**GPU architecture review report**

Ruijian An

1002965552

**Motivation**

**Part 1: CUDA programming model: Minimal extension to C**

**Part 2: GPU Hardware and Execution Model: Hardware, Kernel execution model, problems and solutions**

**Part 3: GPU Memory Model: Hierarchy, Speed, Optimization**

**Motivation:**

In recent years, many problems are encountered when making increasingly powerful CPU. First, processors designers fail to keep increasing clock speed. Second, although CPU has complex control structure and is good at running complicated serial code, it is quite time-consuming when processing huge amounts of data. As a result, to gain more computational resources, it is necessary to apply new ideas to build processors, thus giving birth to GPU. To achieve more computational power and increase power efficiency, contrary to CPU, GPU architecture chooses to: 1) focus on throughput rather than latency,2) optimize for running code in parallel, 3) build simple computing units.

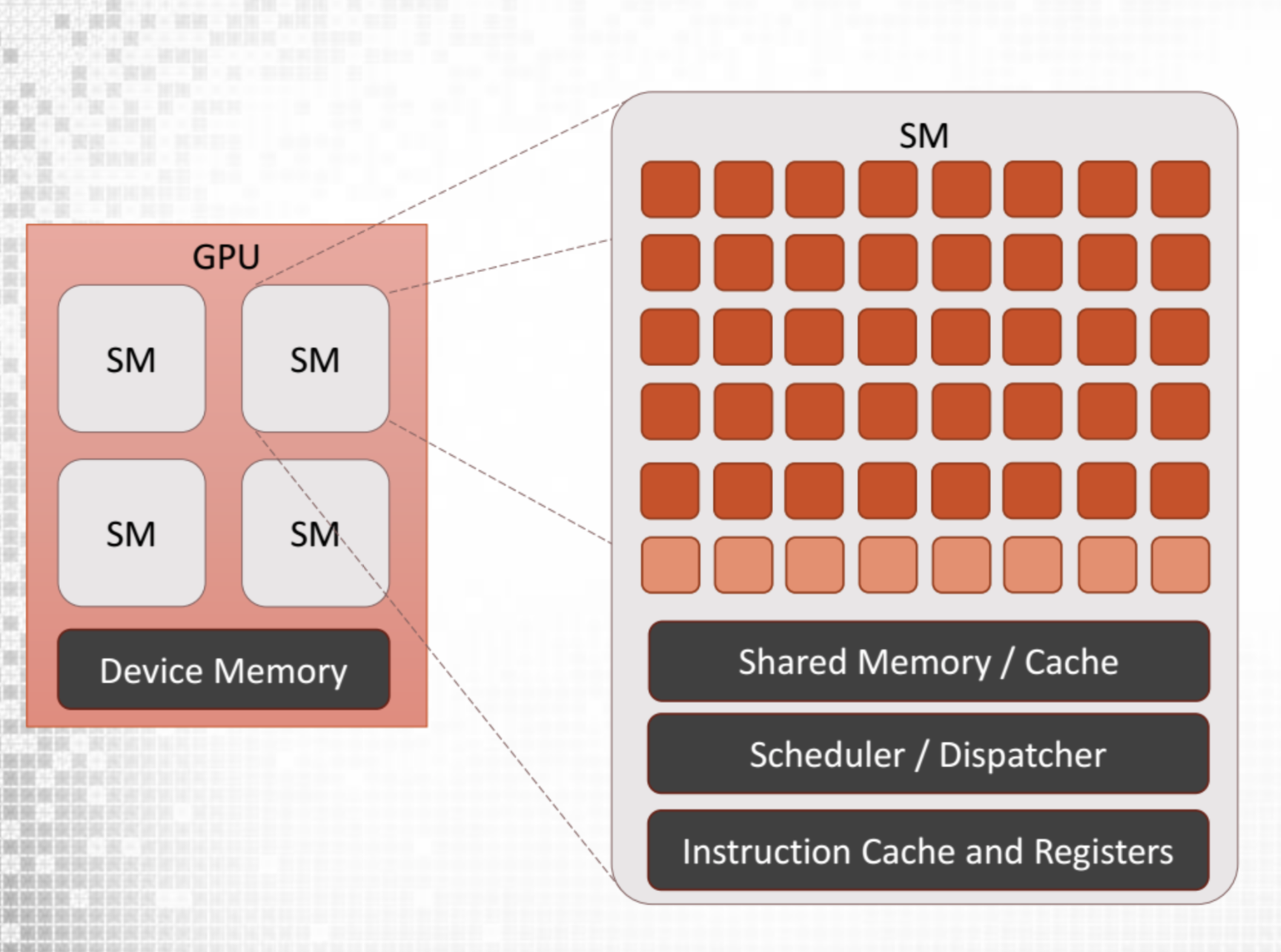
**Part 1:**

**CUDA**（**C**ompute **U**nified **D**evice **A**rchitecture）is a parallel computing platform designed by Nvidia. CUDA programs run on CPU and invoke GPU part by special call and a typical CUDA program consists of following four parts:

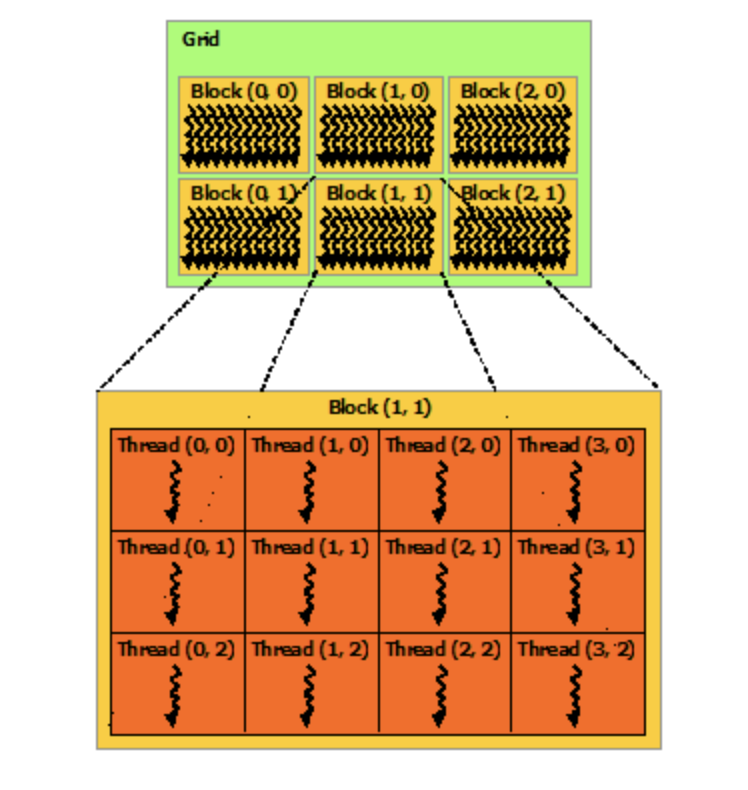
1. CPU allocates storage on GPU.
2. CPU copies input data from CPU to GPU.
3. CPU launches kernels on GPU to process the data.
4. CPU copies result back to CPU from GPU.

CUDA is convenient for programmers, because it is an extension to standard C programming language. Since a CUDA program has both host code and device code, new declaration specifiers like \_\_global\_\_, \_\_device\_\_, \_\_host\_\_ are defined to indicate where things can be called or executed. Kernel, which would initialize the parallel operations on GPU, is the key part of a CUDA program. Kernel will be launched by the syntax Kernel<<<Grid\_szie, Block\_size>>> (arguments);. Also, CUDA gives a straightforward way to program in parallel on GPU.

**Part 2:**



GPU hardware is a two-level hierarchy. It is built around a bunch of *streaming multiprocessors* (also called as *SMs*) where perform actual computations. Each streaming multiprocessor consists of many CUDA cores and only one instruction model, so the CUDA cores on a single SM run the same set of instructions.



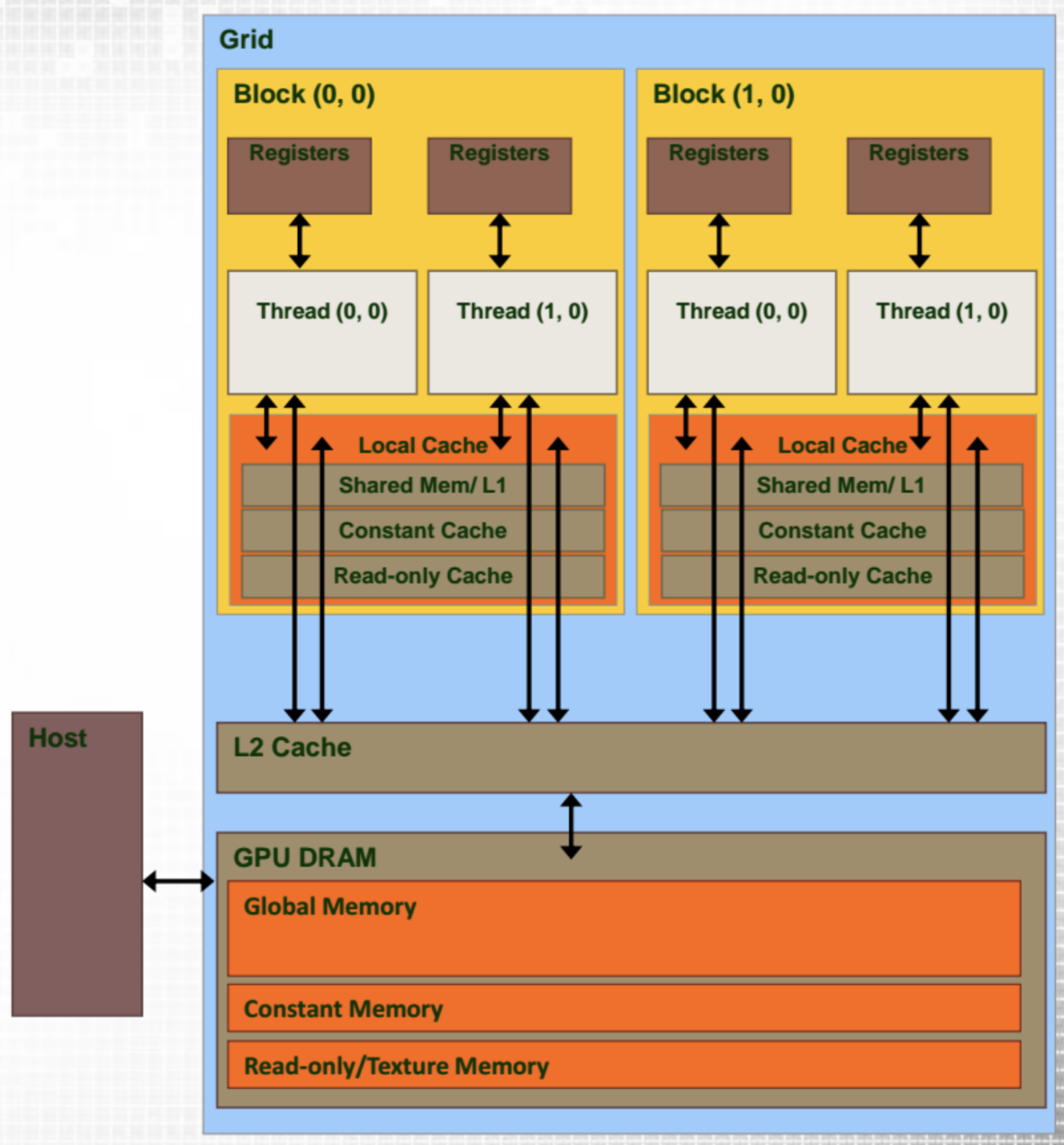
One of the advantages of CUDA is that it virtualizes the hardware.

Every kernel call generates grids of blocks and blocks of threads whose sizes are determined by two arguments Grid\_size and Block\_size, as shown in the right picture. After CPU launches the kernel on the device, GPU allocates each block of threads to a streaming multiprocessor. Since GPU has many multiprocessors, each block would handle a small part the task in parallel, avoiding the sequential loops which would increase time complexity. Inside the streaming multiprocessor, threads are always executed in a 32-threads group which is called *warp* and every thread block would always be mapped to some warps. GPU uses SIMT (Single Instruction, Multiple Thread) execution model which is the combination of SIMD (Single Instruction, Multiple Data) execution of warp-size threads within one single block and SPMD (Single Program, Multiple Data) execution across blocks. Consequently, GPU hardware can do operations parallelly in both thread-level and data-level.

Another advantage of GPU is scalability. When the number of blocks exceeds the number of SM, the hardware is responsible for assigning new thread blocks to streaming multiprocessors once any SM is available. As a result, programmer do not need to modify their programs for different kinds of GPUs with different number of streaming multiprocessors because of the scalability.

CUDA hardware and programming model well fit ideas of parallel computing, but some problems may occur. Different threads in one block can cooperate to do a task and sometimes one thread may need result from another thread. However, since the operation order of threads is not guaranteed by GPU, the thread might read the result before another one writes it. Under such circumstances, synchronization which stops every thread until all executions finishes is needed.

Another problem is thread divergence. Inside a warp, all thread work synchronously and the hardware cannot execute if statement and else statement at the same time. If the threads in a warp takes on different path, hardware runs the code in serial instead of in parallel. To avoid the impact that divergence has on the performance, we need to minimize the conditional code.

**Part 3:**

GPU has a memory hierarchy composed of registers, shared memory, local memory, global memory and L1, L2 cache. Global memory can be accessed by all threads on the GPU, but it is the slowest since global memory is a separate hardware from streaming multiprocessors. Registers are private to and can be used directly by one thread and it is the fastest memory. Given the fact that registers memory is limited, the data that cannot fit in registers would be stored in local memory which is a part of global memory. Shared memory, which all threads on the thread block can access, is located at the streaming multiprocessor and it is very fast too. However, shared memory is arranged into 32 banks, which may cause bank conflicts. Bank conflicts happens when different threads in a warp want to access the data at the same time. GPU hardware can only allow the thread to access the bank in serial. As a result, bank conflict would decrease the I/O efficiency, making shared memory slower than registers.

To optimize the memory hierarchy, it is necessary to minimize the time spent on memory. First, coalescing is a good way to improve performance. In fact, a thread always accesses a large chunk of memory when it reads or writes global memory. Coalescing means the many threads access contiguous memory. It largely improves efficiency since it reduces the number of memory transactions. Second, different kinds of memory’s speed are not uniform, so we can put most frequently accessed data in the fastest memory.

Reference:

1. ‘Introduction to parallel computing’, Udacity
2. University of Sheffield: COM4521: Parallel Computing with GPUs, Taught by Paul Richmond
3. CUDA Programming Guide, Nvidia