Lec4: Architecture and Scheduling

Textbook Chapter 4

Things to discuss

- GPU execution resources: cores (SM), SIMD lane (CUDA core)
- Thread blocks scheduling to SM
- Warp: a group of threads executing the same instruction (SIMD lanes?), a unit of scheduling
- Warp scheduling, latency tolerance, control divergence
- Resource limits

Architecture of a GPU

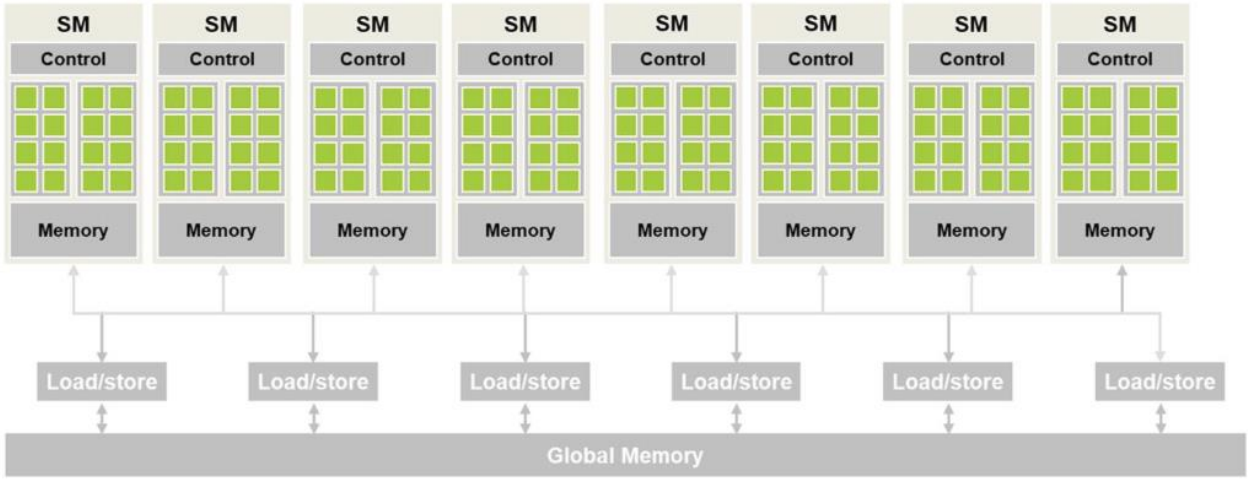

FIGURE 4.1 Architecture of a CUDA-capable GPU.

- Programmer's view of architecture
- E.g. Ampere A100 GPU has 108 SMs with 64 cores each, totaling 6912 CUDA cores
- Global memory: DRAM, HBM
- SM has local memory—called shared memory—dual use as L1 cache and scratchpad

Block Scheduling

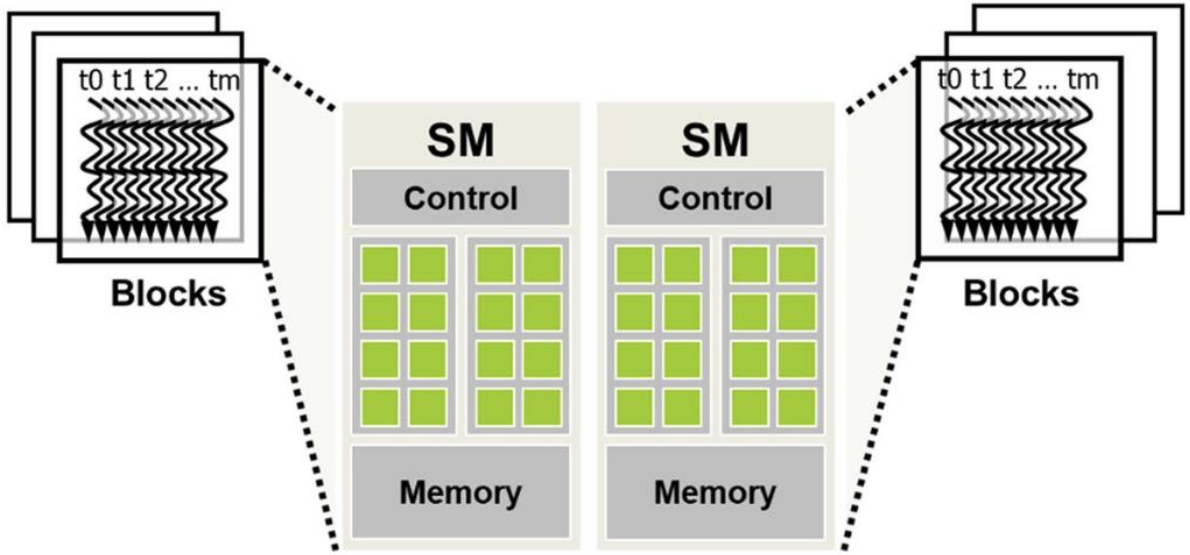

FIGURE 4.2 Thread block assignment to streaming multiprocessors (SMs).

- Upon kernel launch, blocks are assigned to SM on a block-by-block basis
- Block as a unit: to the same SM.
- Once assigned to a SM, the block do not move to other SM.
- Limited number of blocks can be simultaneously assigned to a SM (what limits?)
- There is a queue, a list of blocks which will be assigned to SMs when they become available.

Synchronization & Scalability

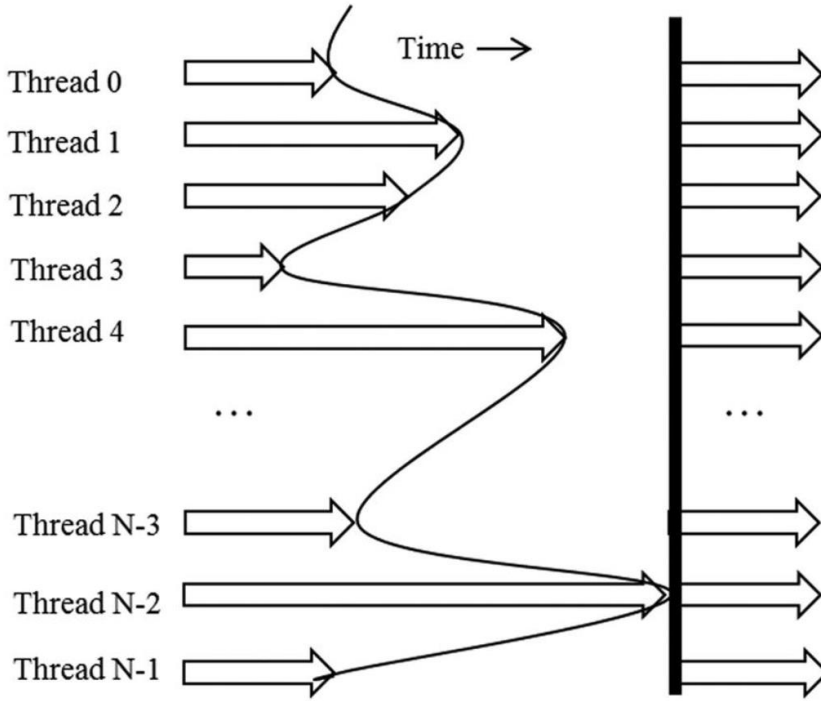

FIGURE 4.3 An example execution of barrier synchronization. The arrows represent execution activities over time. The vertical curve marks the time when each thread executes the `__syncthreads` statement. The empty space to the right of the vertical curve depicts the time that each thread waits for all threads to complete. The vertical line marks the time when the last thread executes the `__syncthreads` statement, after which all threads are allowed to proceed to execute the statements after the `__syncthreads` statement.

- Threads in a block can have a barrier synchronization via __syncthreads() call.
- Threads have their own progress—how do I say wait until other threads has done xxx?
- Barrier: wait until all threads have reached this.
- All threads in the block must call __syncthreads()
- What if there is a branch?

Barrier in branches

```
void incorrect_barrier_example(int n) {
    // ...
    if (threadIdx. x % 2 == 0) {
        //...
        __syncthreads();
    else {
        //...
        __syncthreads();
    }

}
```

- The two __syncthreads() in the two branches are **different** barrier; roughly you can say something like “the barrier at line 5”.
- Because of that, the device code is not valid as it causes deadlocks, as not all threads in a block call the same __syncthreads().

Design Tradeoffs

- Threads in a block (intra-block)—can synchronize and communicate easier and faster
- Threads across blocks (inter-blocks)-–cannot synchronize and communicate slower via global memory.
- Why limiting synchronization to intra-block threads?
  For **scalability**.

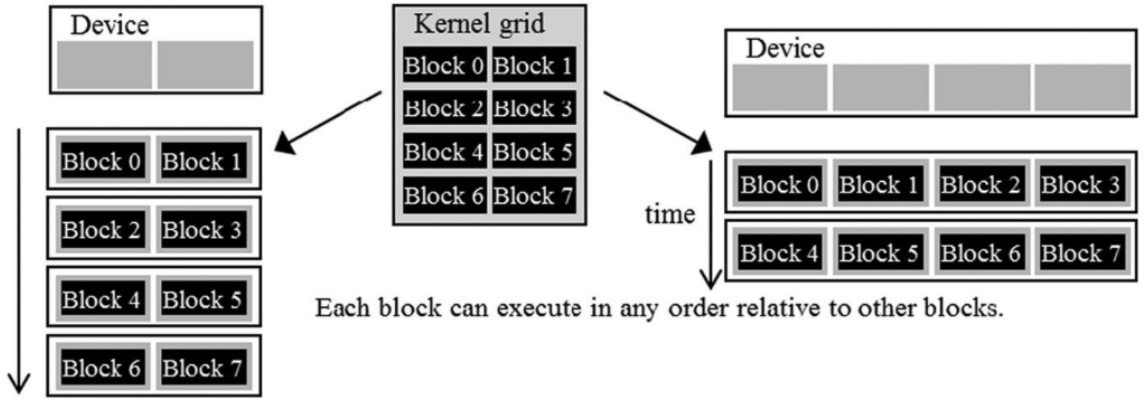

FIGURE 4.5 Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

Warp and SIMD hardware

- Within a block, threads are scheduled for execution in unit of **warp,** a group of 32 threads.
- Warp is an architecture concept, not visible in CUDA/C language
- A warp of threads must be executing the same instruction, because they have a single Program Counter (PC)
- Why? Again, this is a tradeoff.
- In principle, threads are independent, with their own PC and registers, context, so on.
- But how to use SIMD lanes with threads? (SIMD is more efficient, as functional units share control unit to fetch/decode/dispatch)
- You group threads that execute the same PC and schedule them on a SIMD pipeline.
- That’s a warp!

Warp

- Blocks are divided into warps statically based on threadIdx.
- A consecutive 32 threads in threadIdx.x belong to one block.
- I.e., in 1D block, threadIdx.x 0,1,…,31 is warp0, 32,33,…,63 is warp1, …

- For multi-dim block, the threadIdx will be linearized and then divided into 32 consecutive groups as warp

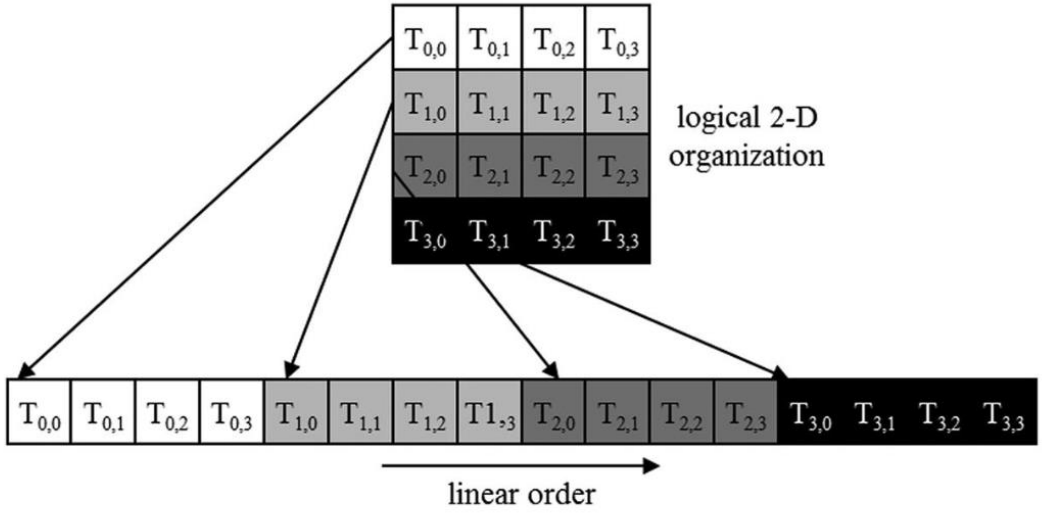

FIGURE 4.7 Placing 2D threads into a linear lay-out.

Warp execution

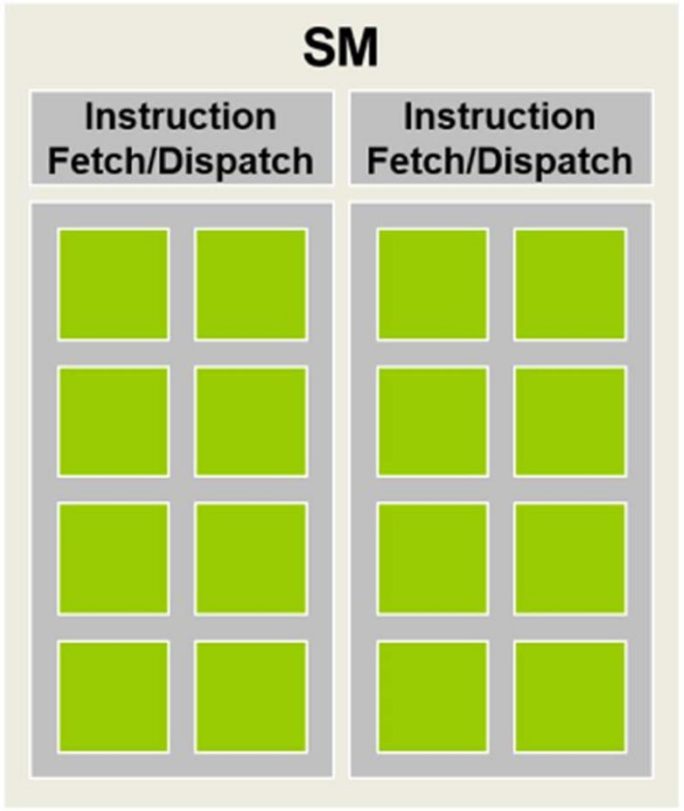

FIGURE 4.8 Streaming multiprocessors are organized into processing blocks for SIMD execution.

- At any point time, a warp is scheduled on a processing block (say 8 CUDA cores).
- E.g. A100 GPU each SM has 64 CUDA cores, forming 4 processing blocks. In this case, 4 warps can be scheduled simultaneously on a SM.
- This is where the term Single Instruction Multiple Threads (SIMT) come from.

Control Divergence

- Wait, but different threads are free to execute different paths!

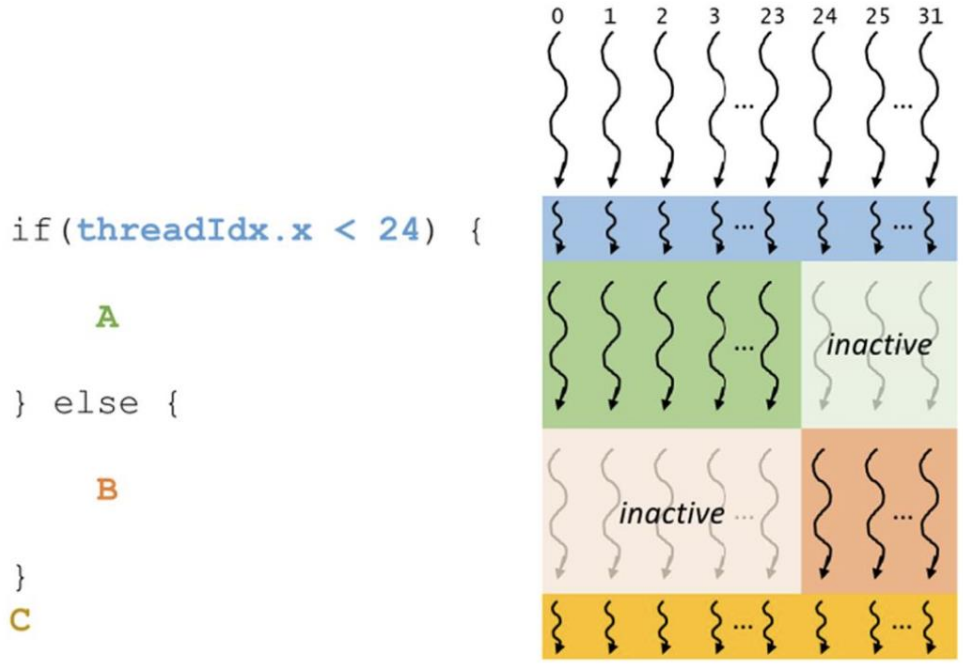

FIGURE 4.9 Example of a warp diverging at an if-else statement.

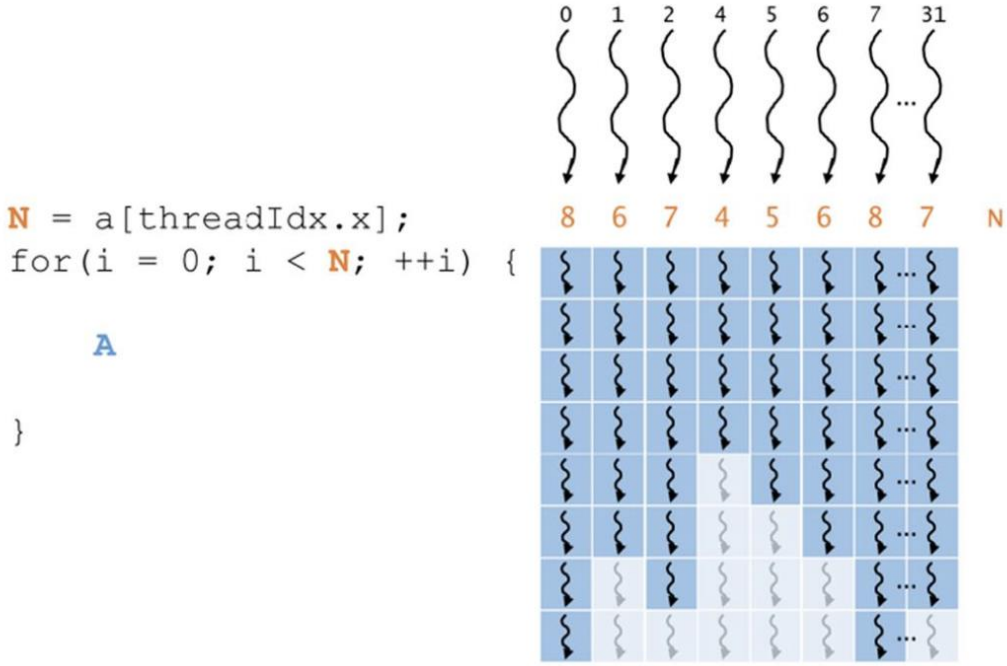

FIGURE 4.10 Example of a warp diverging at a for-loop.

Cost of divergence

- (within a warp) Branches will be **serialized**
- A common pattern is boundary check; e.g. launching 1024 threads to add 1000 size vector; so 1000 threads would be in active branch and the rest 24 in else branch (do nothing)
  in this case, is there cost?
- Control divergence only hurt performance within a warp. Threads in different warps can very well have diverging control and no performance penalty.

Warp scheduling

- There are usually more resident threads (warps) assigned to an SM than there are cores in the SM.
- When a warp is scheduled to execute a long latency instruction, another warp can be scheduled

- Having more warps than processing blocks help **hide latency**, by switching between the warps
- More warps available -> more opportunity to hide long latency e.g. A100, an SM has 64 cores but can be assigned up to 2048 threads
- This is why GPU does not need nearly as much cache/speculative execution/prefetching to deal with long latency

Threads & Context Switching

- A thread: code, the PC, and its variables and data structures.
- On the von Neumann model
  - PC: address of next instruction in memory
  - IR: the current instruction
  - Variables: in registers and memory
- Context switching
  - Suspend execution of thread A
  - Resume execution of thread B

- CPU context switching
  - Suspension and resuming threads involves saving PC & registers into memory, enter into kernel mode sched, and load PC & registers of the other thread.
  - Also will likely invalidate cache
  - 1,000s cycles to switch context
- GPU context switching
  - Zero-overhead (?), completely in hardware
  - This is **Fine-Grained Multi-Threading (FGMT)**
  - Cycle-by-cycle switching between warps
  - How? No need to save registers in memory and loading. By having a large register file!

Resource Partition & Occupancy

- It seems desirable to assign as many warps/threads to one SM as possible.
- Occupancy=$\frac{\#assigned\ warps}{\#maximum\ warps}$
- Maximum warps is a function of hardware; the actual assigned warps is a function of program and hardware

- Why can't occupancy be 1? Limited resources
  - # registers
  - Shared memory space
  - #block slots

Occupancy Examples

- A100 parameters: one SM max
  - 32 blocks
  - 64 warps (2048 threads)
  - 65,536 registers
  - 48KB shared memory
- Eg1: kernel: 32 threads per block, then occupancy<=0.5 because 32*32=1024 threads max
- Eg2: if 768 threads/block, then 2 blocks/SM->1546 threads, occupancy = 0.75
- Eg3: if a kernel uses 64 registers, then max threads = 65536/64=1024, occupancy <=0.5
- Eg4: each block uses 36KB share memory, and 256 threads/block, occupancy = 256/2048=0.125
- Calculator: https://xmartlabs.github.io/cuda-calculator/

#registers per thread

- # registers/thread is a function of your program.

```
$ nvcc solution.cu -I /opt/2025s_cosc4397/libgput
k -lgputk -L /opt/2025s_cosc4397/libgputk/lib --ptxas-options=-v -arch=sm_86
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z10blurKernelPfS_ii' for 'sm_86'
ptxas info    : Function properties for _Z10blurKernelPfS_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 34 registers, used 0 barriers, 376 bytes cmem[0]
```

- You can use -maxrregcount=xx to cap the #regsiters usage per thread, but it might come at severe cost of per thread performance—need to access global memory(!!) more.

Caveats of Occupancy

- It may seem the higher the occupancy the better
- It’s NOT!
- When occupancy is high enough, you don’t get higher performance by increasing it further.
- Typically, low occupancy -> suspicion of bad latency hiding behavior, idle functional units
- But don’t forget threads is only one form of parallelism! Instruction level parallelism is also one.

Summary

- A GPU is organized into SMs (more or less like CPU cores)
- SM consists of processing blocks (like CPU SIMD pipelines)
- When a kernel is launched, blocks are assigned to SM in arbitrary manner.
- Each block is assigned to one SM, and never switch out until done
- Warp: the unit of instruction scheduling, that bridges multiple threads with SIMD lanes
- Very fast context switches between warps is the primary way of GPU to hide long latency instructions