THREADS

Threads

- A thread is the fundamental building block of a parallel program.
- Parallelism in the CPU domain tends to be driven by the desire to run more than one (single-threaded) program on a single CPU. This is the **task-level parallelism(TLP).**
- Programs, which are data intensive, like video encoding, use the data parallelism model and split the task in N parts where N is the number of CPU cores available. Each CPU core calculate one “frame” of data where there are no interdependencies between frames**(DLP)**
- We can split each frame into N segments and allocate each one of the segments to an individual core.
- In the GPU domain, you see exactly these choices when attempting to speed up rendering of 3D worlds in computer games by using more than one GPU. You can send complete, alternate frames to each GPU.

Threads

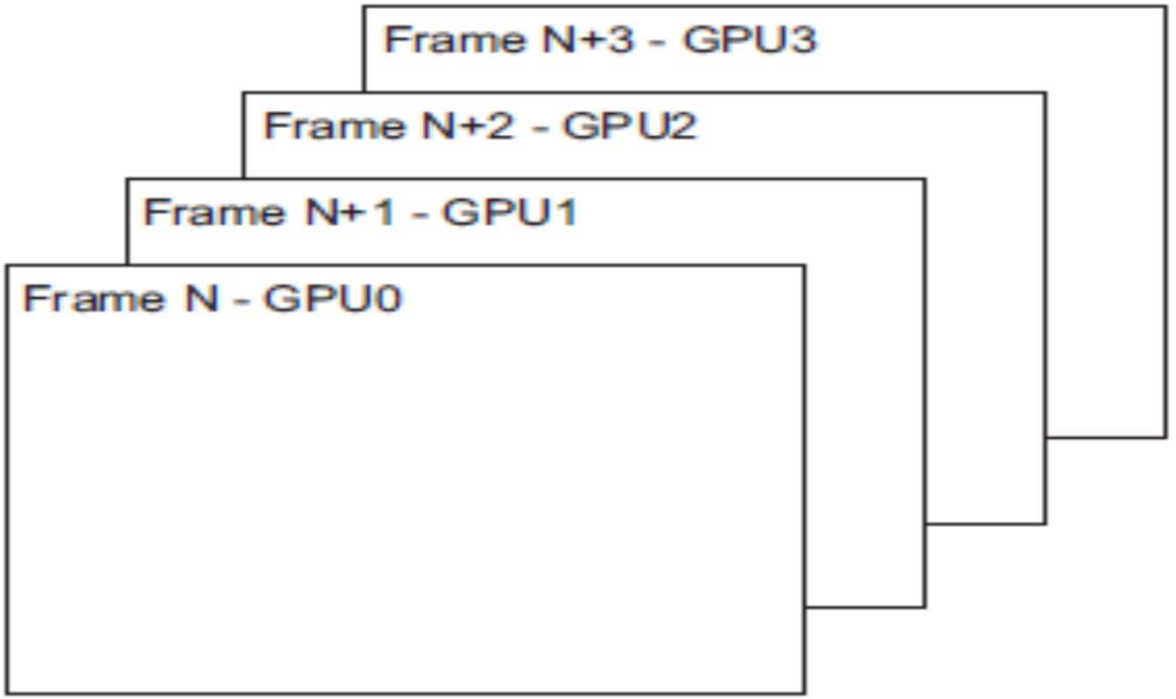


Fig: Alternate frame rendering (AFR) vs. Split Frame Rendering (SFR).

Threads

- **CPU**s support threads, but with a **large overhead** and thus are considered to be useful for more **coarse- grained** parallelism problems.
- **CPU**s, unlike **GPU**s, follow the **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata) model in that they support multiple independent instruction streams.
- Ex: a digital photo where you apply an image correction function to increase the brightness.
  - On a GPU you might choose to assign one thread for every pixel in the image.
  - On a quad-core CPU, you would likely assign one-quarter of the image to each CPU core.

Threads

- **CPUs Vs GPUs:**
- **GPU**s and **CPU**s are architecturally very different devices.
  - **CPU**s are designed for running a small number of potentially quite **complex tasks**.
  - **GPU**s are designed for running a large number of quite **simple tasks**.
- The **CPU** design is aimed at systems that execute a number of discrete and unconnected tasks.
- The **GPU** design is aimed at problems that can be broken down into thousands of tiny fragments and worked on individually.
- **CPU**s are very suitable for running operating systems and application software where there are a vast variety of tasks a computer may be performing at any given time

Threads

- **CPUs Vs GPUs:**

- CPUs and GPUs consequently support threads in very different ways.
  - **CPU** has a small number of registers per core that must be used to execute any given task.
  - To achieve this, they rapidly context switch between tasks.
  - Context switching on CPUs is **expensive** in terms of **time**, in that the entire register set must be saved to RAM and the next one restored from RAM.
  - **GPUs**, by comparison, also use the same concept of context switching, but instead of having a single set of registers, they have **multiple banks of registers**.
  - Consequently, a context switch simply involves setting a bank selector to switch in and out the current set of registers, which is several orders of magnitude faster than having to save to RAM.

Threads

- **CPUs Vs GPUs:**
- Both CPUs and GPUs must deal with stall conditions.
  - These are generally caused by I/O operations and memory fetches.
  - The CPU does this by context switching.
  - As the number of threads increases, the percentage of time spent context switching becomes increasingly large and the efficiency starts to rapidly drop off.
  - GPUs are designed to handle stall conditions and expect this to happen with high frequency.
  - The GPU model is a data-parallel one and thus it needs thousands of threads to work efficiently.
  - when it hits a memory fetch operation or has to wait on the result of a calculation, the streaming processors simply switch to another instruction stream and return to the stalled instruction stream sometime later.

Threads

- **CPUs Vs GPUs**
- One of the major differences between CPUs and GPUs is the sheer number of processors on each device.
- CPUs are typically dual- or quad-core devices. That is to say they have a number of execution cores available to run programs on.
- The current Fermi GPUs have 16 SMs, which can be thought of a lot like CPU cores.
- CPUs often run single-thread programs, meaning they calculate just a single data point per core, per iteration.
- GPUs run in parallel by default. Thus, instead of calculating just a single data point per SM, GPUs calculate 32 per SM.
- This gives a 4 times advantage in terms of number of cores (SMs) over a typical quad core CPU, but also a 32 times advantage in terms of data throughput..

Threading on GPUs

- look at a section of code and see what this means from a programming perspective.

```
void some_func(void)
{
        int i;
        for (i=0;i<128;i++)
        {
                a[i] = b[i] * c[i];
        }
}
```

- It stores the result of a multiplication of b and c value for a given index in the result variable a for that same index. The for loop iterates 128 times (indexes 0 to 127).
- In CUDA you could translate this to 128 threads, each of which executes the line

```
a[i] = b[i] * c[i];
```

Threading on GPUs

- There is no dependency between one iteration of the loop .
- we can transform this into a parallel program easily.
- This is called loop parallelization and is very much the basis for one of the more popular parallel language extensions, OpenMP.
- On a quad-core CPU you could also translate this to four blocks, where CPU core 1 handles indexes 0–31,

  core 2 handles indexes 32–63,

  core 3 handles indexes 64–95,

  & core 4 handles indexes 96–127.

Threading on GPUs

- In CUDA, translate this loop by creating a kernel function, which is a function that executes on the GPU.
- In CUDA the CPU handles the serial code execution which is where it excels.
- When you come to a computationally intense section of code the CPU hands it over to the GPU.
- Applications that used a large amount of floating-point math ran many times faster on machines fitted with such coprocessors.
- GPUs are used to accelerate computationally intensive sections of a program

Threading on GPUs

- The GPU kernel function:

```
__global__ void some_kernel_func(int * const a, const int * const b, const int *
const c)
        {
                a[i] = b[i] * c[i];
        }
```

- __global__ prefix added to the C function that tells the compiler to generate GPU code and not CPU code when compiling this function, and to make that GPU code globally visible from within the CPU.

Threading on GPUs

- The CPU and GPU have separate memory spaces, meaning you cannot access CPU parameters in the GPU code and vice versa.
- The global arrays a, b, and c at the CPU level are no longer visible on the GPU level.
- You have to declare memory space on the GPU, copy over the arrays from the CPU, and pass the kernel function pointers to the GPU memory space to both read and write from.
- CUDA provides a special parameter, different for each thread, which defines the thread ID or number.
- You can use this to directly index into the array. This is very similar to MPI, where you get the process rank for each process.
- The thread information is provided in a structure i.e thread_idx .

Threading on GPUs

```
__global__ void some_kernel_func(int * const a, const int * const b, const int * const c)
        {
                const unsigned int thread_idx = threadIdx.x;
                a[thread_idx] =  b[thread_idx] * c[thread_idx];
        }
```

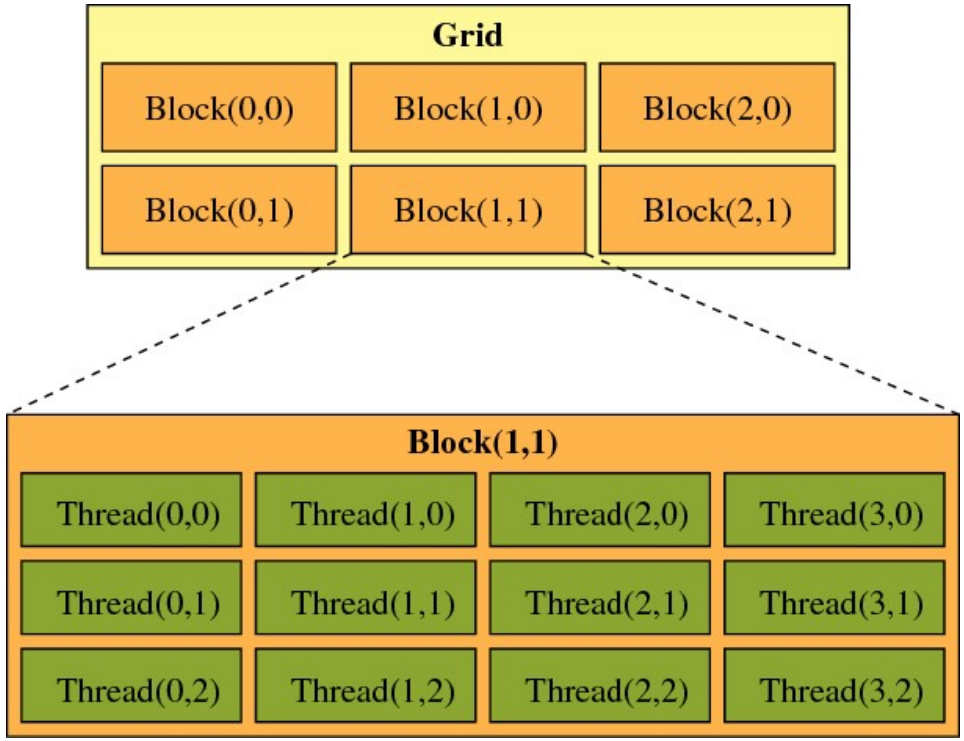
- Each thread does exactly two reads from memory, one multiply and one store operation, and then terminates.
- The code executed by each thread is identical but the data changes.
- This is at the heart of the CUDA and SPMD model.

Threading on GPUs

- Threads are grouped into 32 thread groups and these groups of threads is a **warp** (32 threads) and a **half warp** (16 threads).
- The 128 threads translate into four groups of 32 threads.
- The first set all run together to extract the thread ID and then calculate the address in the arrays and issue a memory fetch request

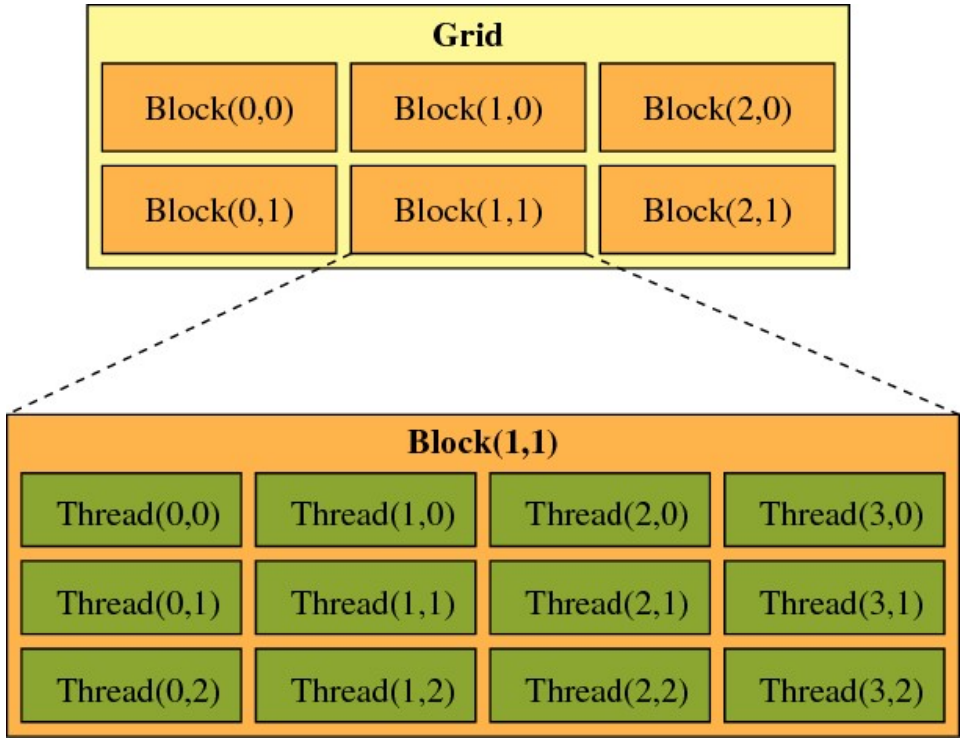
Thread Hierarchy

- A kernel executes in parallel across a set of parallel threads
- All threads that are generated by a kernel launch are collectively called a grid
- Threads are organized in thread blocks, and blocks are organized in to grids



Thread Hierarchy

- A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory
- A grid is an array of thread blocks that execute the same kernel
  - Read inputs to and write results to
    global memory
  - Synchronize between dependent kernel calls



Dimension and Index Variables

Dimension

- `gridDim` specifies the number of blocks in the grid
- `blockDim` specifies the number of threads in each block

Index

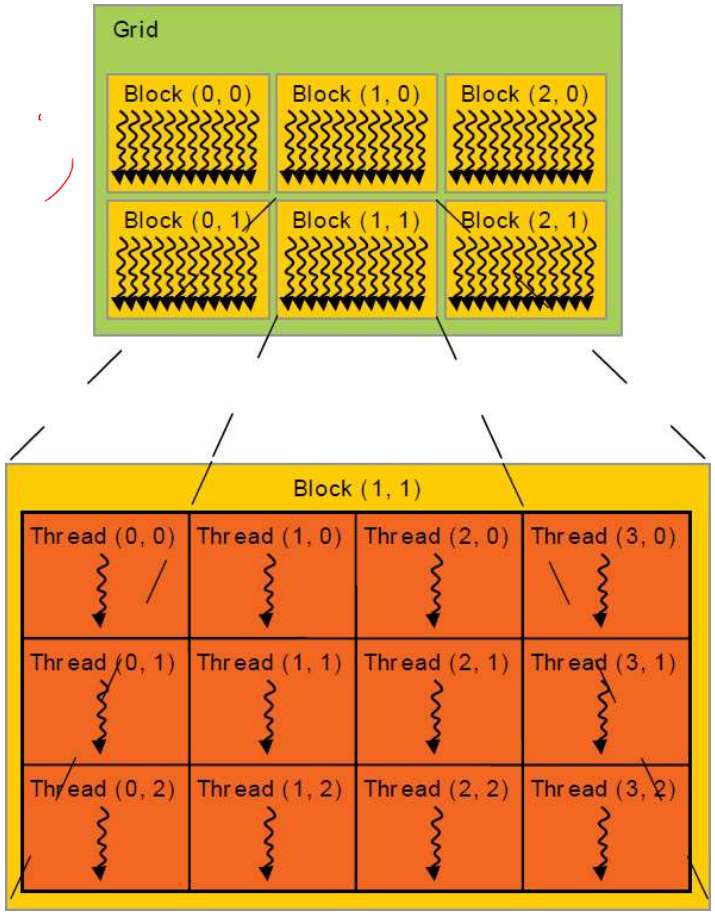
- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

Thread Hierarchy

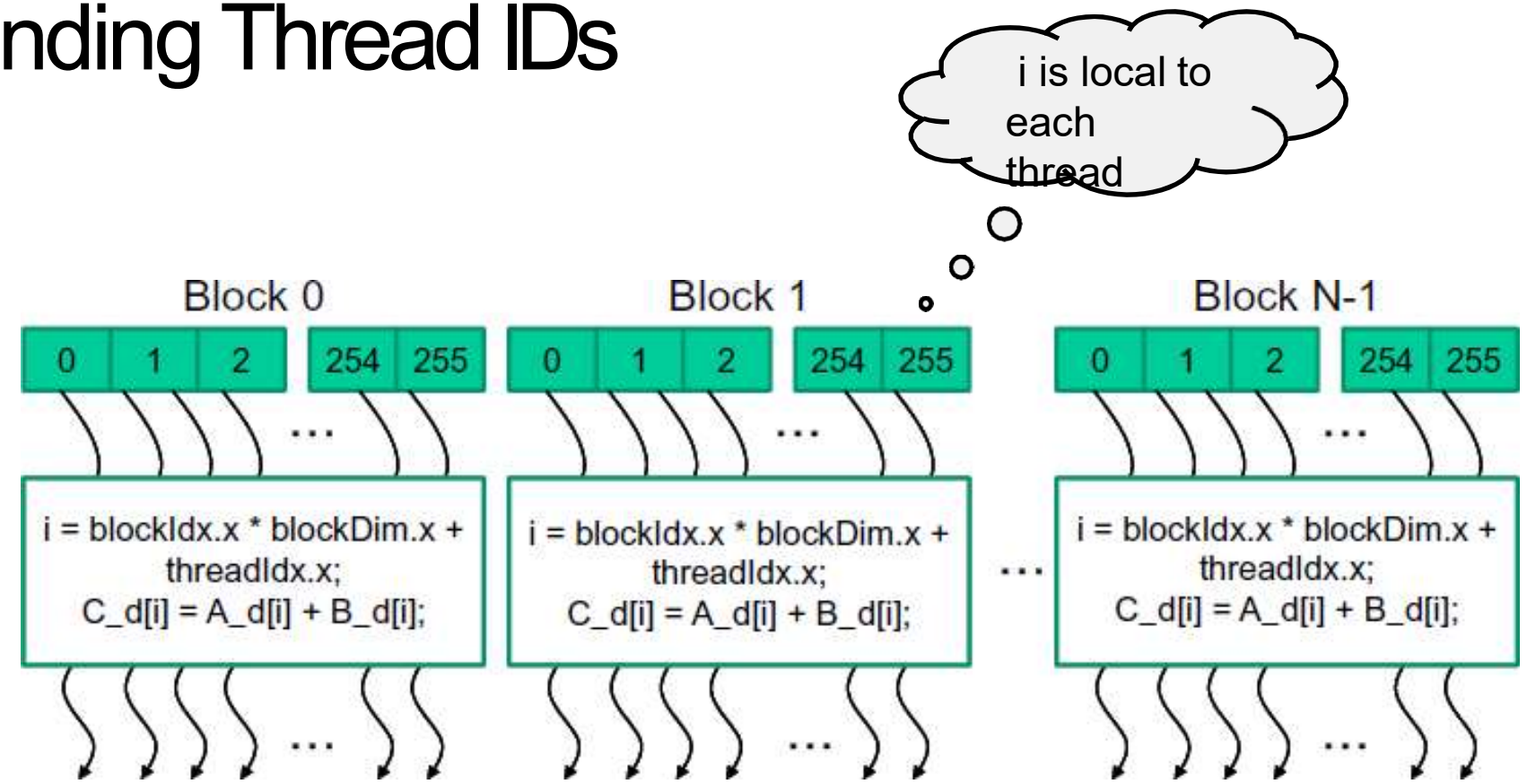
- `threadIdx` is a 3-component vector
  - Thread index can be 1D, 2D, or 3D
  - Thread blocks as a result can be 1D, 2D, or 3D
- How to find out the relation between thread ids and `threadIdx`?
  - 1D: `tid = threadIdx.x`
  - 2D block of size `(Dx, Dy)`: thread ID of a thread of index `(x, y)` is `(x + yDx)`
  - 3D block of size `(Dx, Dy, Dz)`: thread ID of a thread of index `(x, y, z)` is `(x + yDx + zDxDy)`

Thread Hierarchy

- Threads in a block reside on the same core, max 1024 threads in a block
- Thread blocks are organized into 1D, 2D, or 3D grids
  - Also called cooperative thread array
  - Grid dimension is given by `gridDim` variable
- Identify block within a grid with the `blockIdx` variable
  - Block dimension is given by `blockDim` variable



Finding Thread IDs



Threading on GPUs

- Fetches from consecutive threads are grouped together.
- This reduces the overall latency
- As a result of the grouping, the memory fetch returns with the data for a whole group of threads
- This will enable an entire warp.
- These threads are placed in the ready state and become available for the GPU
- Having executed all the warps (groups of 32 threads) the GPU becomes idle waiting for any one of the pending memory accesses to complete.

Threading on GPUs

- CUDA kernels:

- A kernel is just a name for a function that executes on the GPU. To invoke a kernel you use the following syntax:
- kernel_function<<<num_blocks, num_threads>>>(param1, param2, .)
- The parameters : **num_blocks** and **num_threads**. These can be either variables or literal values.
- The **num_block*s*** parameter is no of blocks you launch or ensure you have at least one block of threads.
- The **num_threads** parameter is simply the number of threads you wish to launch into the kernel.

Mapping Blocks and Threads

- A GPU executes one or more kernel grids
- When a CUDA kernel is launched, the thread blocks are enumerated and distributed to SMs
  - Potentially >1 block per SM
- An SM executes one or more thread blocks
  - Each GPU has a limit on the number of blocks that can be assigned to each SM
  - For example, a CUDA device may allow up to eight blocks to be assigned to each SM
  - Multiple thread blocks can execute concurrently on one SM

Mapping Blocks and Threads

- The threads of a block execute concurrently on one SM
  - CUDA cores in the SM execute threads
- A block begins execution only when it has secured all execution resources necessary for all the threads
- As thread blocks terminate, new blocks are launched on the vacated multiprocessors
- Blocks are mostly not supposed to synchronize with each other
  - Allows for simple hardware support for data parallelism

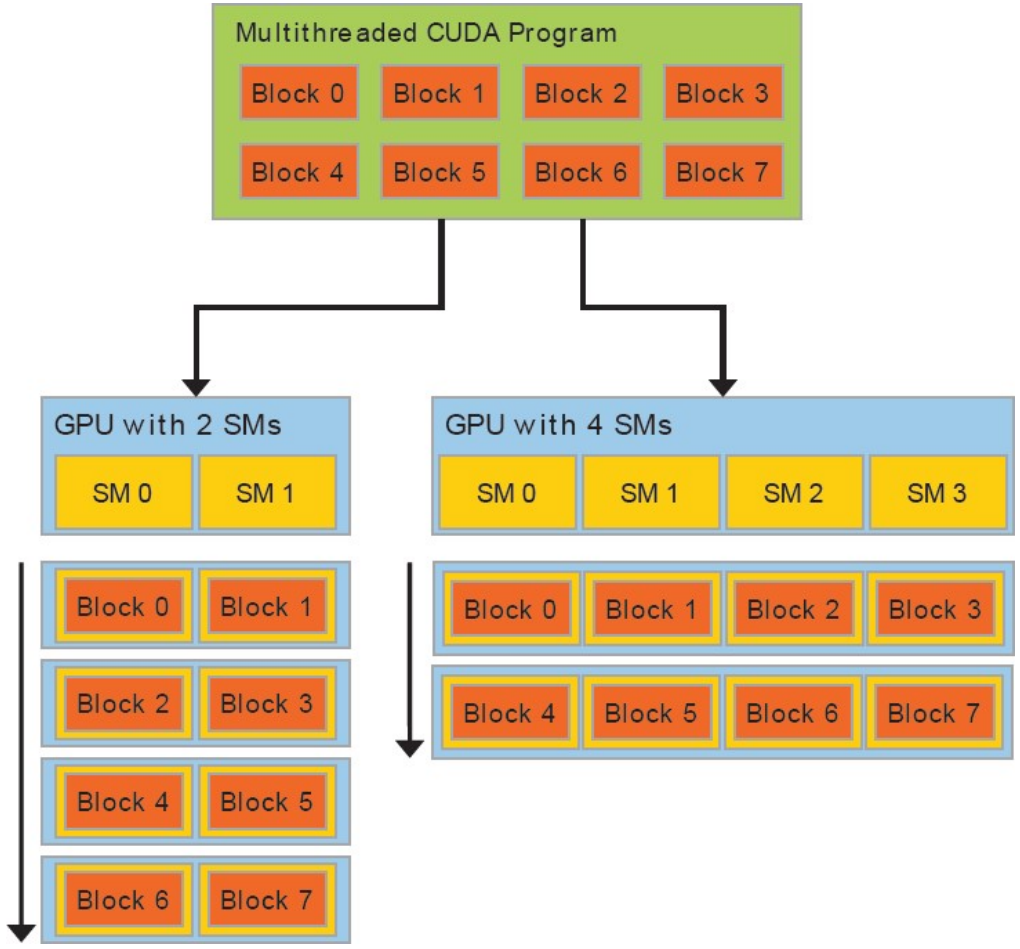
Scheduling Blocks

- Number of threads that can be simultaneously tracked and scheduled is bounded
  - Requires resources for an SM to maintain block and thread indices and their execution status
- Up to 2048 threads can be assigned to each SM on recent CUDA devices
  - For example, 8 blocks of 256 threads, or 4 blocks of 512 threads
- Assume a CUDA device with 28 SMs
  - Each SM can accommodate up to 2048 threads
  - The device can have up to 57344 threads simultaneously residing in the device for execution

Scalability of GPU Architecture

A multithreaded program is partitioned into blocks of threads that execute independently from each other.

A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.



Thread Warps

- Conceptually, threads in a block can execute in any order
- Sharing a control unit among compute units reduce hardware complexity, cost, and power consumption
- A set of consecutive threads (currently 32) that execute in SIMD fashion is called a **warp**
  - These are called **wavefront** (with 64 threads) on **AMD**
- **Warps** are scheduling units in an **SM**
  - Part of the implementation in NVIDIA, not the programming model

Thread Warps

- All threads in a warp run in lockstep
  - Warps share an instruction stream
  - Same instruction is fetched for all threads in a warp during the instruction fetch cycle
    - Prior to Volta, warps used a single shared program counter
  - In the execution phase, each thread will either execute the instruction or will execute nothing
  - Individual threads in a warp have their own instruction address counter and register state

Thread Warps

- Warp threads are fully synchronized
  - There is an implicit barrier after each step/instruction

- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
  - Each Block is divided into 256/32 = 8 warps
  - There are 8 * 3 = 24 warps

Thread Divergence

- If some threads take the if branch and other threads take the else branch, they cannot operate in lockstep
  - Some threads must wait for the others to execute
  - Renders code at that point to be serial rather than parallel
- Divergence occurs only within a warp
- The programming model does not prevent thread divergence
  - Performance problem at the warp level