CSC367 Parallel computing

Lecture 16: General-purpose computing with Graphics Processing Units (GPUs)

(Introduction)

Today

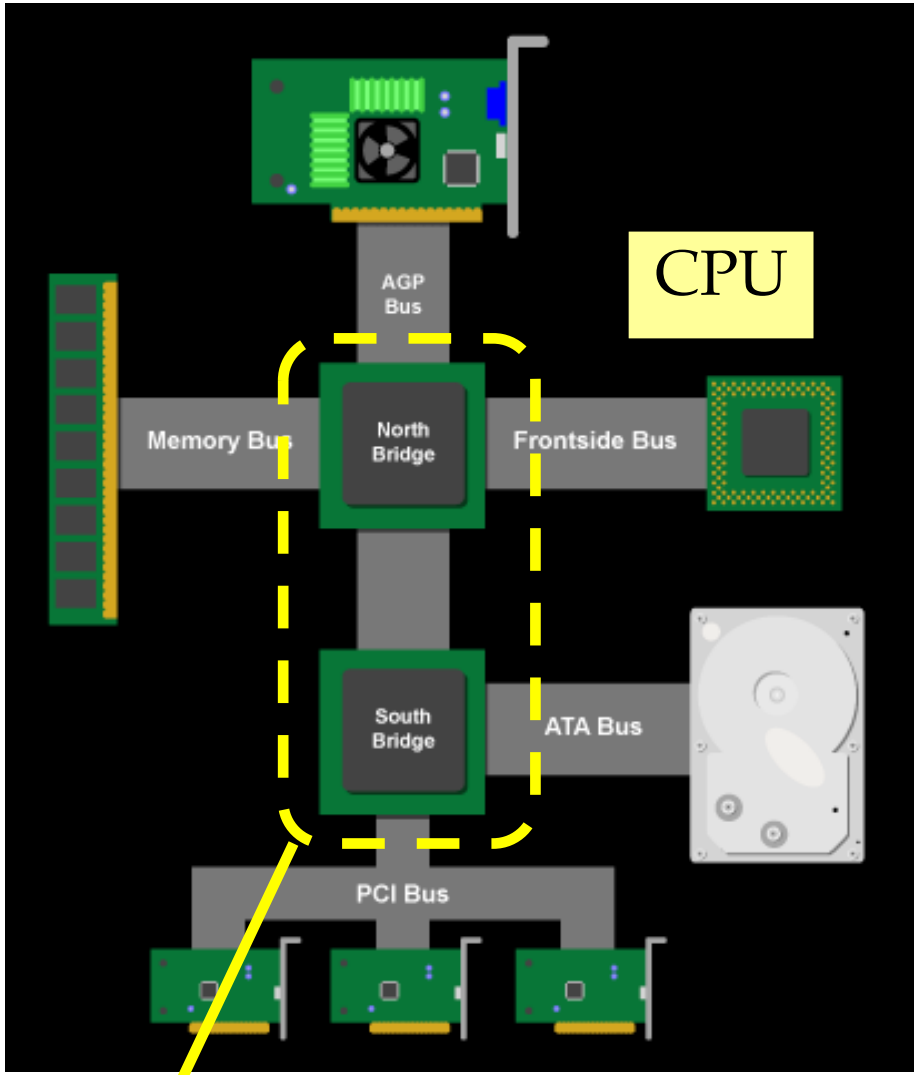
- Revisiting PC architecture
- Why GPUs?
- History
- General-purpose GPUs – the architecture basics

For all your CUDA questions see the CUDA documentation:

https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Classic PC architecture

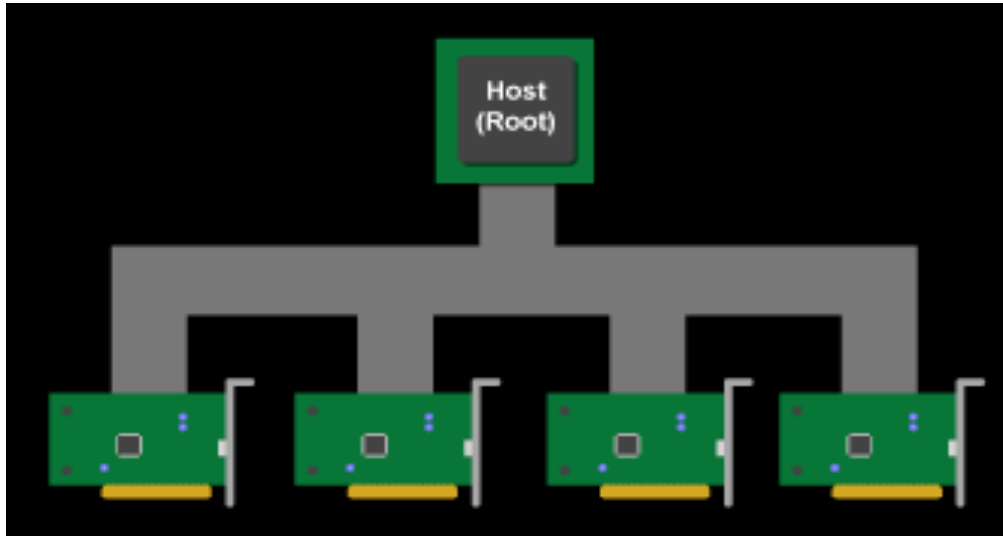
- Northbridge connects 3 components that must communicate at high speed
  - CPU, DRAM, video
  - Video also needs to have 1st-class access to DRAM
  - Older NVIDIA cards are connected to AGP, up to 2 GB/s transfers
- Southbridge serves slower I/O devices
  - e.g., hard drives, USB ports, Ethernet ports, etc.
- Core logic chipset acts as a switch
  - Routes I/O traffic among the different devices that make up the system.



Img src: https://arstechnica.com/features/2004/07/pcie/

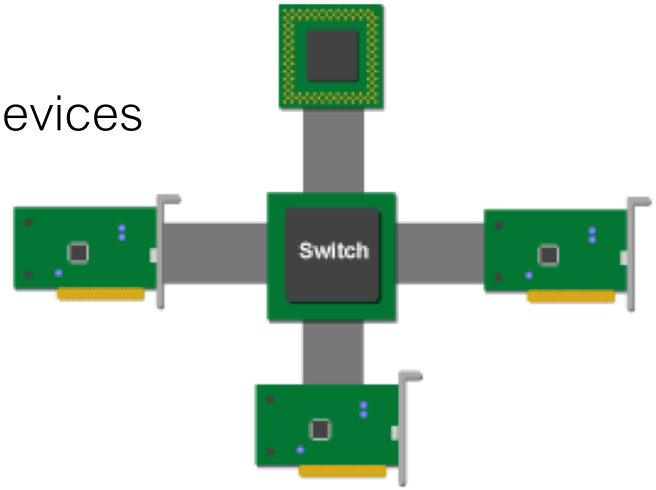
Original PCI bus

- Shared bus topology
  - Network, sound card, etc. use bus to communicate with CPU
  - Needs bus arbitration: who gets access and when?
- Southbridge, Northbridge and CPU together form the host (root) role
  - Detect and initialize PCI devices, controls the PCI bus
- Simple, easy to implement
  - Access PCI devices in the same way as accessing memory (LD/ST)
- Limitations:
  - Performance (doesn't scale)
  - Advanced functionality



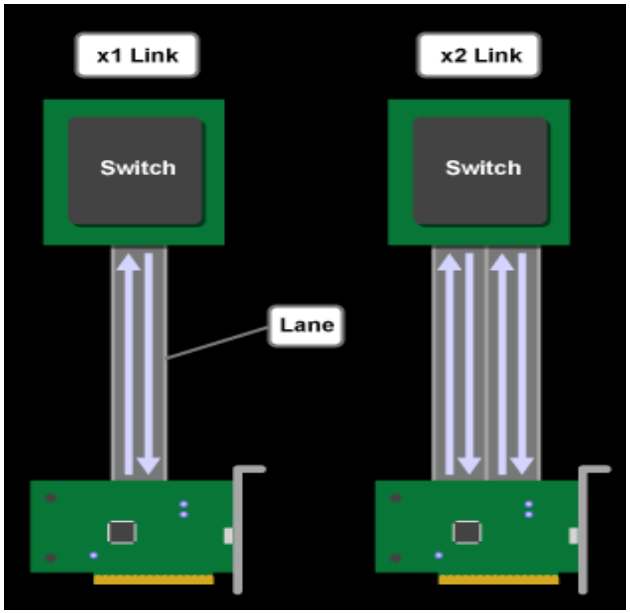
PCI Express (PCIe)

- Point-to-point bus topology
  - Point-to-point connections between any two devices
- Shared switch instead of shared bus
  - Each device has its own link
  - CPU can talk to any device directly
  - Switch manages all traffic



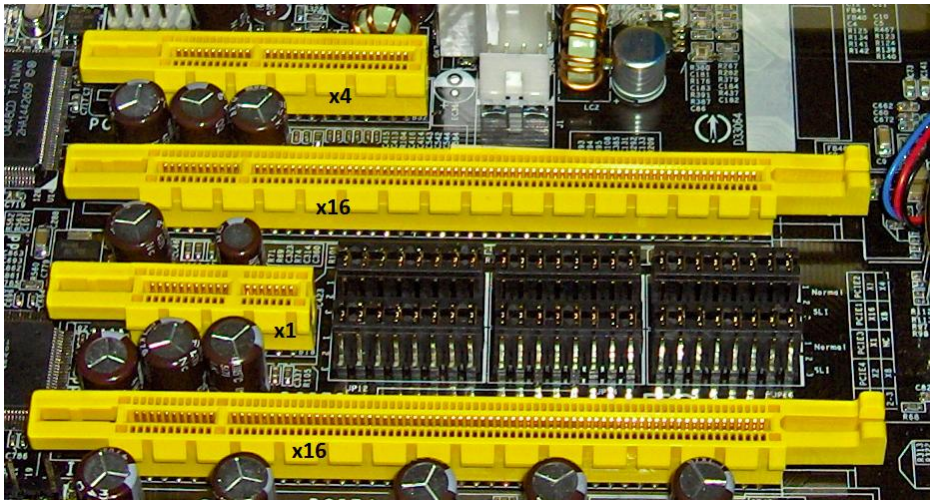
PCIe Links and Lanes

- Each link consists of one or more lanes
- Lane: 4 wires (1 bit per cycle in both directions)



Std size	PCIe 1.0	PCIe 2.0	PCIe 3.0	PCIe 4.0
x1	250MB/s	500MB/s	~1GB/s	~2GB/s
x4	1GB/s	2GB/s	~4GB/s	~8GB/s
x8	2GB/s	4GB/s	~8GB/s	~16GB/s
x16	4GB/s	8GB/s	~16GB/s	~32GB/s

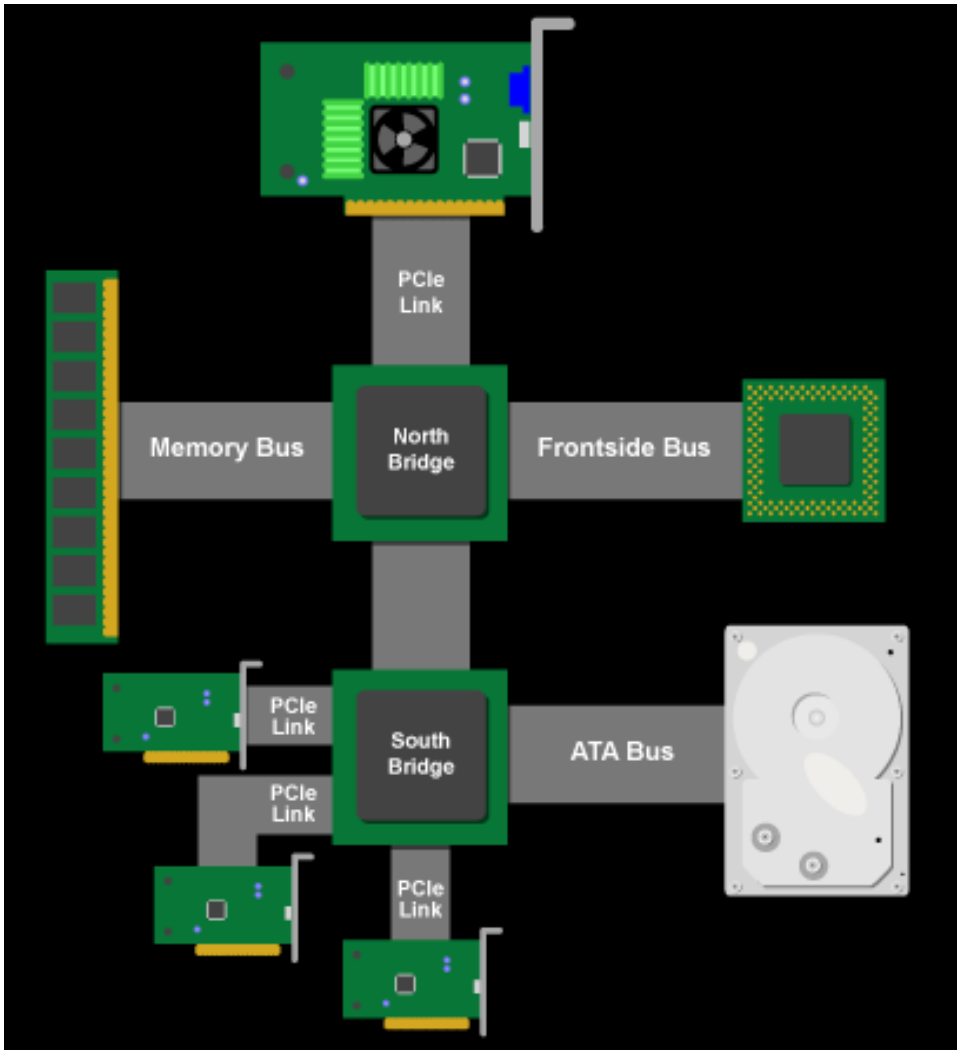
- Smaller card may go in larger slot
- Small cards might take only 1 lane
- High-end graphics cards – 16 lanes



Source: http://en.wikipedia.org/wiki/PCI_Express

PCIe PC architecture

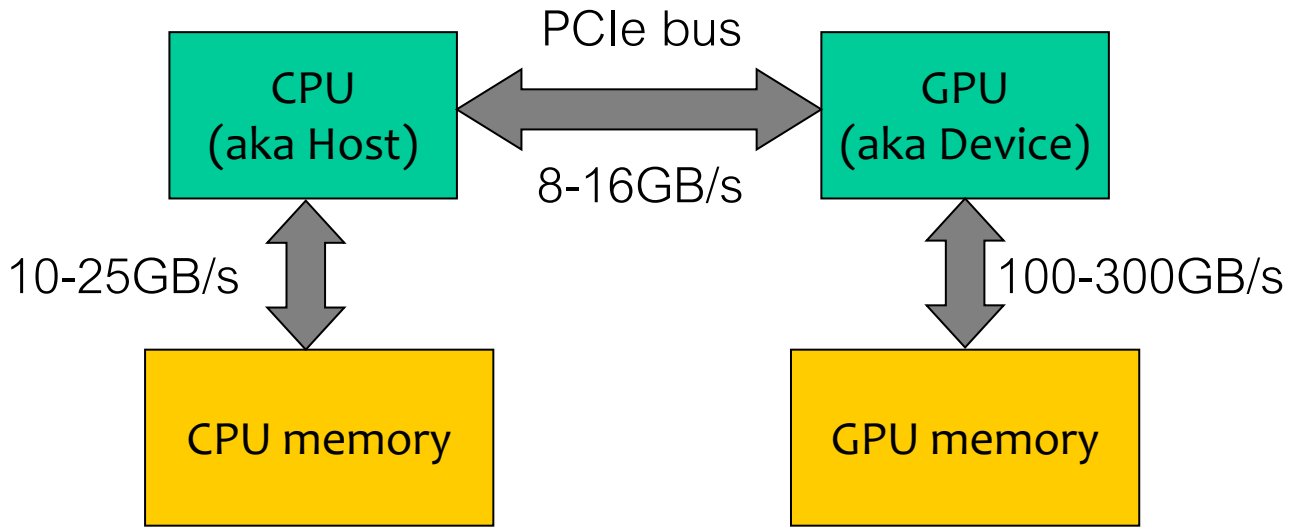
- PCIe forms the interconnect backbone
  - Northbridge and Southbridge are both PCIe switches
  - Some Southbridge designs have built-in PCI-PCIe bridge to allow old PCI cards
  - Some PCIe I/O cards are PCI cards with a PCI-PCIe bridge



Source: Jon Stokes, PCI Express: An Overview
https://arstechnica.com/features/2004/07/pcie/

CPU-GPU architecture

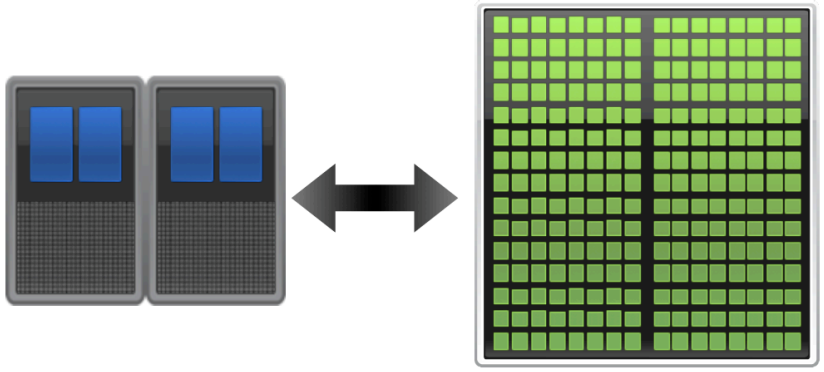
- The GPU is connected to the CPU via the PCIe bus
- The GPU has its own memory (usually much smaller that the CPU memory)



“If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?”—Seymour Cray

CPU vs. GPU architecture

- CPU is latency-optimized
- GPU is throughput-optimized but hides latency pretty well!



Latency Optimized CPU

Fast Serial Processing

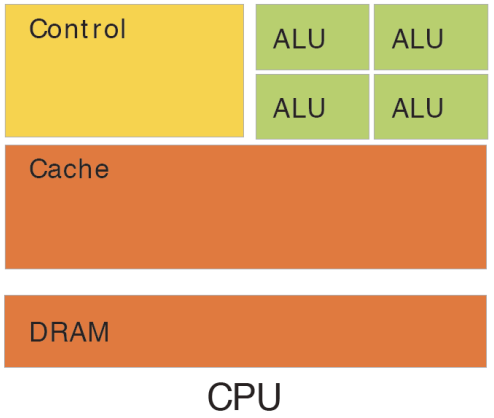
Throughput Optimized GPU

Scalable Parallel Processing

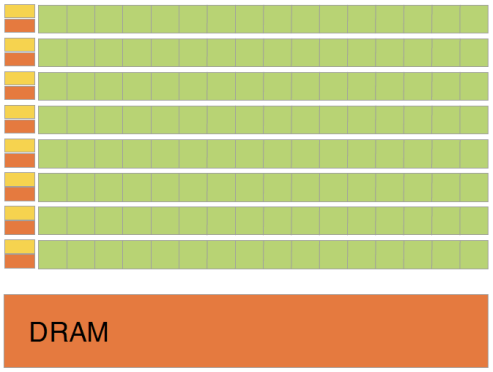
Image source: Demmel

CPU vs. GPU architecture

- CPU is latency-optimized
  - ➢ Optimized for low-latency access to cached data sets.
  - ➢ Has a large control logic for effective execution of serial code.
  - ➢ A few arithmetic logic units (ALUs), a large control logic.
- GPU: throughput-optimized
  - Ideal for data-parallel computation
  - More transistors dedicated to computation
  - Lots of ALUs/cores, a small control logic: good for high throughput.
- The GPU has a small brain but lots of compute power!
  - ➢ It is up to the programmer to make their program embarrassingly parallel to use the compute power on the GPU!



CPU

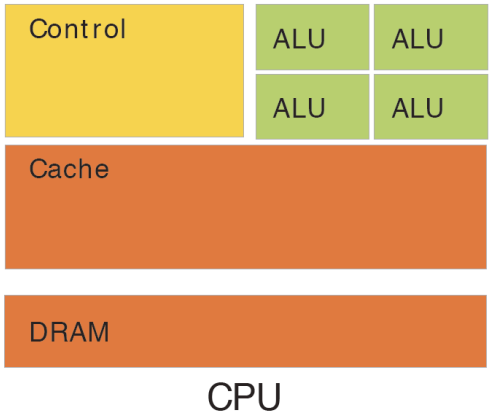


GPU

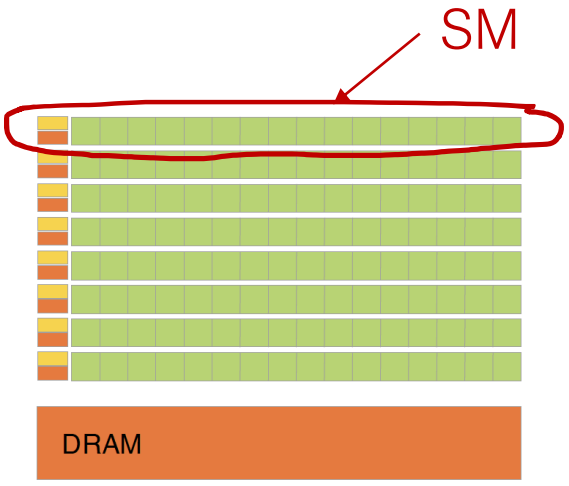
Image source: NVIDIA

CPU vs. GPU architecture

- CPU is latency-optimized
  - ➢ Optimized for low-latency access to cached data sets.
  - ➢ Has a large control logic for effective execution of serial code.
  - ➢ A few arithmetic logic units (ALUs), a large control logic.
- GPU: throughput-optimized
  - Ideal for data-parallel computation
  - More transistors dedicated to computation
  - Lots of ALUs/cores, a small control logic: good for high throughput.
- The GPU has a small brain but lots of compute power!
  - ➢ It is up to the programmer to make their program embarrassingly parallel to use the compute power on the GPU!

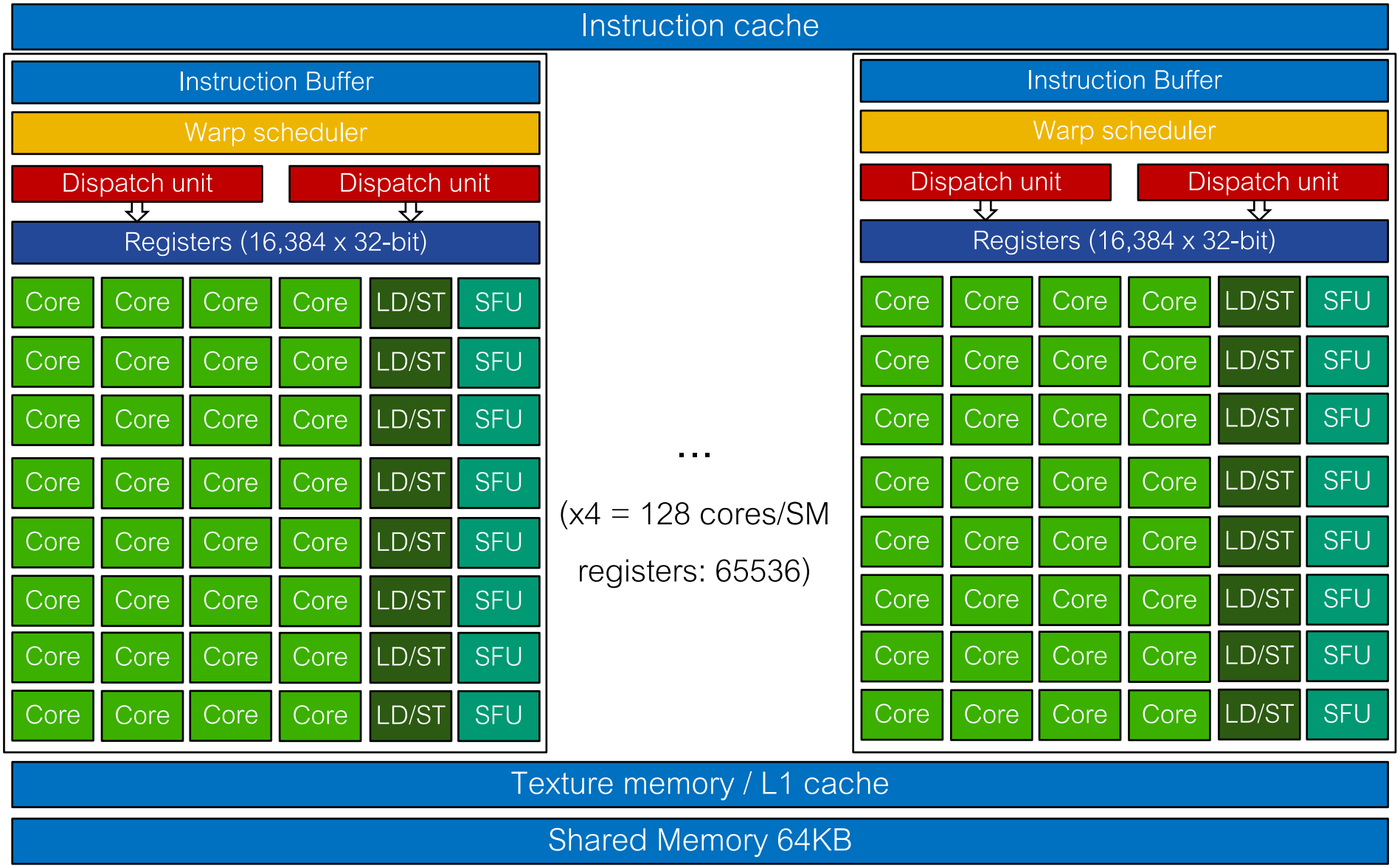


CPU



GPU

Streaming Multiprocessor (SM)



Why GPUs?

- Large collection of Single Instruction Multiple Data (SIMD) multiprocessors
  - **Massive thread parallelism** – 100s of processors
- Good for data-parallel computations
  - Must have an inherently high level of parallelism in our application

GPUs vs CPUs – conceptual approach

- Example: add arrays A and B, into array C
  - CPU (sequentially): Allocate memory, for loop to add elements pairwise
  - CPU (parallel):
    - create N threads (N = number of cores on the CPU)
    - partition the data equally into ranges among the threads
    - each thread: for each i in its range of elements, add those elements
    - wait for all threads to finish
- How does performance scale?
  - Assuming 8 cores => Smax = ~8X
  - How to scale this further? How many threads?
  - Limited cores, memory bus contention, penalty switching between threads

GPU-based parallelization basics

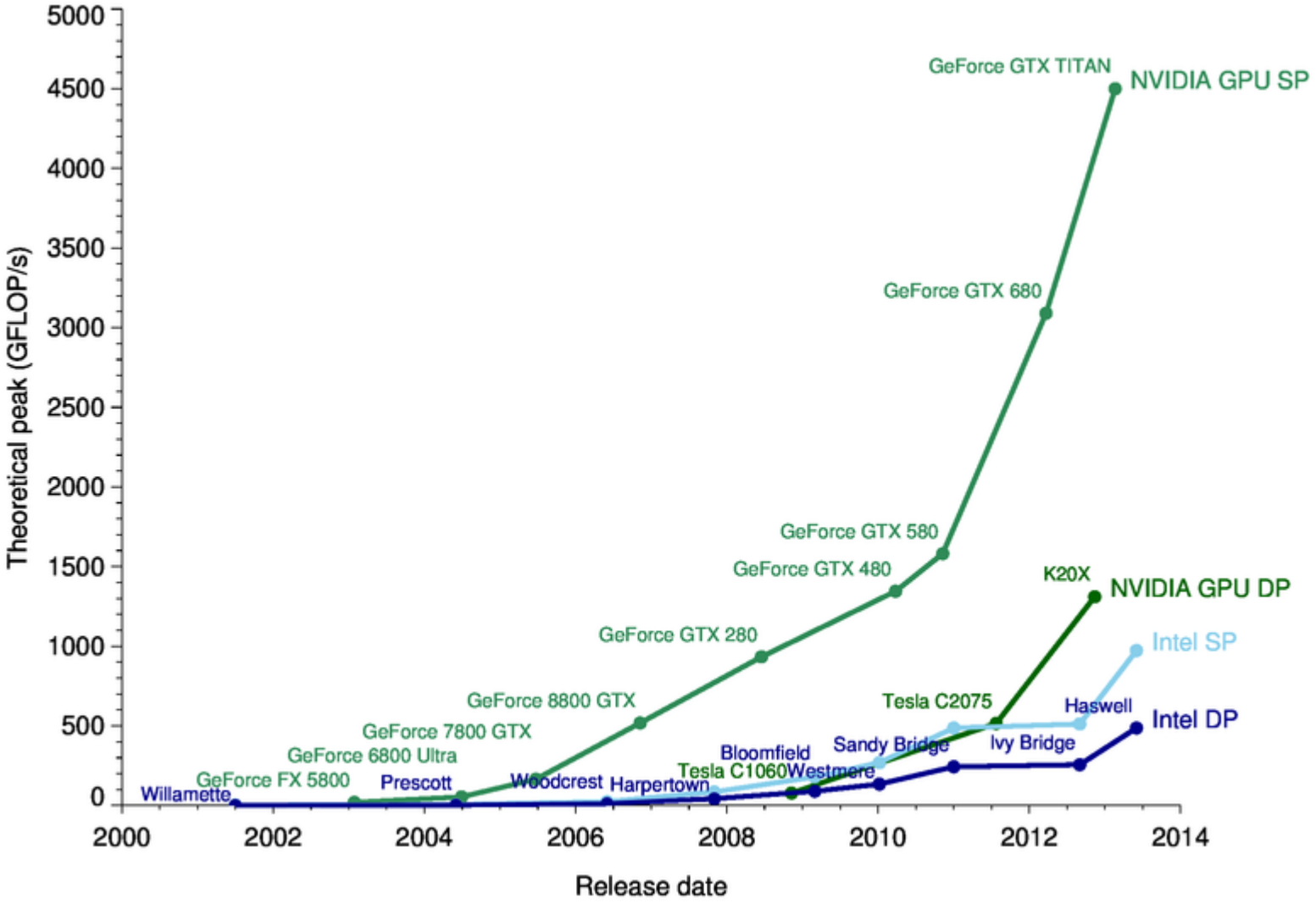
- GPU-based parallelization – steps:
  - allocate memory for arrays on the GPU
  - transfer the data from CPU memory into GPU memory
  - launch a kernel (function executed by each processing thread)
    - spawn a **massive** amount of threads (e.g., 32000, 64000, etc.)
    - each thread is instructed to only handle a few elements
  - wait for all threads to finish
  - transfer results back to CPU memory
- GPU hides memory latency better than CPU
  - GPU switches between threads to hide latency
  - Threads do light-weight jobs and we have tons of registers, switching between threads is basically free

Summary: Main tradeoffs

- CPU - jack-of-all-trades: runs your OS, a variety of applications, all of which must get good latency
  - Complex control unit, large chip to provide the logic
  - Several levels of caches, increasingly-larger

- GPU – process one thing in massively-parallel fashion
  - A lot more cores, but smaller clock speed (100s of MHz)
  - Control unit much simpler too => Less diverse computations!
  - Smaller caches, not quite the same goal as for CPUs

CPU vs GPU performance



Source: http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html

CUDA History

- Past GPUs: only for graphics processing (vertex & pixel shaders, etc.)
- CUDA architecture: support for general purpose computing
  - Hard to leverage architectural features, must disguise computations as graphics problem
  - Until ... CUDA C/C++ language, hardware driver, compiler
- GPUs became popular for non-graphics applications after the release of CUDA.
- It all started with stencil computations since the algorithm is inherently embarrassingly parallel: Scientific computing people were super excited (2008 onwards!), they even sold GPUs with their software!
- Soon they realized that 1) not all applications can be made embarrassingly parallel, 2) not all data fits on the GPU memory so they have to pay high costs of transferring data from CPU to GPU
- The hype was back after researchers realized GPUs are great for deep learning!

Diversity of applications

- High performance computing
- Numerical analysis
- Physics simulations
- Machine learning
- Databases
- ... and many more!
- Still good for graphics processing and rendering though!

NVIDIA GPU generations

- Some of the recent GPU generations from NVIDIA:
  - Tesla (before 2010): The name is still used for high-end GPUs.
  - Fermi (2010)
  - Kepler (2012)
  - Maxwell (2014)
  - Pascal (2016) – teaching labs!
  - Volta!
  - Turing!
- Many new features in recent versions, as hardware evolved

Pascal architecture (1050 cards) in the teaching labs!

CUDA Compute Capability

- Compute capability tells us what features are supported by that GPU
- First compute capabilities: 1.0, 1.1, 1.2, 1.3, and 2.0 (your textbook)
- The NVIDIA software development kit is called CUDA SDK
- The CUDA SDK's have different versions.
  - CUDA SDK 8.0 (in lab machines) supports compute capability 2.0 – 6.x (Fermi, Kepler, Maxwell, Pascal).

CUDA Compute Capability

- Higher capability versions are supersets of lower capabilities
  - Support newer features, but older ones too
  - e.g., double-precision floating point, atomics, unified virtual addressing, etc.

- Can compile code for a specific capability:
  - nvcc –arch=sm_11 (sm_11 means its compiling for capability 1.1)
  - nvcc –arch=sm_30 (sm_30 means its compiling for capability 3.0)

CUDA Compute Capability

- Find the compute capability of your GPU from the NVIDIA website:

  https://developer.nvidia.com/cuda-gpus

  - For example GeForce GTX 1050 supports compute capability 6.1

- See appendix H from the CUDA documentation to see what features are supported for each compute capability:

  https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

- Lets look at some examples (copied directly from the appendix H in the above link):

Feature support for compute capability

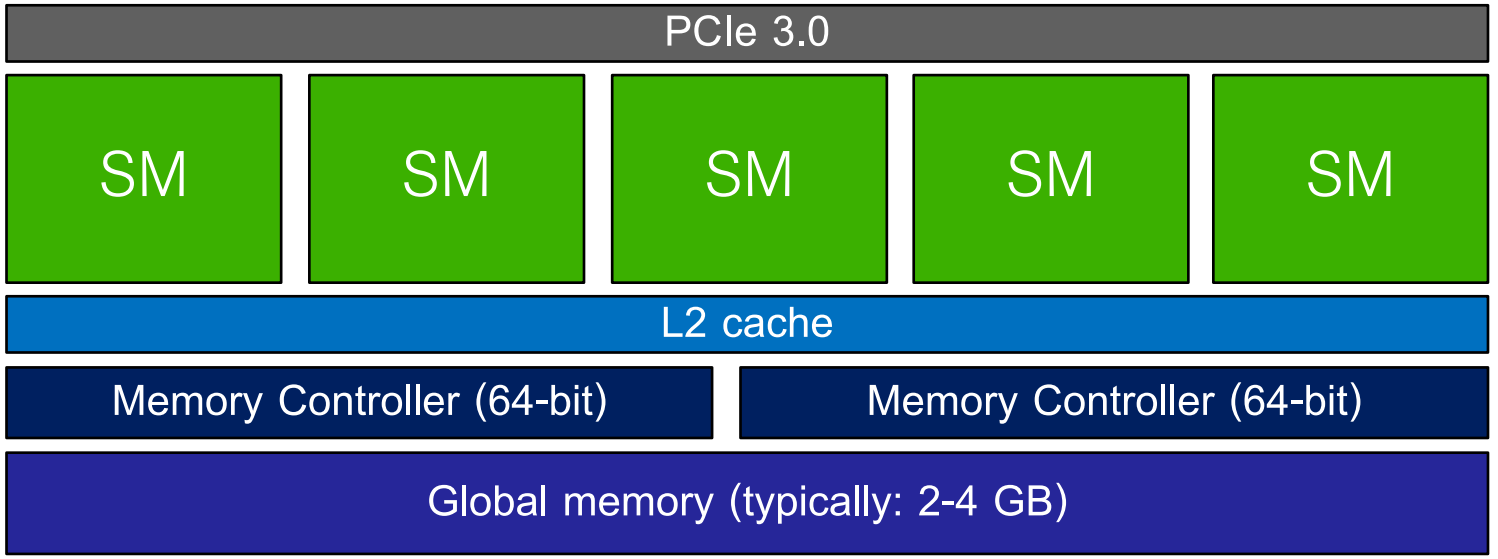
Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	**3.0**	**3.2**	**3.5, 3.7, 5.0, 5.2**	**5.3**	**6.x**	**7.x**
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes					
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No				Yes	
Warp vote and ballot functions (Warp Vote Functions)	Yes					
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Unified Memory Programming						
Funnel shift (see reference manual)	No	Yes				
Dynamic Parallelism	No		Yes			
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No			Yes		
Tensor Core	No					Yes

Feature support for compute capability

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32

CUDA architecture

Pascal architecture (1050 cards) in the teaching labs!



- Maxwell
  - GTX 750 Ti: 5 SMs, total 640 CUDA cores @ 1110MHz, two 64-bit memory controllers (128-bit memory bus width), memory bandwidth: (only) 86.4 GB/s
- Newer Pascal architecture:
  - GTX 1080: 20 SMs, 2560 CUDA cores @1.6GHz, eight 32-bit memory controllers (256-bit total), memory bandwidth: 320 GB/s
  - GTX 1080 Ti: 3584 cores @ ~1.6GHz, 352-bit bus width, 484 GB/s memory bandwidth

Streaming Multiprocessor (SM)

- Thousands of registers (can be partitioned among threads of execution)
- Caches (we'll get to these later..)
  - Shared memory – fast data interchange between threads
  - Constant cache – fast broadcast of reads from constant memory
  - Texture cache – to aggregate bandwidth from texture memory
  - L1 cache – to reduce latency to local or global memory
- Warp scheduler
  - Can quickly switch between thread contexts, and issue instructions to groups of threads (aka "warps") which are ready to execute
- Execution units for integer and floating-point operations

Streaming Multiprocessor (SM)