CS250: Foundations of Computer Systems [GPUs]

A retrospective. As the sun sets on the semester …

The shadows lengthen
and wind through
representations binary, hexa, and octal

With numbers floating and signed
mantissas and exponents
working in tandem
representing things … miniscule and gargantuan

Logic crunching through gates
powered by electrons in motion
synthesizing functions from truth tables
in-silico, our one-man band, Nand

Data ensconced along for a ride
over multiplexed, circuit-switched networks
riding ether, glass, and copper

Using sockets, routers, and protocol stacks
fragmented *en route* to destinations
coalesced finally reliably and in order

Trees that balance
on their leaves
powering indexes signposts on steroids
without accesses mired in the I/O quicksand

of CPUs, GPUs, and their love for speed
low power and high throughputs
crunching through tasks and data

Here's to your journey
through computing systems
In silico, abstractions, and software
the road to everywhere

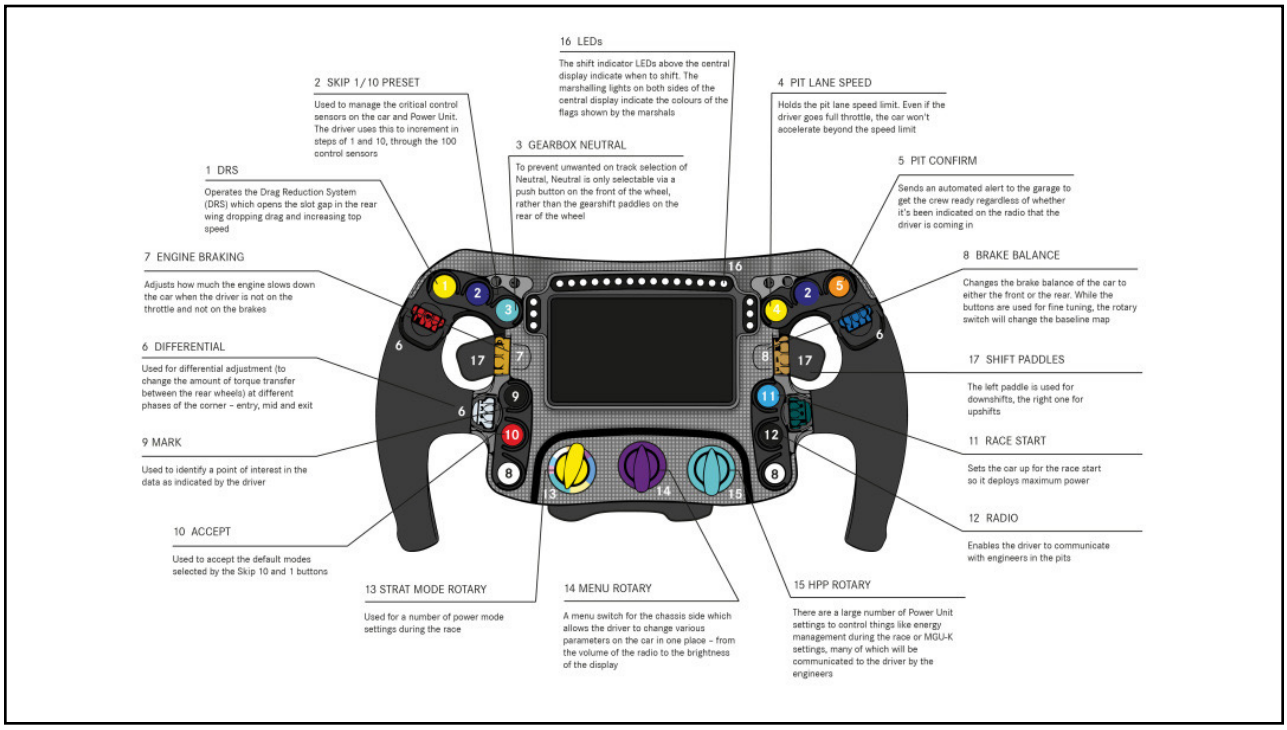
Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- Will a coordinator core help with "scaling" coherency issues in a CPU?
- Can SISD be thought of as the simplest implementation of SIMD?
- Why do GPUs need more work from the programmer? Why not give it the "self-done" abilities of a CPU?

2



3

Topics covered in this lecture

- GPUs wrap-up
- Final Exam

4

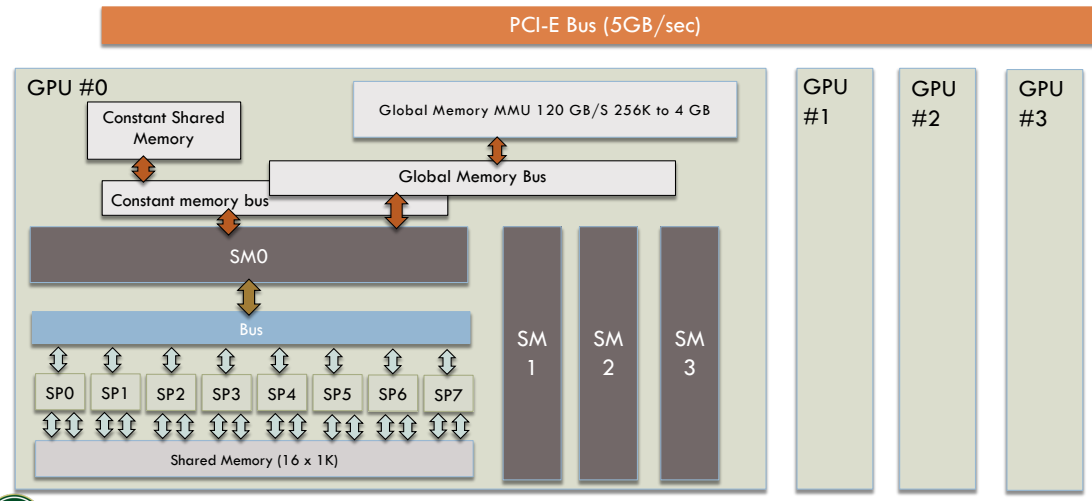
The GPU hardware

- The GPU hardware consists of a number of key blocks:
  - Memory (global, constant, shared)
  - Streaming multiprocessors (SMs)
  - Streaming processors (SPs)

GPUs and SMs

- A GPU device consists of one or more SMs
- Add more SMs to the device and you make the GPU able to
  - Process **more tasks** at the same time, or
  - Process the **same task quicker**, if you have enough parallelism in the task

Overview of a GPU



Taking a closer look at the SMs

- There are multiple SPs in each SM
  - Our diagram showed 8
  - Older architectures: In Fermi this grows to 32–48 SPs and in Kepler up to 192
- Modern NVIDIA architectures:
  - Ampere A100: 64 FP32 CUDA cores per SM
  - Ampere GA10x / RTX 30-series: 128 CUDA cores per SM
  - Ada Lovelace / RTX 40-series: 128 CUDA cores per SM
  - Hopper H100: 128 FP32 CUDA cores per SM
  - RTX Blackwell / RTX 50-series: 128 FP32 CUDA cores per SM
    - Flagship: 192 SMs for a total of 24,576 CUDA cores
- Each hardware revision increases both the number of SMs and the number of SPs (in each SM)

SMs and memory [1/3]

- Each SM has access to something called a **register file**
  - A **chunk of memory** that runs at the *same speed* as the SP units, so there is effectively zero wait time on this memory
  - Used for storing the registers in use within the threads running on an SP
- There is also a shared memory block accessible only to the individual SM; this can be used as a **program-managed cache**
  - Unlike a CPU cache, hardware does not evict data from the cache behind your back
    - Entirely under programmer control

SMs and memory [2/3]

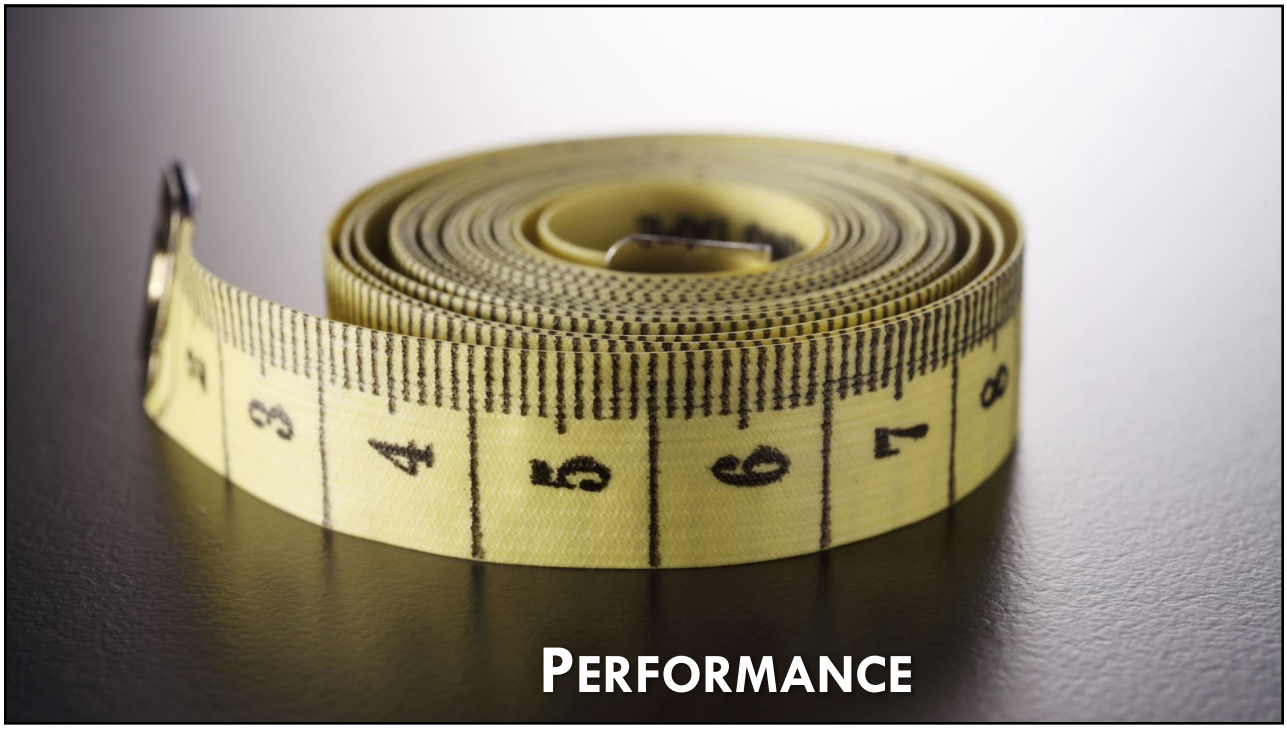
- Each SM has a **separate bus** into the texture memory, constant memory, and global memory spaces
- **Texture memory** is a special view onto the global memory
  - Useful for data needing interpolation; for e.g., with 2D/3D lookup tables
  - Special feature of hardware-based interpolation
- **Constant memory** is used for read-only data
  - Like texture memory, constant memory is simply a view into the main global memory

SMs and memory [3/3]

- Global memory is supplied via GDDR (Graphic Double Data Rate) on the graphics card
  - A high-performance version of DDR (Double Data Rate) memory
- Memory bus width can be up to **512 bits wide**, giving a bandwidth of 5 to 10 times more than found on CPUs
  - Up to 190 GB/s with the Fermi hardware
  - The NVIDIA Blackwell architecture features an extremely wide memory interface designed to handle high-bandwidth AI workloads
    - The flagship B200 GPU utilizes a 4096-bit

Where to? From here …

- Now we have all the pieces:
  - SMs do the work, registers and shared memory keep data close, global memory is large but slower, and warps hide waiting
- Performance comes from making all of these **cooperate**



13

From hardware to programming discipline

- GPUs have many SMs, SPs, registers, and memory spaces
- But hardware alone does not guarantee speed
- GPU performance depends on how the programmer exposes:
  - **Concurrency**: enough *independent* work
  - **Locality**: data *close* to where it is used
  - **Regularity**: threads doing *similar* things together

14

Achieving peak performance [1/2]

- The approach CUDA uses with all problems is to require the programmer to **break the problem into smaller parts**
- Most parallel approaches make use of this concept in one way or another
- Even in huge supercomputers, problems such climate models must be broken down into hundreds of thousands of blocks
  - Each of which is then allocated to one of 1000s of processing elements
- This type of *parallel decomposition* has the huge advantage that it **scales** really well

Achieving peak performance [2/2]

- While peak performance on GPUs may be impressive, achieving anything like this is **not possible without specially crafted programs**
- The reported peak performance often does not include things such as memory access, which is somewhat key to any real program
- To achieve good performance on any platform requires a good knowledge of hardware and understanding two key concepts
  - Concurrency
  - Locality

So how do we recognize GPU-friendly code?

- We start with the most familiar structure in programming
  - The loop!

A GPU question: Can this loop be split? [1/2]

- We are all very familiar with loops
  - They vary primarily in terms of entry and exit conditions (`for`, `do…while`, `while`), and whether they create dependencies between loop iterations or not
- A loop-based **iteration dependency** is where one iteration of the loop depends on one or more previous iterations.
  - We want to remove these (if possible) as they make implementing parallel algorithms more difficult
  - If in fact this can't be done, the loop is typically broken into a number of blocks that are executed in parallel
    - The result from block 0 is then *retrospectively* applied to block 1, then to block 2, and so on

A GPU question: Can this loop be split? [2/2]

- Loop-based iteration is one of the easiest patterns to parallelize
- With inter-loop dependencies removed?
  - It's then simply a matter of deciding how to split (or partition) the work between the available processors

Divide and conquer pattern [1/2]

- The **divide-and-conquer** pattern is also a pattern for breaking down large problems into smaller sections, each of which can be conquered
- Taken together these individual computations allow a much larger problem to be solved
- Typically, you see divide-and-conquer algorithms used with recursion
  - Quick sort is a classic example of this.
    - It recursively partitions the data into two sets, those above a pivot point and those below the pivot point
    - When the partition finally consists of just two items, they are compared and swapped

Divide and conquer pattern [2/2]

- Most *recursive* algorithms can also be represented as an *iterative* one
- **Iteration is usually easier** to map onto the GPU
  - Fits better into the primary tile-based decomposition model of the GPU

To iterate, or to recurse: That is the computation [1/2]

- In selecting a recursive algorithm be aware that you are making a **tradeoff** between
  - Development time versus
  - Performance
- A recursive algorithm may be *easier to conceptualize* (and therefore code) than to try to convert such an approach to an iterative one

To iterate, or to recurse: That is the computation [2/2]

- However, each recursive call causes any formal parameters (*i.e.*, arguments) to be pushed onto the stack along with any local variables
  - GPUs and CPUs implement a stack in the same way, simply an area of memory
- Use **iterative solutions where possible** as they will generally perform much better and run on a wider range of GPU hardware

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

But where did the `i` go?

- In the CPU version, `i` comes from the loop:
  - ```
    for (i = 0; i < 128; i++)
    ```
- In the GPU version, the loop is replaced by many threads
  - The **loop counter becomes the thread's identity**
- Each thread computes its *own* index using CUDA's built-in variables
  - Then each thread performs one piece of work

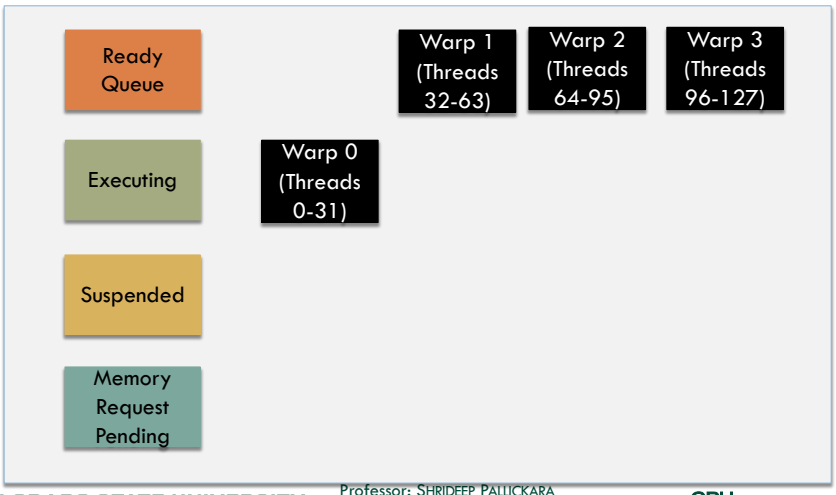
Threads are, in practice, grouped into sets of 32 [1/2]

- 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 [2/2]

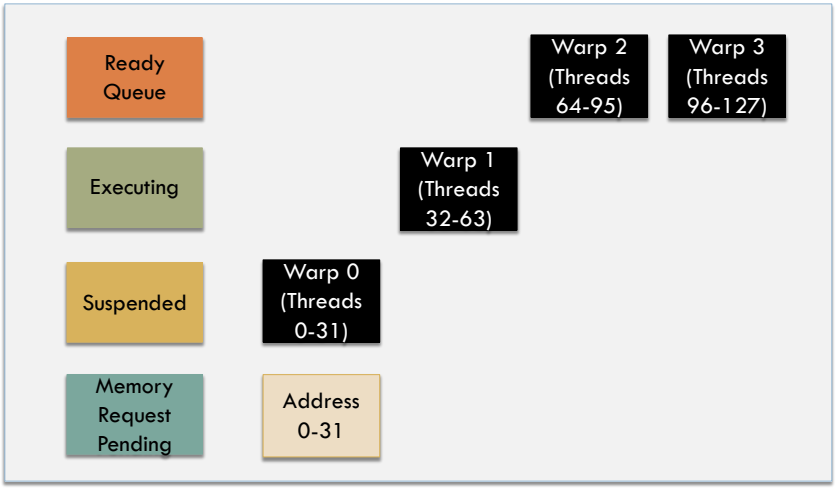
- 128 threads translate into four groups of 32 threads
- The first set all run together to extract the thread ID and then calculate the address in the arrays and issue a memory fetch request
  - So, the threads are suspended
  - When all 32 threads in that block of 32 threads are suspended?
    - The hardware switches to another warp

Depicting this execution



31

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



32

Prior to issuing the memory fetch

- Fetches from consecutive threads are usually **coalesced** or grouped together
  - Instead of issuing many separate memory requests, the hardware combines them
- 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 [1/2]

- 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 [2/2]

- Parameters can be passed via registers or constant memory
  - Choice is based on the compiler
- If using registers: one register for every thread per parameter passed
  - Thus, for 128 threads with three parameters, we use 3 × 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

An important concept: arithmetic intensity

- **Arithmetic intensity** means how much computation you do for each piece of data fetched from memory
  - Low arithmetic intensity: load two numbers, multiply once, store result
  - High arithmetic intensity: load data once, perform many calculations before storing result
- If you go all the way to the hardware store, try not to buy one screw!

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

Grids, blocks, warps

- At the heart of concurrent 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

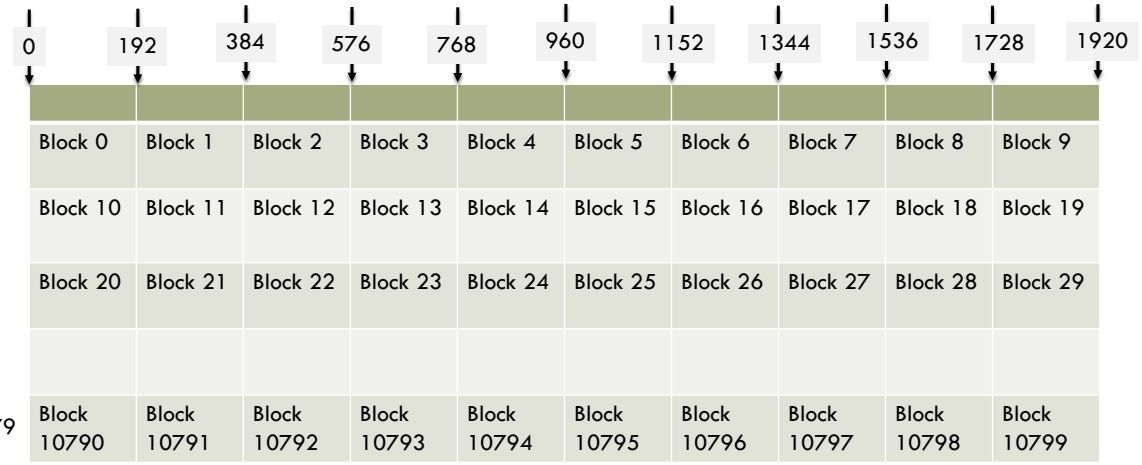
Grids

- 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]



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 × 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

The general pattern

- 1D data ⟶ 1D thread layout
  - Arrays, vectors, streams
- 2D data ⟶ 2D thread/block layout
  - Images, matrices, grids
- 3D data ⟶ 3D layouts
  - Volumes, simulations, spatial models
- The programmer's job is to map:
  - data shape ⟶ thread layout
  - memory access ⟶ coalesced access
  - repeated work ⟶ warps and blocks

The GPU memory is still ultimately linear.
The 2D/3D layout is a convenient way to assign threads to data.

What you should remember about GPUs

- GPUs are built for **throughput**, not single-thread latency
- They win when work can be split into many independent pieces
- Warps execute groups of threads together
- Memory access patterns matter enormously
- Shared memory, coalescing, and enough active warps help keep SMs busy
- The programmer's job is to expose:
  - Concurrency
  - Locality
  - Regularity

And finally …. a GPU is not a faster CPU

- It is a **different bargain**
- Give it thousands of simple, regular tasks, and it will repay you handsomely!

FINAL EXAM

Topic	Points
Number Representations: Binary representations and properties of binary numbers, Two's complement representations, binary arithmetic, single and double-precision floating point numbers. Hexadecimal numbers.	10
Boolean Logic, Boolean algebra, De Morgan's Laws, Synthesizing boolean functions from truth tables, the fundamental theorem of logic design and the expressive power of Nand gates.	10
Memory Subsystem: Speed differential of the memory hierarchy, Registers, Caches (L1, L2, and L3), and main Memory. Spatial and temporal locality of references. Different types of caches: direct-mapped, fully associative, and N-way associative cache. Memory addressing.	15
Networking and Communications: Data encoding formats, delay-bandwidth product, switched networks, internetworking, layering of protocols, encapsulation, fragmentation and routing of packets, IPv4/IPv6, UDP, TCP and sliding windows, and TCP optimizations. Private addresses and network address translation, and DNS systems.	20
von Neumann Architecture & Software : The stored program concept, the vonNeumann architecture, Moore's Law and Dennard Scaling, the Harvard architecture. Memory-mapped I/O, and Software (including machine language)	15
Storage Systems: Binary search, BST, Indexes, and B-Trees	20
Graphics Processing Units (GPUs)	10

49

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]

50