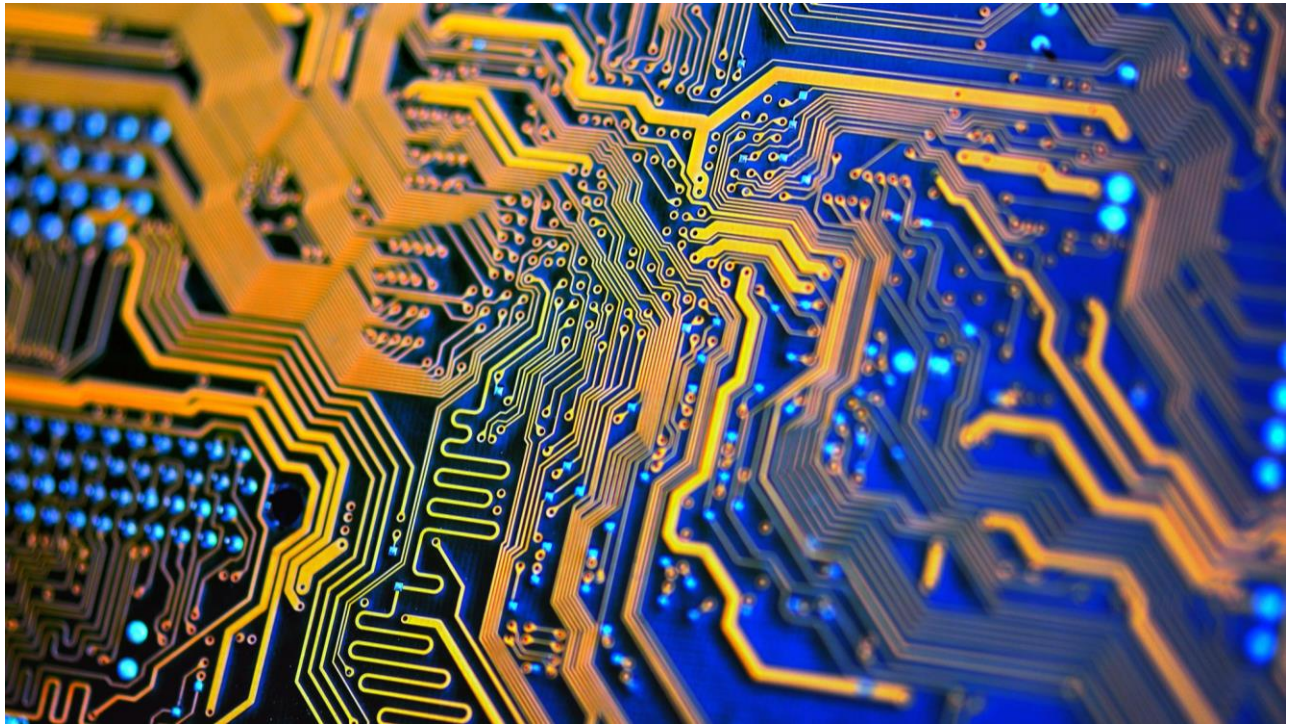
CS250: Foundation of Computer Systems

[Graphics Processing Units GPUs]



Computer Science
Colorado State University

** Lecture slides created by: Shrideep Pallickara

Topics covered in this lecture

- GPUs wrap-up

How GPUs and CPUs are different

How GPUs and CPUs differ [1/2]

- CPUs are very suitable for running
  - Operating systems
  - Application software
- On CPUs there are a **vast variety of tasks** a computer may be performing at any given time

How GPUs and CPUs differ [2/2]

- CPUs are designed for running a *small number* of potentially quite **complex tasks**
  - GPUs are designed for running a *large number* of quite **simple tasks**
- The CPU design is aimed at systems that execute several **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

CPUs and GPUs support threads in very different ways [1/2]

- The **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
  - On CPUs, *context switching is expensive* in terms of time
    - The entire register set must be saved to RAM and the next one restored from RAM

CPUs and GPUs support threads in very different ways [2/2]

- GPUs also use the same concept of context switching
- But instead of having a single set of registers, they have **multiple banks of registers**
  - 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

Both CPUs and GPUs must deal with stall conditions [1/2]

- Stalls are generally caused by I/O operations and memory fetches
- The CPU does this by **context switching**
  - Providing there are enough tasks, and the runtime of a thread is not too small, this works reasonably well
  - If there are not enough processes to keep the CPU busy, it will idle
  - If there are too many small tasks, each blocking after a short period?
    - The CPU will spend most of its time context switching; very little time doing work
    - CPU scheduling policies are often based on time slicing
    - As the number of threads increases, the percentage of time spent context switching becomes increasingly large and the efficiency starts to rapidly drop off

Both CPUs and GPUs must deal with stall conditions [2/2]

- GPUs are designed to handle stall conditions and **expect this to happen with high frequency**
- The GPU model is a data-parallel one and needs thousands of threads to work efficiently
- GPUs uses this pool of available work to ensure it always has something useful to work on
  - Thus, when it hits a memory fetch or must wait on a computation result?
    - The SPs simply switch to another instruction stream and return to the stalled instruction stream sometime later

GPUs also provide something quite unique

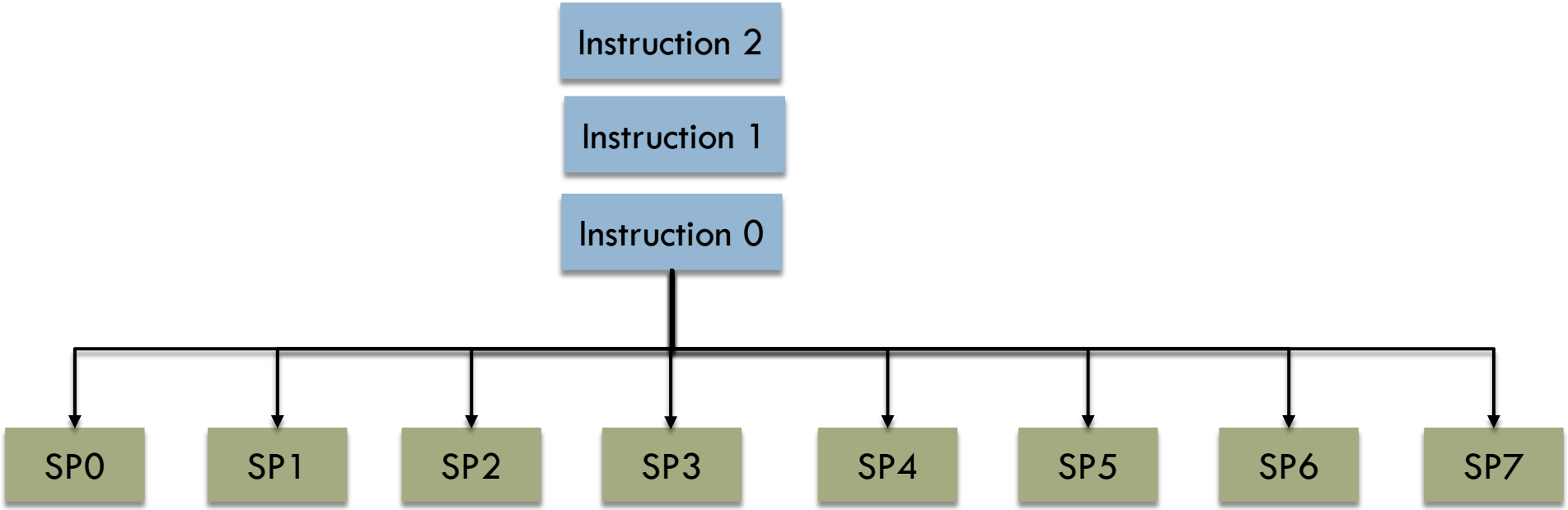
- High-speed memory next to the SM, so-called **shared memory**
- Programmer can leave data in this shared memory
  - Knowing that hardware will not evict it behind programmer's back
- This shared memory is also the primary mechanism communication between threads

The GPU's task execution model differs in two key ways

- Groups of N SPs execute in a **lock-step basis** running the same program but on different data
- The second is that, because of the **huge register file**
  - Switching threads has effectively zero overhead
  - As a result, GPUs can support a very large number of threads

Lock-step instruction dispatch in GPUs

- Each instruction in the instruction queue is **dispatched to every SP within an SM**



COMING BACK TO THREADS IN GPUS

Let's look at a section of code and see what this means from a programming perspective [1/2]

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

- Stores the result of a multiplication of **b** and **c** value for a given index in the result variable **a** for that same index
- Loop iterates 128 times (indexes 0 to 127)

Let's look at a section of code and see what this means from a programming perspective [2/2]

- In CUDA you could translate this to 128 threads
  - Each of which executes the line a[i] = b[i] * c[i]
  - Possible because there is no dependency between one iteration of the loop and the next
    - Transformation into a parallel program is rather easy
- In CUDA, we create a kernel function
  - A function that executes on the GPU only and not on the CPU

The GPU kernel function looks identical to the loop body, but with the loop structure removed

- Thus, you have the following:
  - __global__ void
    some_kernel_func(int * const a, const int * const b, const int * const c)
    { a[i] = b[i] * c[i]; }
- Notice
  - You have lost the loop and the loop control variable, i
  - You also have a __global__ prefix added to the C function that
    - Tells the compiler to generate GPU code and not CPU code when compiling
    - And to make that GPU code globally visible from within the CPU

Threads are, in practice, grouped into sets of 32

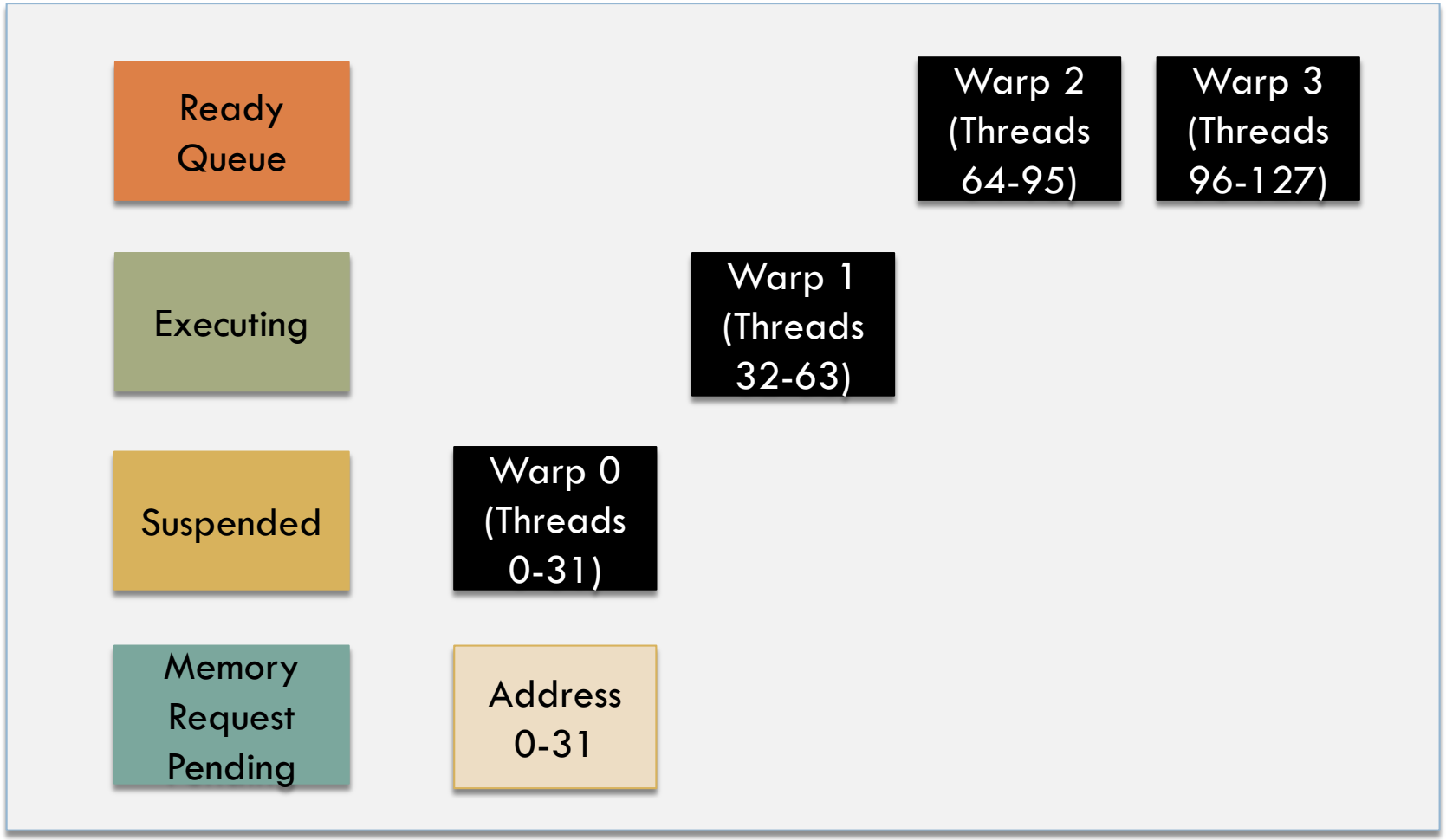
- When the threads are waiting on something such as memory access, they are *all* suspended
- The technical term for these groups of threads is a **warp** (32 threads) and a half warp (16 threads)

Threads are, in practice, grouped into sets of 32

- 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
  - So, the threads are suspended
  - When all 32 threads in that block of 32 threads are suspended?
    - The hardware switches to another warp

Depicting this execution

When Warp-0 is suspended, Warp-1 becomes the executing warp



Prior to issuing the memory fetch

- Fetches from consecutive threads are usually **coalesced** or grouped together
- Reduces the overall latency (time to respond to the request), as there is an overhead associated in the hardware with managing each request
- As a result of the coalescing:
  - Memory fetch returns with the data for a whole group of threads
    - Usually enough to enable an entire warp
  - These threads are then placed in the ready state and become available for the GPU to switch in the next time it hits a blocking operation
  - Upon having executed all the warps (groups of 32 threads) the GPU becomes idle

Now let's look a little more at how exactly you invoke a kernel

- CUDA defines an **extension to the C language** used to invoke a kernel
- To invoke a kernel, you use the following syntax:
  `kernel_function<<<num_blocks, num_threads>>>(param1, param2, …)`
  - The `num_blocks` parameter – there should be at least 1 block of threads
  - The `num_threads` parameter is simply the number of threads you wish to launch into the kernel
    - For our simple example, this directly translates to the number of loop iterations
    - However, be aware that the hardware limits you to 512 threads per block on the early hardware and 1024 on the later hardware

Now let's look a little more at how exactly you invoke a kernel

- Parameters can be passed via registers or constant memory
  - Choice is based on the compilers
- If using registers: one register for every thread per parameter passed.
  - Thus, for 128 threads with three parameters, we use $3 \times 128 = 384$ registers
  - This may sound like a lot but remember that you have at least 8192 registers in each SM and potentially more on later hardware revisions.
    - So, with 128 threads, you have a total of 64 registers (8192 registers ÷ 128 threads) available to you, if you run just one block of threads on an SM

However, running one block of 128 threads per SM is a very bad idea

- Even if you can use 64 registers per thread
- As soon as you access memory, the SM would effectively idle
- Only in the very limited case of **heavy arithmetic intensity** utilizing the 64 registers should you even consider this sort of approach
- In practice, multiple blocks are run on each SM to avoid any idle states

Threads: Grids, Blocks, & Warps

Grids, blocks, warps

- At the heart of parallel programming is the idea of a thread
  - A single flow of execution through the program in the same way a piece of cotton flows through a garment
- Just as threads of cotton are woven into cloth, threads used together make up a parallel program
  - The CUDA programming model groups threads into special groups it calls warps, blocks, and grids

Warps

- **Warps** are the basic unit of execution on the GPU
- Each group of threads, or warps, is executed together
  - Typically, only *one fetch from memory* for the current instruction and a broadcast of that instruction to the entire set of SPs in the warp
  - This is much more efficient than the CPU model, which fetches independent execution streams to support task-level parallelism

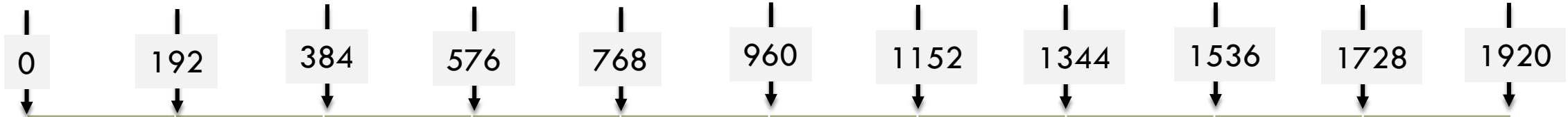
Grids

- A grid is simply a **set of blocks** where you have an X and a Y axis, in effect a 2D mapping
  - The final Y mapping gives you Y × X × T possibilities for a thread index
- The number of threads in a block should always be a multiple of the warp size (currently 32)
  - You can only schedule a full warp on the hardware, if you don't do this, then the remaining part of the warp goes unused
- To avoid poor memory coalescing, you should always try to arrange the memory and thread usage so they map
  - Failure to do so will result in something in the order of a **5X drop** in performance

Grids

- If you were to look at a typical HD image, you have a 1920 × 1080 resolution
- We avoid tiny blocks, as they don't make full use of the hardware; we'll pick 192 threads per block
  - Typically, this is the minimum number of threads you should think about using
  - This gives you exactly 10 blocks across each row of the image

HD Image and Block allocations to rows [1/2]



Row 0	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8	Block 9
Row 1	Block 10	Block 11	Block 12	Block 13	Block 14	Block 15	Block 16	Block 17	Block 18	Block 19
Row 2	Block 20	Block 21	Block 22	Block 23	Block 24	Block 25	Block 26	Block 27	Block 28	Block 29
Row …										
Row 1079	Block 10790	Block 10791	Block 10792	Block 10793	Block 10794	Block 10795	Block 10796	Block 10797	Block 10798	Block 10799

HD Image and Block allocations to rows [2/2]

- Along the top on the X axis, you have the **thread index**
- The row index forms the Y axis; the row height is exactly one pixel
- With1080 rows of 10 blocks, we have $1080 \times 10 = 10{,}800$ blocks
  - Since each block has 192 threads, you are scheduling just over two million threads, one for each pixel
  - This particular layout is useful where you have
    - One operation on a single pixel or data point, or
    - Some operation on a number of data points in the same row

This is all very well, but what if your data is not row based?

- As with arrays, you are not limited to a single dimension
- You can have a 2D thread block arrangement

The contents of this slide-set are based on the following references

- Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (Applications of GPU Computing)*. ISBN-10/ISBN-13: 0124159338/978-0124159334. 1st Edition. Morgan Kaufmann. [Chapters 3, 4, 5]