CS 250: Foundations of Computer Systems
[Graphics Processing Units GPUs]

XKCD

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- Why don't we use TPUs more prolifically throughout the AI ecosystem?
- Why is the NVIDIA 5090 series so much better than the 5060 series?
  - More CUDA cores: 21,000 to ~4,000
  - Memory bandwidth: 512-bit bus vs 128-bits
  - Geared toward 4/8K gaming vs HD gaming
- CUDA kernels

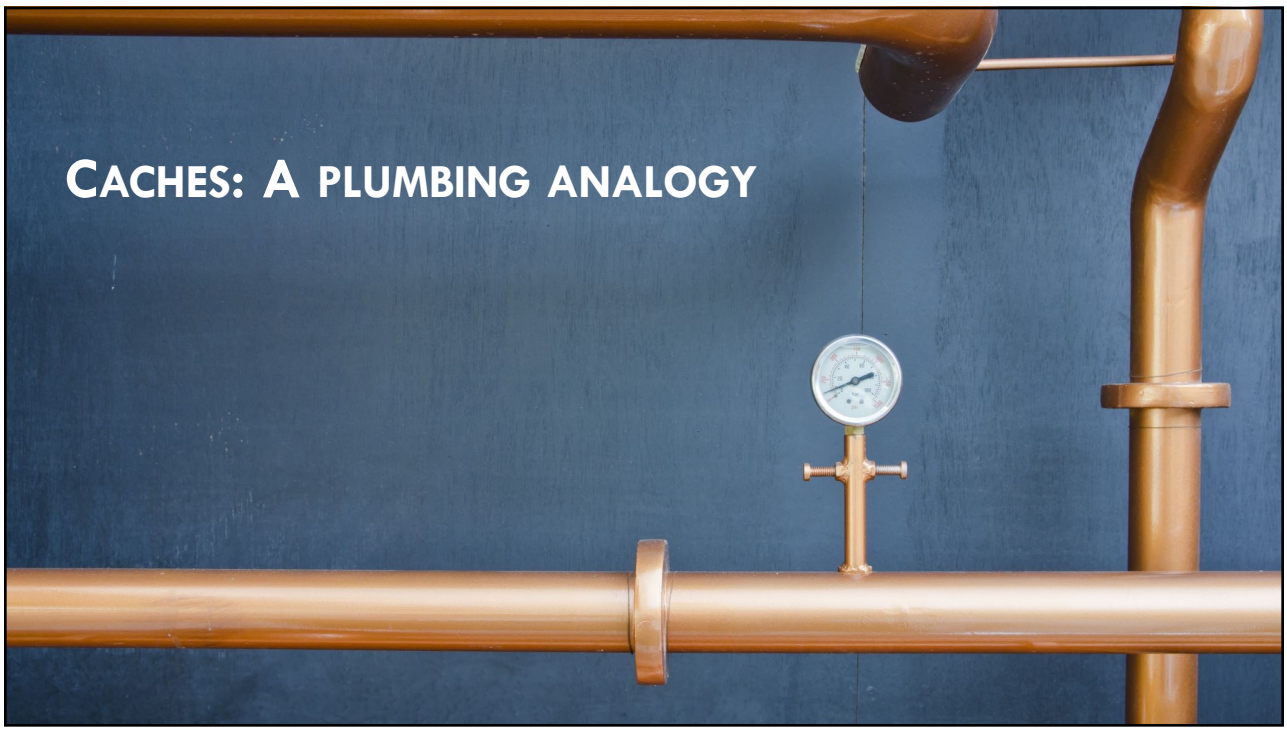
2

Topics covered in this lecture

- GPUs
  - Caching
  - Data and task based parallelism
  - Flynn's taxonomy
  - CPU-GPU differences

Why can thousands of GPU threads still be slow?

- Previous lecture: GPUs are powerful because they run many simple operations in parallel
- But computation is only part of the story
- Threads need data
- Data may be far away
- Fetching data can take hundreds of clock cycles
- So, today's real question is:
  - How do CPUs and GPUs keep work moving when memory is slow?



5

Two words before the plumbing starts
Latency and Bandwidth

- Latency: *how long* it takes to get one thing
  - "How long until the plumber gets the tool?"
- Bandwidth: *how much* can be moved at once
  - "How many tools fit in the van?"
- Caches help because nearby data has lower latency
- Wider memory buses help because more data can move at once
- CPUs and GPUs both fight memory delay, but they fight it differently

6

Caches: A plumbing analogy

- Consider a plumber with a *toolbox* (a **cache**) that can hold four tools
- A number of the jobs being attended to are similar
  - So, the same four tools are *repeatedly used* (a **cache hit**)
- However, a significant number of jobs require additional tools; if the plumber does not know in advance what the job will entail?
  - Arrives and starts work
  - Partway through the job, the plumber needs an additional tool
  - Since it's not in the *toolbox* (**L1 cache**), plumber retrieves the item from the *van* (**L2 cache**)

Some jobs are tougher for the plumber

- Occasionally the plumber needs a special tool
  - Must leave the job, drive to the local *hardware store* (**global memory**), fetch the needed item, and return
- Plumber does not know *how long* (the latency) this operation will take
  - There may be *congestion* on the freeway and/or *queues* at the hardware store (other processes **competing for main memory access**)
  - Clearly, this is not a very efficient use of the plumber's time

While fetching new tools the plumber is not working on the problem at hand

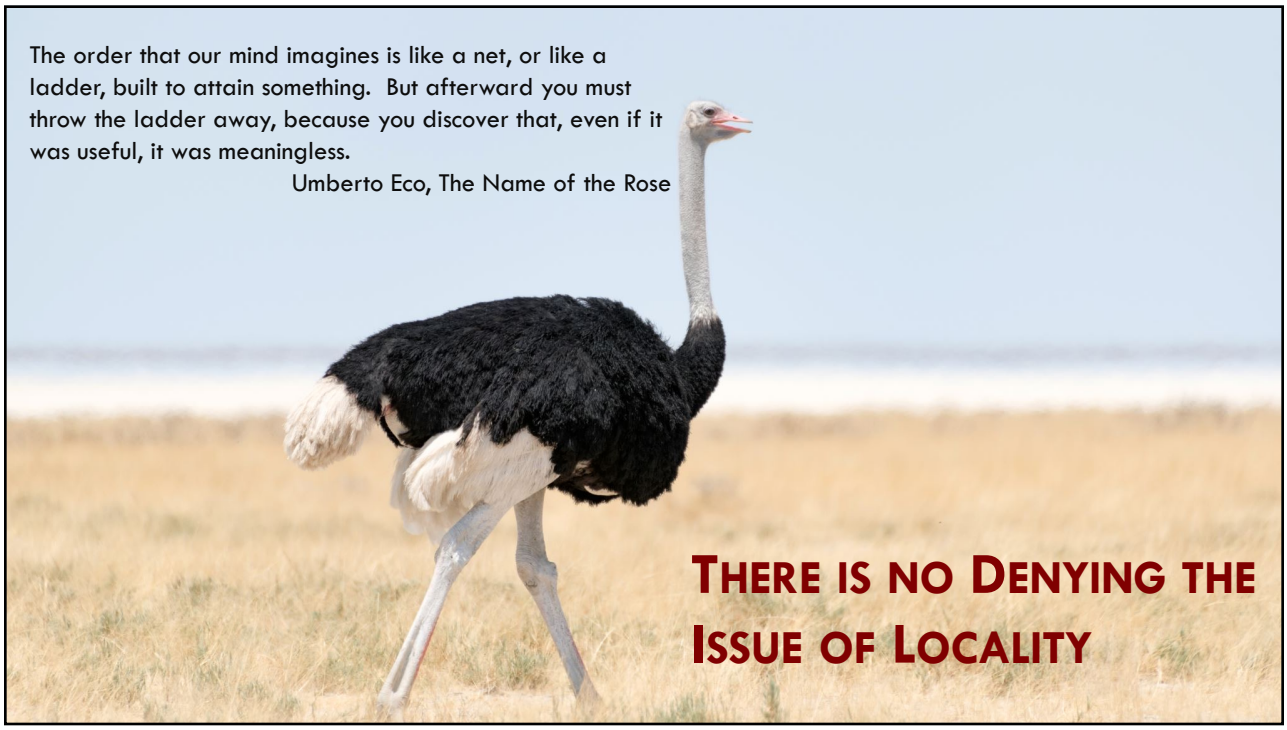
- Each time a different tool or part is needed?
  - It needs to be fetched by the plumber from either the van or the hardware store
- While this might seem bad, fetching data from an HDD or an SSD is even worse, akin to **ordering an item** at the hardware store
  - In comparative form, data arrives by *snail mail*

Typical latencies to global memory are in the order of hundreds of clock cycles

- Increasingly, the answer to this problem from traditional CPU processor design has been to increase the size of the cache
  - In effect, **arrive with a bigger van** so *fewer trips* to the hardware store are necessary
- There is, however, an **increasing cost** to this approach
  - Both in terms of **capital outlay** for a larger van and
  - The time it takes to **search** a bigger van for the tool/part

The approach taken by most CPU designs today?

- Arrive with a *van* (**L2 cache**) and a *truck* (**L3 cache**)
- In the extreme case of the server processors?
  - A huge 18-wheeler is brought in to try to ensure that the plumber is kept busy for just that little bit longer
- All of this work is necessary because of one fundamental reason
  - The CPUs are designed to run software where the **programmer does not have to care about locality**



Locality

- Locality is an issue, regardless of whether the CPU tries to hide it from the programmer or not
- The denial that this is an issue on CPUs is what leads to the huge amount of hardware necessary to deal with memory latency

GPU design takes a different approach [1/2]

- Places the GPU programmer in charge of dealing with locality
- Instead of an 18-wheeler truck, gives programmer a number of small vans and a very large number of plumbers
- The **programmer must deal with locality**
  - Programmer needs to think in advance about what tools/parts (memory locations/data structures) will be needed for a given job
  - These then need to be **collected in a single trip** to the hardware store (global memory)
    - And placed in the correct van (on chip memory) for a given job **at the outset**

GPU design takes a different approach [2/2]

- Given that this data has been collected, **as much work as possible** needs to be performed with the data
  - To avoid having to fetch and return it only to fetch it again later for another purpose
- Thus, the continual cycle of *work-stall-fetch* from global memory is broken

GPU design: Analogy

- **Workers at a production line**
- Workers are supplied with baskets of parts to process
  - This simple process of planning ahead allows the programmer to schedule memory loads into the on-chip memory ***before*** they are needed

Source: Wikipedia

- What if each worker individually fetches widgets one at a time from the store manager's desk?
  - Terribly inefficient!

But this could be done with a CPU cache as well, couldn't it? Yes, and no …

- You can use special cache for instructions that allow **prefilling** of the cache with data you expect the program to use later
- The downside of the cache approach over the GPU shared memory approach is **eviction** and **dirty data**

Data in a cache is said to be **dirty** if it has been written by the program

- To free up the space in the cache for new useful data?
  - The dirty data must be written back to global memory before the cache space can be used again
  - This means instead of one trip to global memory of an unknown latency?
    - We now have two … one to write the old data and one to get the new data

Let's contrast this with the explicit local memory model such as the **GPU's shared memory**

- The big advantage of the programmer-controlled on-chip memory is that the programmer is in control of *when* the writes happen
- If you are performing some local transformation of the data, there may be no need to write the intermediate transformation back to global memory
- With a cache, the cache controller *does not know* what needs to be written and what can be discarded
  - Thus, it writes everything, potentially creating lots of useless memory traffic that may, in turn, cause unnecessary congestion on the memory interface

Cache coherency

- One important distinction between the caches found in GPUs and CPUs is **cache coherency**
- In a cache-coherent system, a write to a memory location needs to be **communicated to all levels of the cache in all cores**
  - Thus, all CPU processor cores see the same view of memory at any point in time
  - This is one of the key factors that *limits the number of cores in a processor*
    - Communication becomes increasingly more expensive in terms of time as the processor core count increases
  - The worst case in a cache-coherent system?
    - Each core writes adjacent memory locations as each write forces a global update to every core's cache

Non cache-coherent systems

- A non cache-coherent system by comparison does not automatically update the other core's caches
- **Relies on the programmer** to write the output of each processor core to separate areas/addresses
- Supports the view of a program where a single core is responsible for a single or small set of outputs
- CPUs follow the cache-coherent approach whereas the GPU does not
  - Allows GPUs to scale to a far larger number of cores (SMs) per device

DATA PARALLELISM

Computational capabilities have grown exponentially over the past couple of decades

- What has not kept pace with this evolution of compute power is the access time for data
- **Data-based parallelism** *looks first to the data* and how it needs to be transformed
  - Not so much the tasks that need to be performed
- **Task-based parallelism** tends to be a *coarse grained* approach

Contrasting task and data parallelism

- Example:
  - Performing four different transformations on four separate, unrelated, and similarly sized arrays
- We will contrast approaches to this using task and data-based parallelism

Task-based parallelism

- Assign one array to each of the CPU cores (or SMs in the GPU)
  - On the CPU side we could create four threads or processes to achieve this
  - On GPUs, we would create four separate kernels, one to process each array and run it concurrently
- The parallel decomposition of the problem is driven by thinking about the tasks or transformations, not the data

Data parallelism

- A data-based decomposition would instead **split the first array into four blocks**
  - Assign one CPU core or one GPU SM to each section of the array
- Once completed, the remaining three arrays would be processed in a similar way
- The parallel decomposition here is driven by thinking about the data first and the transformations second

Why data parallelism fits GPUs {Same work, many pieces of data}

- GPUs are happiest when the program says:
  - "Do this same operation…"
  - "…to many pieces of data…"
  - "…with limited coordination."
- Examples:
  - Brighten every pixel in an image
  - Add two large arrays
  - Multiply blocks of a matrix
  - Update many particles in a simulation
- This is why data parallelism connects naturally to GPU hardware

What's in a name? That which we call a rose
By any other name would smell as sweet.

—Juliet
Romeo and Juliet (II, ii, 1-2)
(Shakespeare)

FLYNN'S TAXONOMY

Flynn's taxonomy is a classification of different computer architectures

- SISD : single instruction, single data
- MIMD : multiple instructions, multiple data
- SIMD : single instruction, multiple data
- MISD : multiple instructions, single data

The standard serial programming we are familiar with follows the **SISD** model

- There is a single instruction stream working on a single data item at any one point in time
- This equates to a single-core CPU able to perform one task at a time
- Of course, it's quite possible to provide the illusion of being able to perform more than a single task
  - By simply switching between tasks very quickly, so-called time-slicing

MIMD systems are what we see today in dual- or quad-core desktop machines

- Typical multi-core desktops have a worker pool of threads/processes that the OS will allocate to one of N CPU cores
- Each thread/process has an independent stream of instructions
  - The hardware contains all the control logic for **decoding many separate instruction streams** concurrently

SIMD systems try to simplify the approach taken in MIMD systems

- They accomplish this with the **data parallelism** model
- SIMD systems follow a single instruction stream at any one point in time
  - Require a single set of logic inside the device to decode and execute the instruction stream, rather than multiple-instruction decode paths
- By removing this silicon real estate from the device?
  - Can be smaller, cheaper, consume less power, and run at higher clock rates than their MIMD cousins

Multiple instruction, single data (**MISD**)

- Uncommon architecture that is generally used for fault tolerance
- A type of parallel computing architecture where many functional units perform **different operations on the same data**
  - E.g.: Heterogeneous systems operating on the same data stream and needing to agree on the result
    - Space shuttle flight control computer
  - Task replication may be considered as MISD as well
    - Executing the same instructions redundantly in order to detect and mask errors
- Applications for MISD are much less common than MIMD and SIMD
  - MIMD and SIMD are often more appropriate for common data parallel techniques

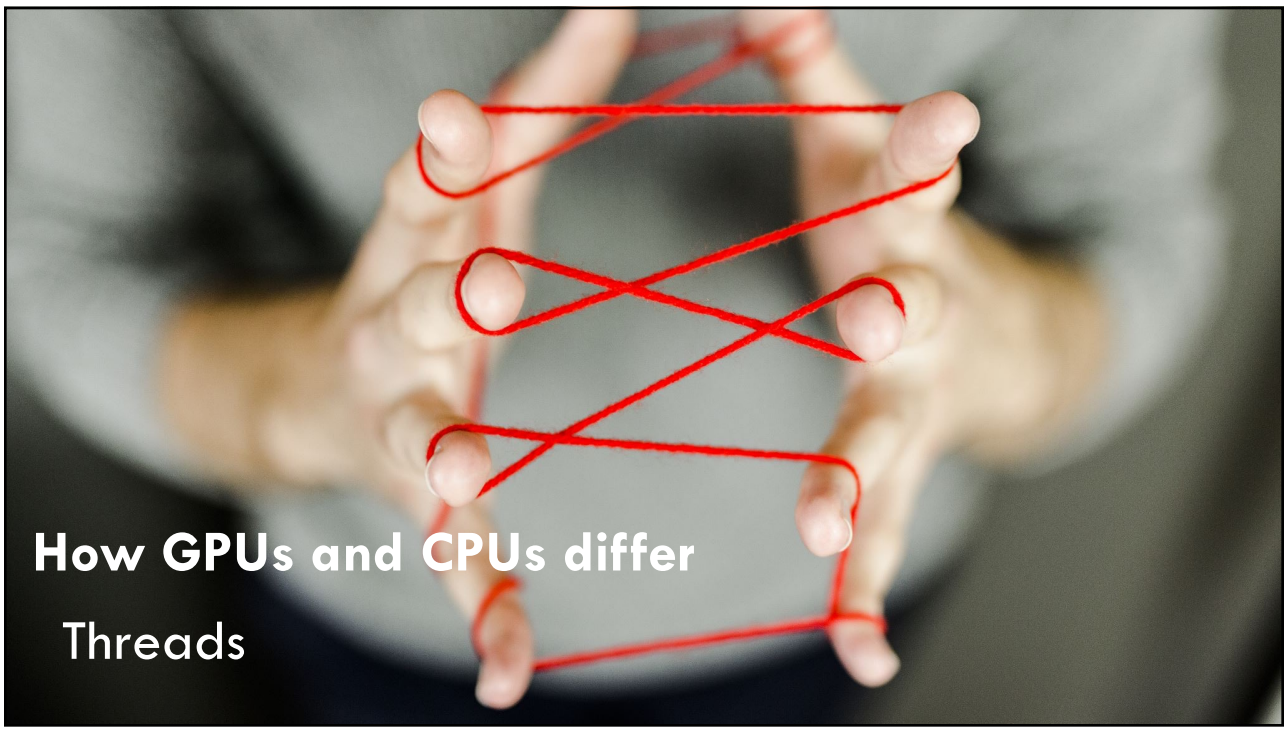
HOW GPUS AND CPUS ARE DIFFERENT

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



37

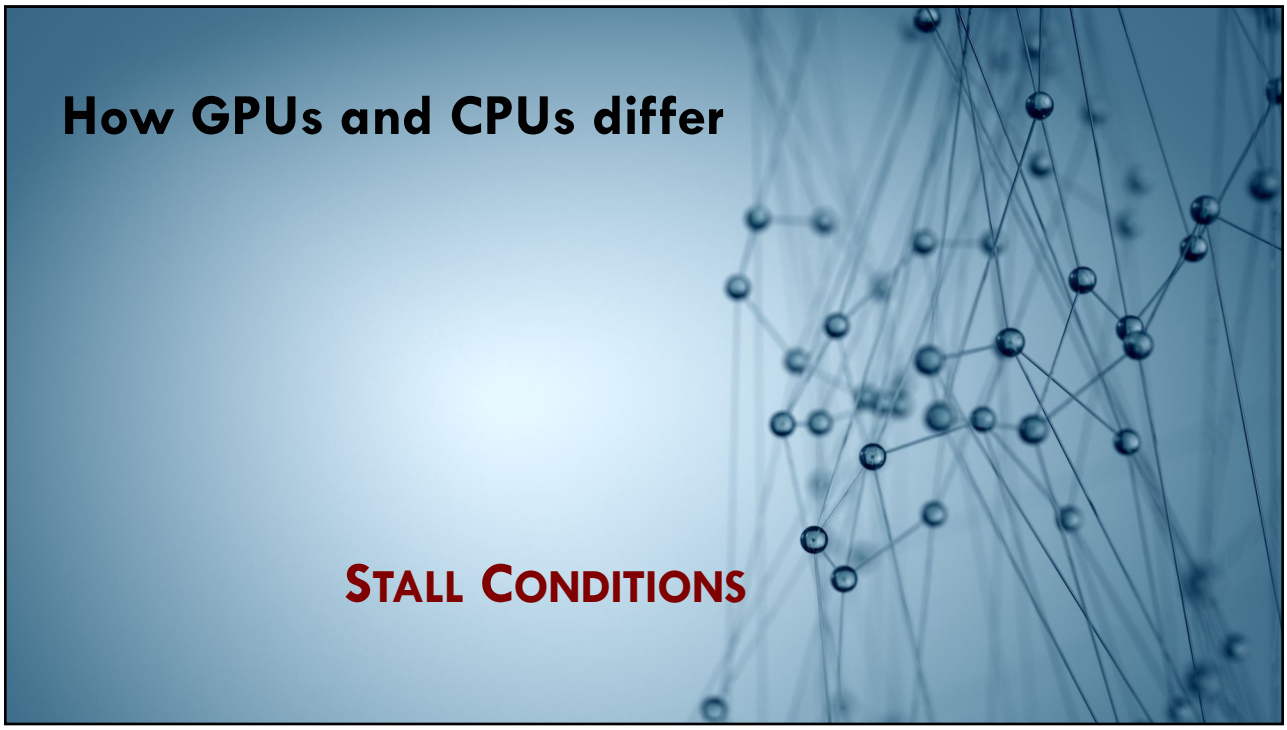
CPUs and GPUs support threads in very different ways [1/2]

- 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

38

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**
  - Provided that 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 use 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

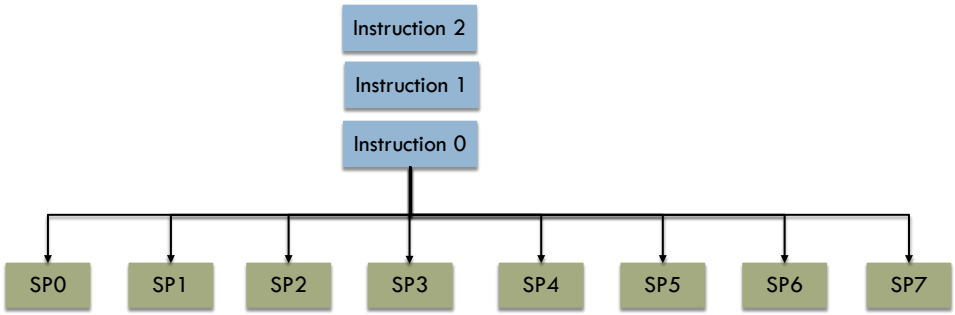
- High-speed memory next to the SM, so-called **shared memory**
- Programmer can leave data in this shared memory
  - Knowing that hardware will not evict it behind programmer's back
- This shared memory is also the primary mechanism communication between threads

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**



When lock-step stumbles: The branching problem

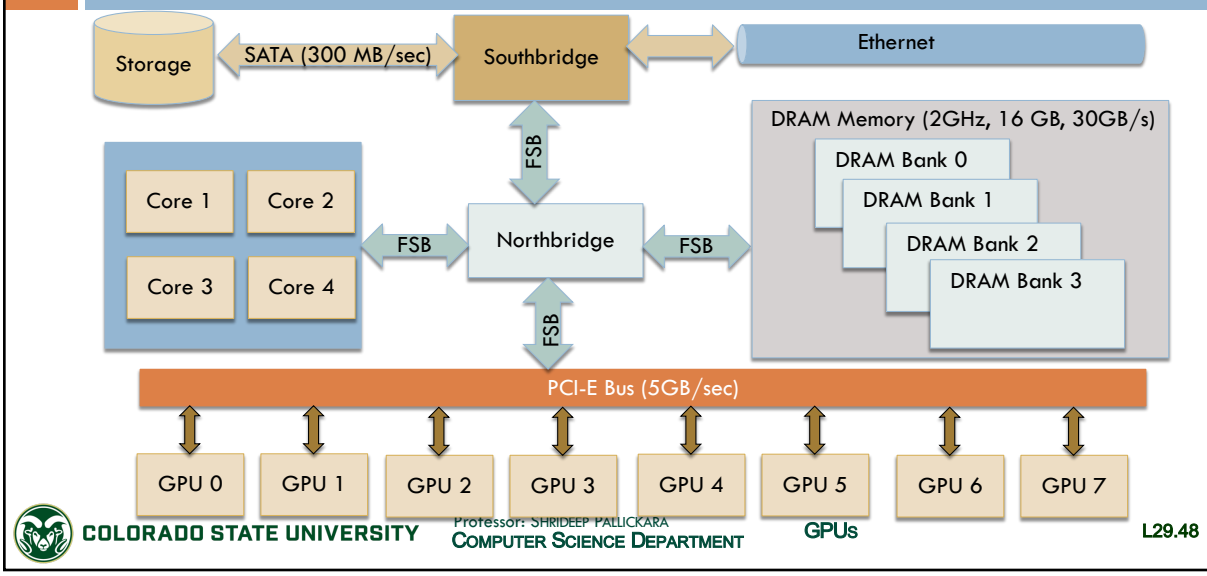
- GPU threads often move together in groups
- This works beautifully when threads do the same thing
- But **branches can split the group**:
  - Some threads take the `if` path
  - Others take the `else` path
- The GPU may have to run both paths and mask off inactive threads
- Result: less parallelism than the program seemed to promise

A TYPICAL PC ARCHITECTURE

COMPUTER SCIENCE DEPARTMENT COLORADO STATE UNIVERSITY

47

Typical PC architecture



COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT GPUs L29.48

48

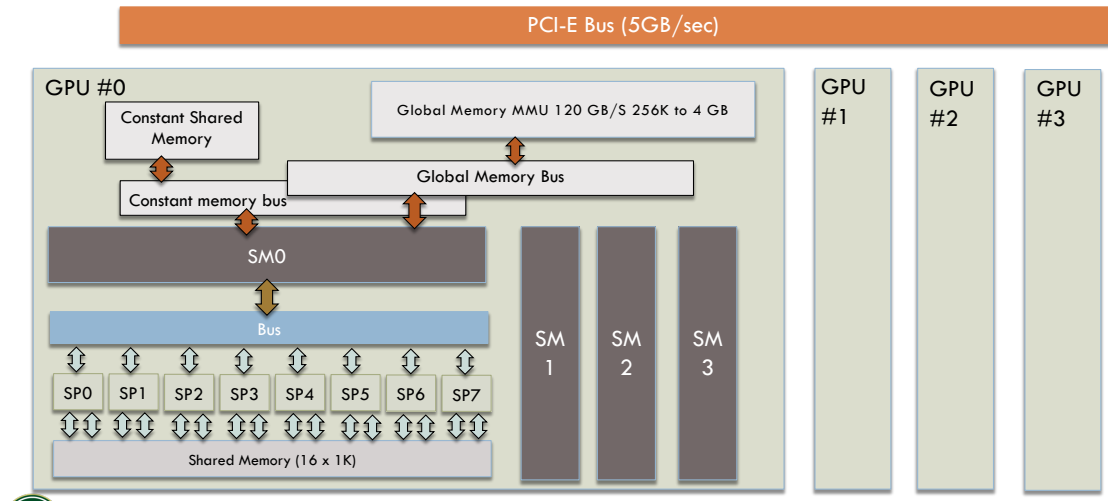
The GPU hardware

- 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
- 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

Single core CPUs and parallelism

- Most programs written in the past few decades, with the exception of perhaps the past 15 years or so, were single-thread programs
  - The primary hardware on which they would execute was a single-core CPU
- Sure, you had clusters and supercomputers that sought to exploit a high level of parallelism
  - **Duplicating the hardware** and having thousands of commodity servers instead of a handful of massively powerful machines

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]