programming languages like C/C++.

Let me introduce two keywords widely used in CUDA programming model: *host* and *device*.

The host is the CPU available in the system. The system memory associated with the CPU is called host memory. The GPU is called a device and GPU memory likewise called device memory.

To execute any CUDA program, there are three main steps:

- Copy the input data from host memory to device memory, also known as host-to-device transfer.
- Load the GPU program and execute, caching data on-chip for performance.
- Copy the results from device memory to host memory, also called device-to-host transfer.

CUDA kernel and thread hierarchy

Figure 1 shows that the CUDA kernel is a 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 C/C++ functions.

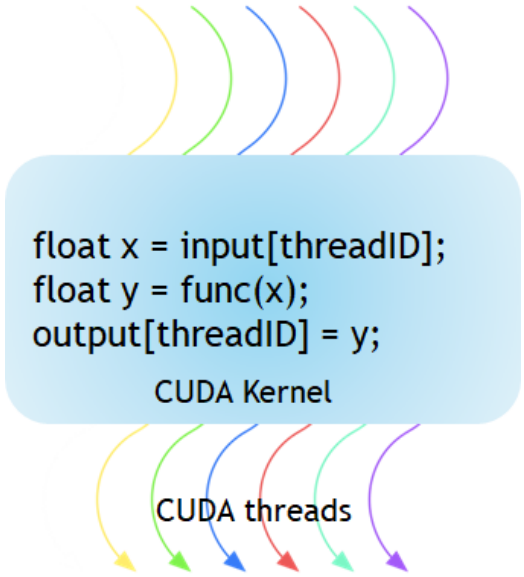


Figure 1. The kernel is a function executed on the GPU.

Every CUDA kernel starts with a `__global__` declaration specifier. Programmers provide a unique global ID to each thread by using built-in variables.

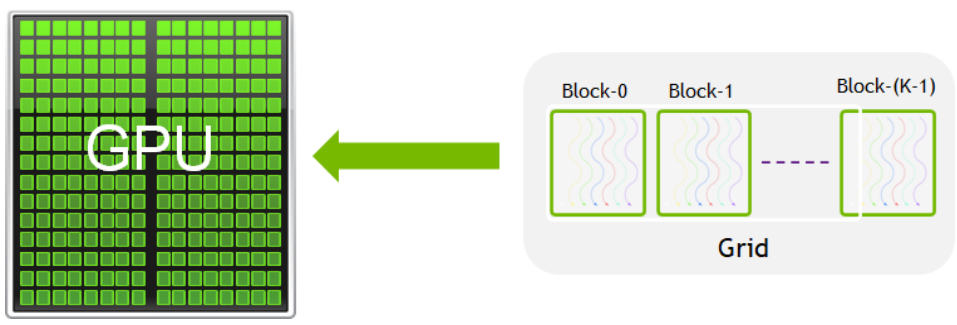


Figure 2. CUDA kernels are subdivided into blocks.

A group of threads is called a CUDA block. CUDA blocks are grouped into a grid. A kernel is executed as a grid of blocks of threads (Figure 2).

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. Figure 3 shows the kernel execution and mapping on hardware resources available in GPU.

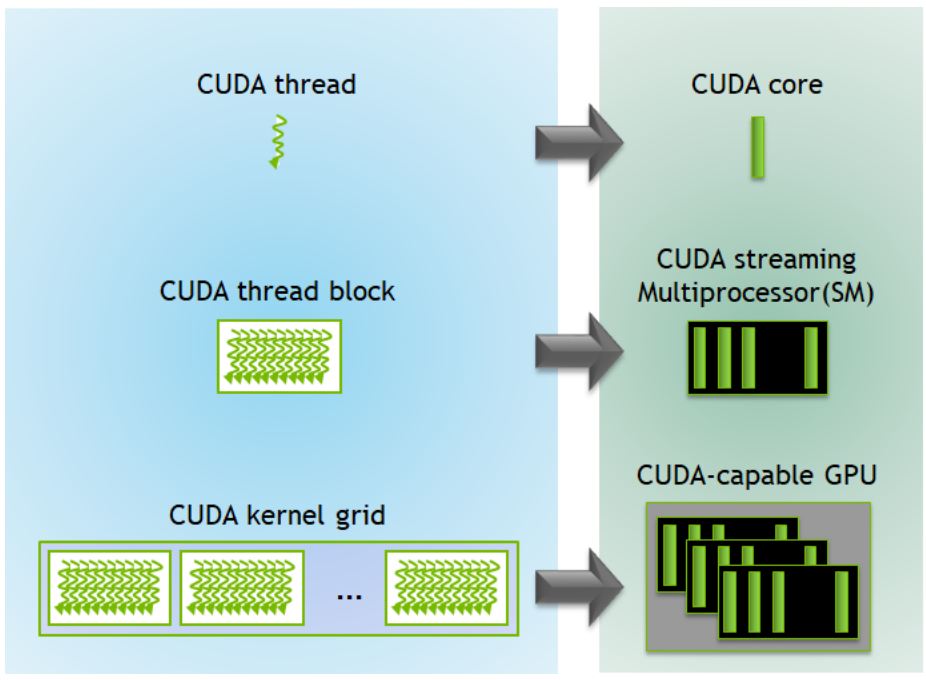


Figure 3. Kernel execution on GPU.

CUDA defines built-in 3D variables for threads and blocks. Threads are indexed using the built-in 3D variable `threadIdx`. Three-dimensional indexing provides a natural way to index elements in vectors, matrix, and volume and makes CUDA programming easier. Similarly, blocks are also indexed using the in-built 3D variable called `blockIdx`.

Here are a few noticeable points:

- CUDA architecture limits the numbers of threads per block (1024 threads per block limit).
- The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.
- All threads within a block can be synchronized using an intrinsic function `__syncthreads`. With `__syncthreads`, all threads in the block must wait before anyone can proceed.
- The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type int or dim3. These triple angle brackets mark a call from host code to device code. It is also called a kernel launch.

The CUDA program for adding two matrices below shows multi-dimensional `blockIdx` and `threadIdx` and other variables like `blockDim`. In the example below, a 2D block is chosen for ease of indexing and each block has 256 threads with 16 each in x and y-direction. The total number of blocks are computed using the data size divided by the size of each block.

```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N],
float MatC[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        MatC[i][j] = MatA[i][j] + MatB[i][j];
}

int main()
{
    ...
    // Matrix addition kernel launch from host code
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + threadsPerBlock.x -1) / threadsPerBlock.x, (N+threadsPerBlock.y -1) / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(MatA, MatB, MatC);
    ...
}
```

Memory hierarchy

CUDA-capable GPUs have a memory hierarchy as depicted in Figure 4.

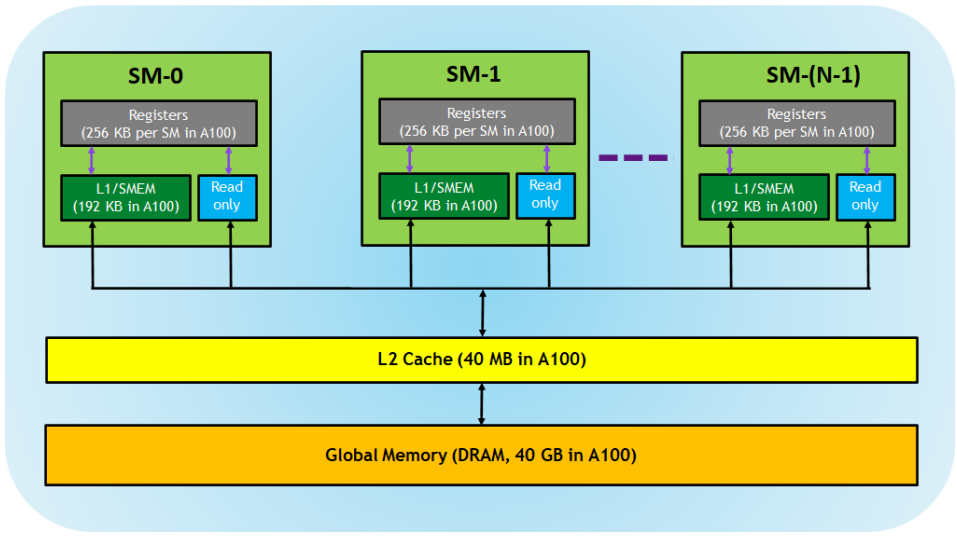


Figure 4. Memory hierarchy in GPUs.

The following memories are exposed by the GPU architecture:

- **Registers**—These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.
- **L1/Shared memory (SMEM)**—Every SM has a fast, on-chip scratchpad memory that can be used as L1 cache and shared memory. All threads in a CUDA block can share shared memory, and all CUDA blocks running on a given SM can share the physical memory resource provided by the SM..
- **Read-only memory**—Each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.
- **L2 cache**—The L2 cache is shared across all SMs, so every thread in every CUDA block can access this memory. The **NVIDIA A100 GPU** has increased the L2 cache size to 40 MB as compared to 6 MB in V100 GPUs.
- **Global memory**—This is the framebuffer size of the GPU and DRAM sitting in the GPU.

The NVIDIA CUDA compiler does a good job in optimizing memory resources but an expert CUDA developer can choose to use this memory hierarchy efficiently to optimize the CUDA programs as needed.

Compute capability

The compute capability of a GPU determines its general specifications and available features supported by the GPU hardware. This version number can be used by applications at runtime to determine which hardware features or instructions are available on the present GPU.

Every GPU comes with a version number denoted as X.Y where X comprises a major revision number and Y a minor revision number. The minor revision number corresponds to an incremental improvement to the architecture, possibly including new features.

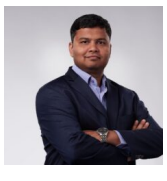
For more information about the compute capability of any CUDA-enabled device, see the CUDA sample code **deviceQuery**. This sample enumerates the properties of the CUDA devices present in the system

Summary

The CUDA programming model provides a heterogeneous environment where the host code is running the C/C++ program on the CPU and the kernel runs on a physically separate GPU device. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively. CUDA code also provides for data transfer between host and device memory, over the PCIe bus.

CUDA also exposes many built-in variables and provides the flexibility of multi-dimensional indexing to ease programming. CUDA also manages different memories including registers, shared memory and L1 cache, L2 cache, and global memory. Advanced developers can use some of these memories efficiently to optimize the CUDA program.

About the Authors



About Pradeep Gupta

Pradeep Gupta is director of the Solutions Architecture and Engineering team at NVIDIA. He is responsible for running technical customer engagements for industries like autonomous driving, healthcare, and telecoms where AI is transforming many possible aspects of industry solutions. His focus is on building production-grade AI that can be deployed in life-critical systems. Previously, Pradeep worked in areas like high-performance computing, computer vision, mathematical library development, and data center technologies. He received a master's degree in research from the Indian Institute of Science (IISc), Bangalore. His research focused on developing compute-efficient algorithms.

Follow @@pkgnvi on Twitter

View all posts by Pradeep Gupta »

Comments

Start the discussion at forums.developer.nvidia.com