Topic 6

Introduction to GPU

Xinfei Guo
xinfei.guo@sjtu.edu.cn

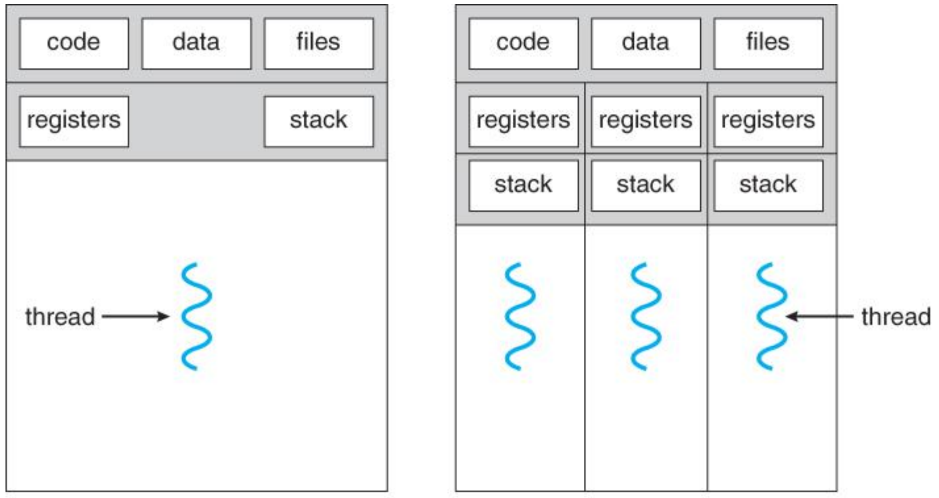
July 4th, 2022

T6 learning goals

- SIMD
  - Introduction to SIMD
  - Case Study: GPU

The concept of “Thread”

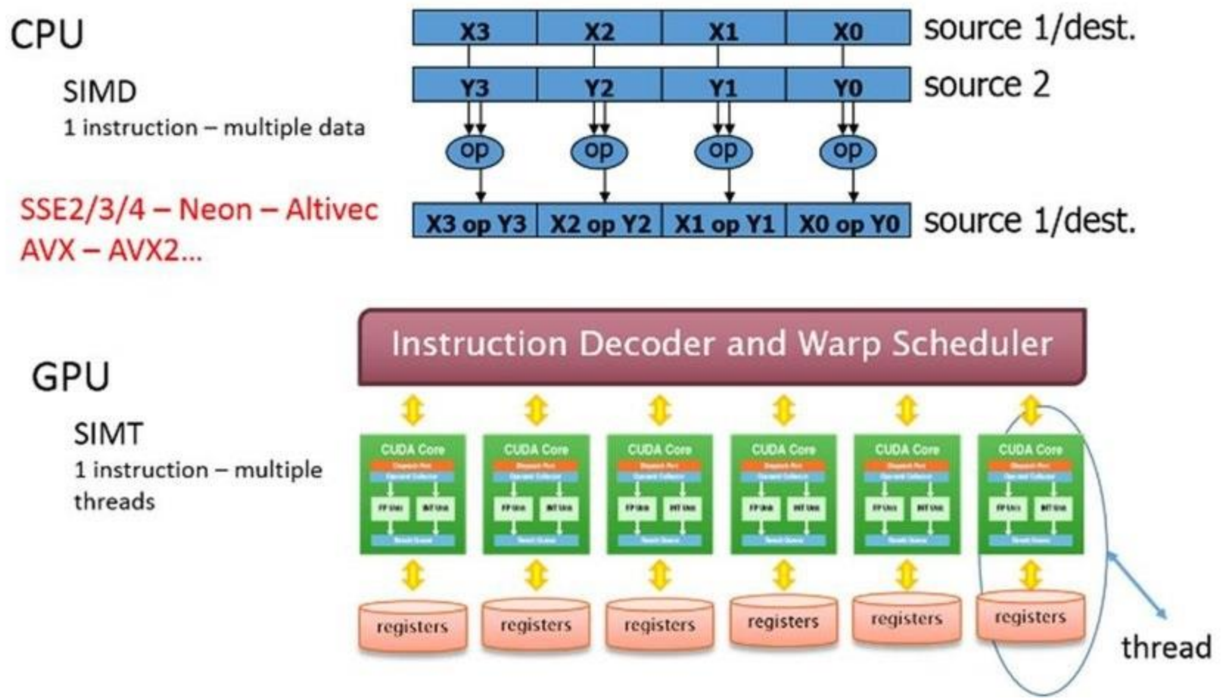
- A thread is a sequence of instructions. It is the virtual component or code, which divides the physical core of a CPU into virtual multiple cores. A single CPU core can have up-to 2 threads per core.
- ***Multithreading*** refers to the ability of a given processor (e.g. CPU, GPU, etc.) to run multiple threads. You can think of threads as *tasks* or *jobs*.
- Each thread has a PC+Processor register and access the shared memory
- Each processor provides one (or more) hardware threads that actively execute the instructions.



single-threaded process multithreaded process

SIMD vs. SMT vs. SIMT

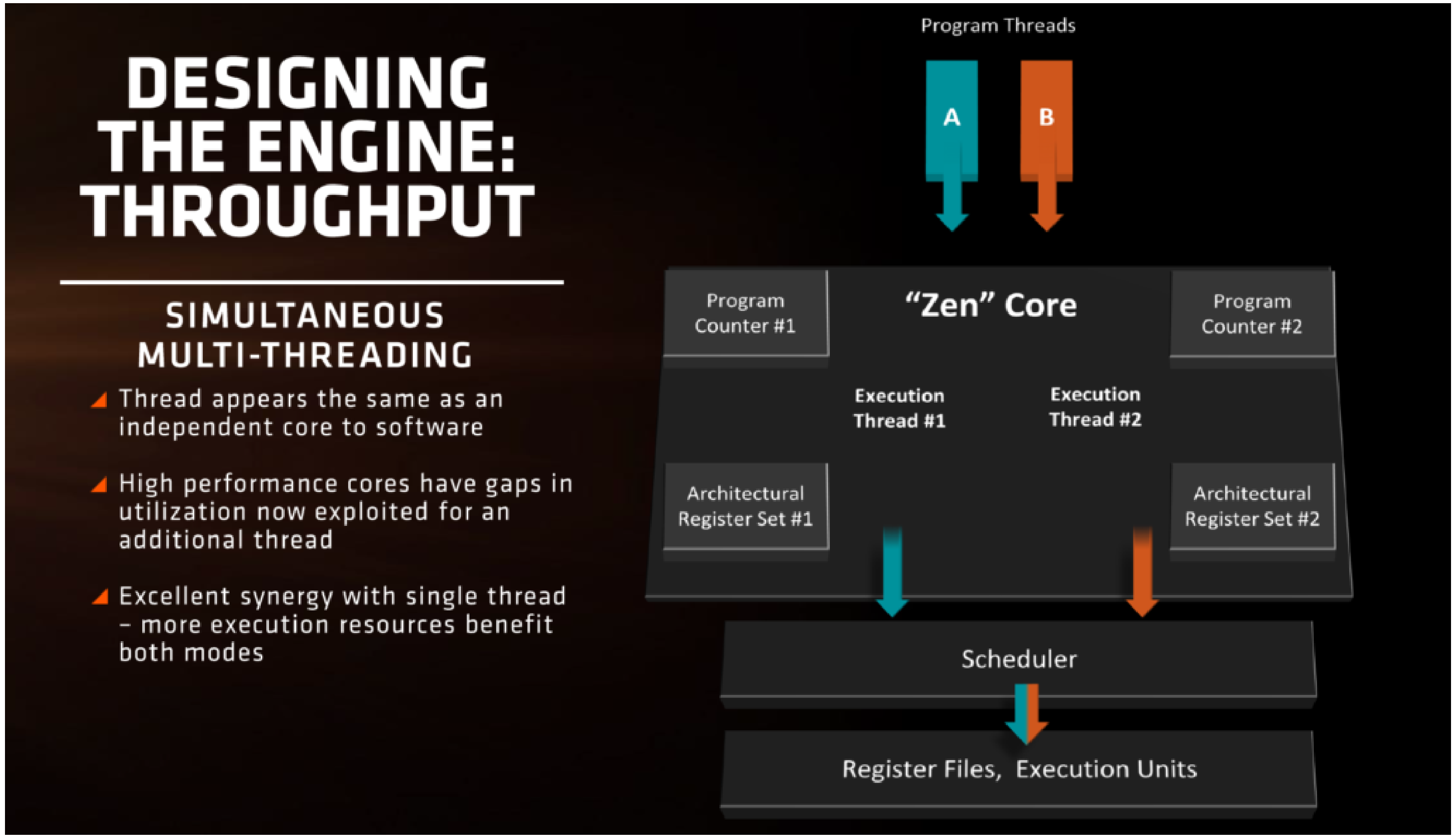
- In SIMD, elements of short vectors are processed in parallel.
- In SMT, instructions of several threads are run in parallel.
- SIMT is somewhere in between – an interesting hybrid between vector processing and hardware threading.



SIMT is generally used in Super-scalar processors to implement SIMD. So technically, each core is scalar in nature but it still works similarly to an SIMD model by leveraging multiple threads to do the same task on various data sets.

SMT: Simultaneous Multi-Threading

- SMT allows a CPU core to leverage multiple threads at a time. Although theoretically, you can have up to 8 threads per core via SMT, it's only feasible to have two.
- SMT is analogous to having two cargo belts at the airport luggage sorting, and one person sorting them. There will be times when one belt is empty but the other still has pending work. In this instance, the person will switch to the other belt and continue sorting till the first belt gets more luggage.



DESIGNING THE ENGINE: THROUGHPUT

SIMULTANEOUS MULTI-THREADING

- Thread appears the same as an independent core to software
- High performance cores have gaps in utilization now exploited for an additional thread
- Excellent synergy with single thread – more execution resources benefit both modes

source: AMD

Graphics Processing Units (GPU)

- A SIMT machine
- A processor optimized for 2D and 3D graphics, video, visual computing, and display, now designed for rapid manipulation and parallel processing of data (e.g. AI)
- GPU becomes a key element of modern computing systems
  - Deep Machine Learning
  - Image processing
  - 3D mapping
  - Gaming
  - Self-driving
  - VR/AR
  - Supercomputers

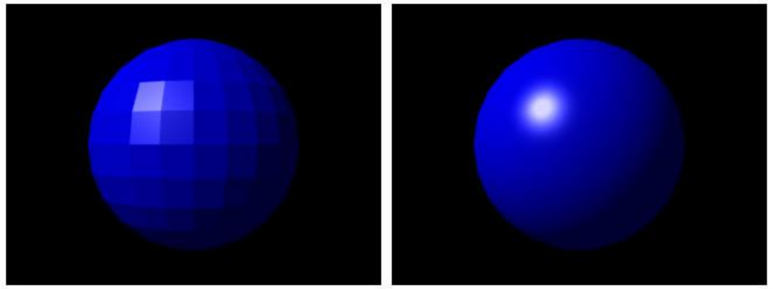
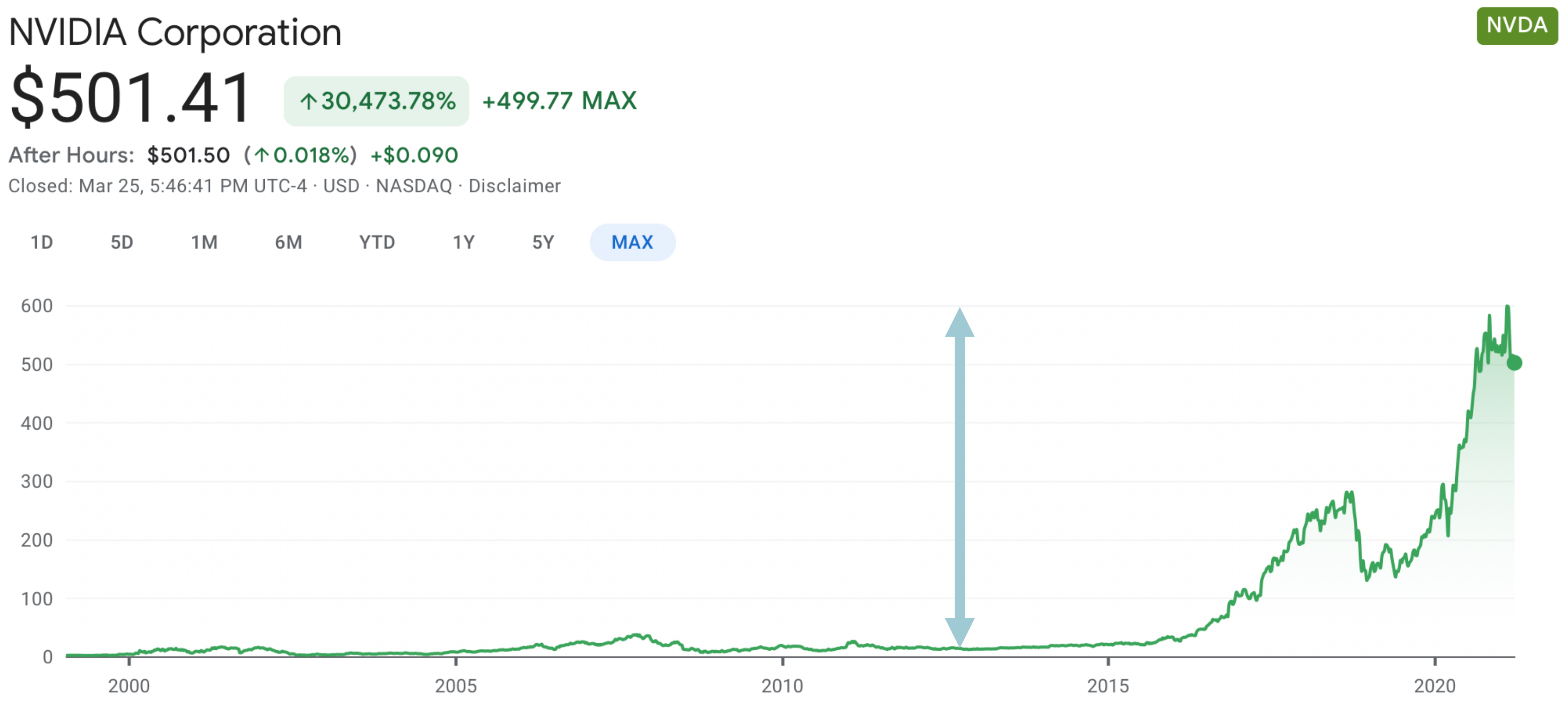

shader

Image: Nvidia

GPUs are everywhere!

The rise of the GPU


NVIDIA Corporation

NVDA

$501.41 ↑30,473.78% +499.77 MAX

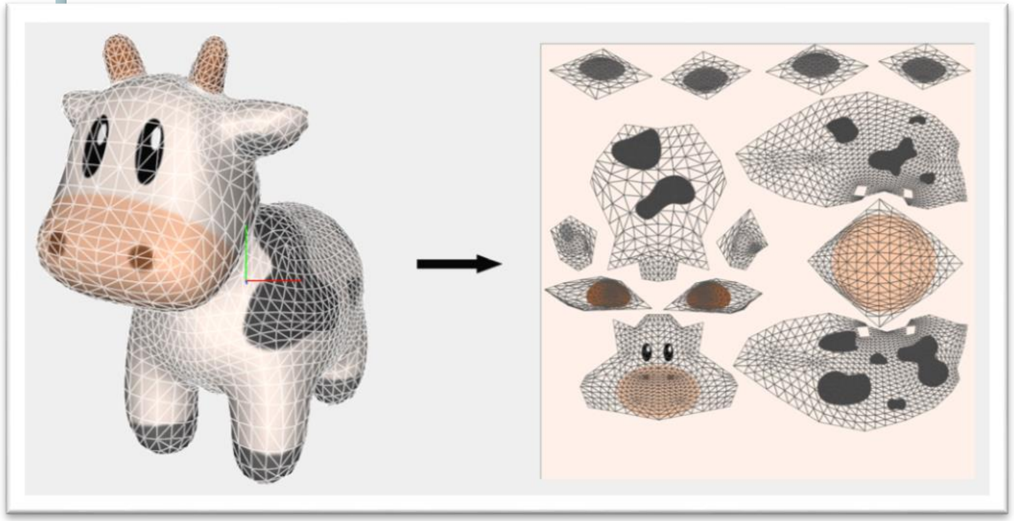
After Hours: $501.50 (↑0.018%) +$0.090

Closed: Mar 25, 5:46:41 PM UTC-4 · USD · NASDAQ · Disclaimer

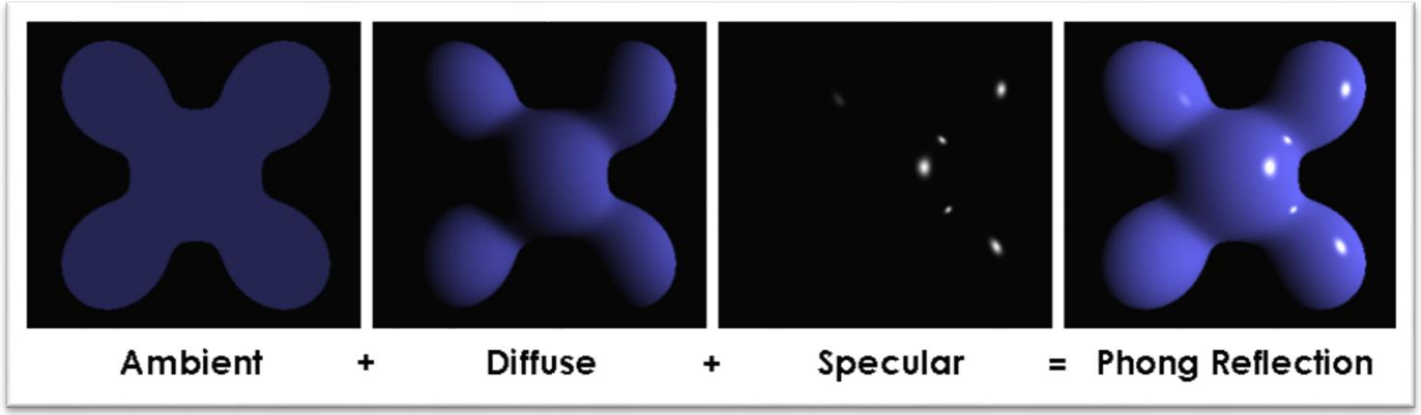
NVIDIA stock price 40x in 9 years (since deep learning became important)

Graphic Processing – Some History

- 1990s: Real-time 3D rendering for video games were becoming common
  - Doom, Quake, Descent, … (Nostalgia!)
- 3D graphics processing is immensely computation-intensive



Texture mapping



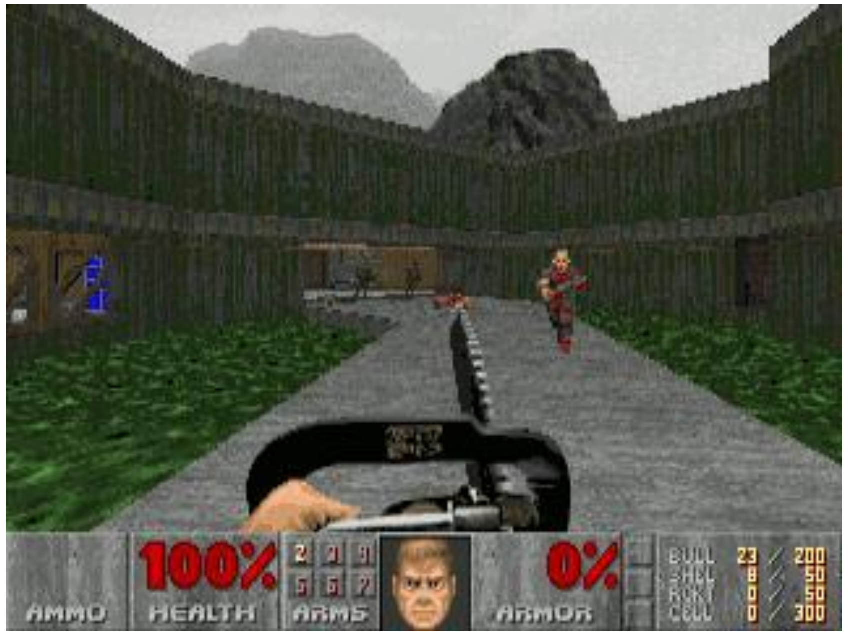
Shading

Warren Moore, “Textures and Samplers in Metal,” Metal by Example, 2014

Gray Olsen, “CSE 470 Assignment 3 Part 2 - Gourad/Phong Shading,” grayolsen.com, 2018

Graphic Processing – Some History

- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played



Doom (1993) : “Affine texture mapping”

- Linearly maps textures to screen location, disregarding depth
- Doom levels did not have slanted walls or ramps, to hide this

Graphic Processing – Some History

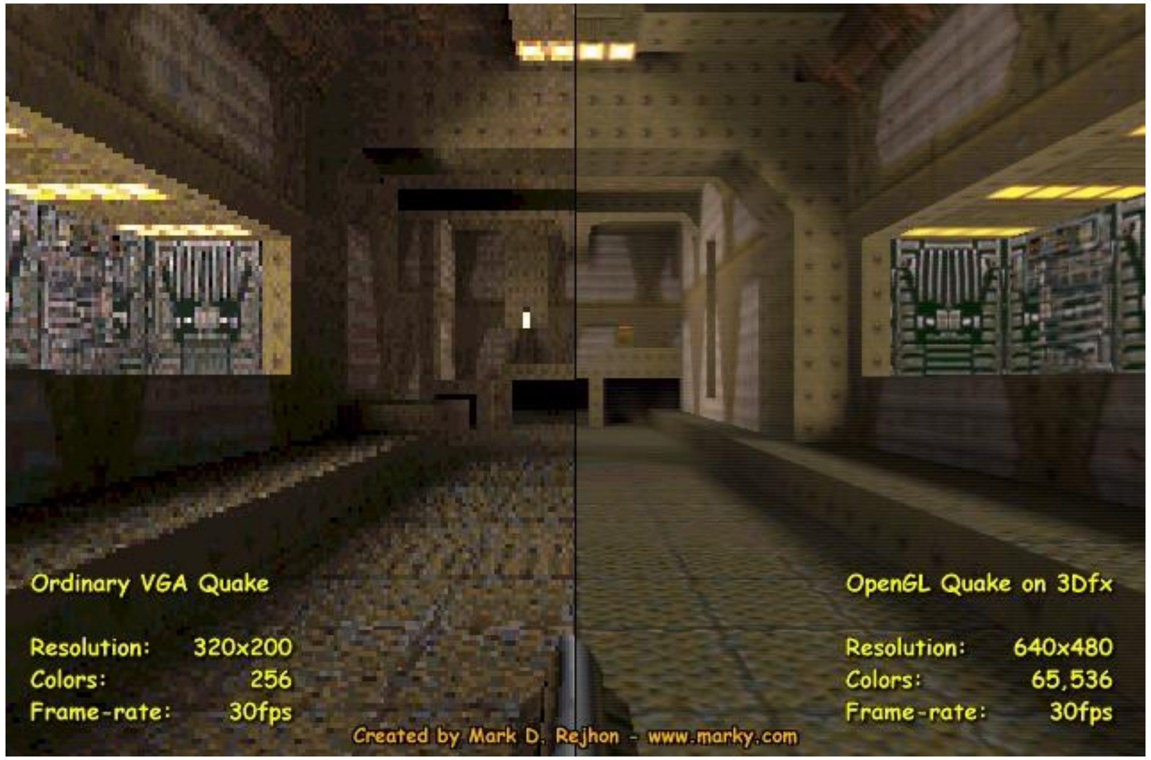
- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played

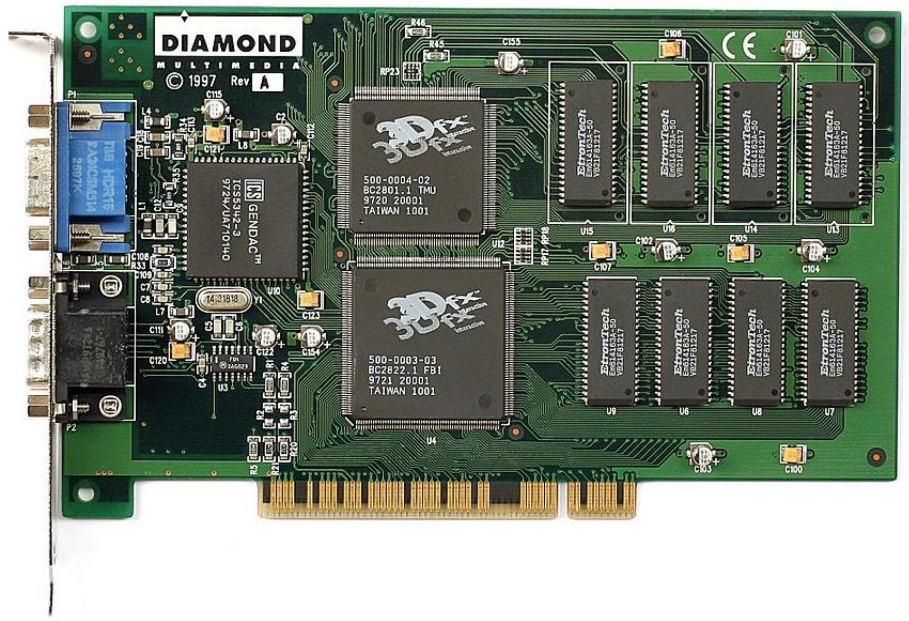
Quake III arena (1999) : “Fast inverse square root” magic!

```
float Q_rsqrt( float number )
{
    const float x2 = number * 0.5F;
    const float threehalfs = 1.5F;

    union {
        float f;
        uint32_t i;
    } conv = {number}; // member 'f' set to value of 'number'.
    conv.i  = 0x5f3759df - ( conv.i >> 1 );
    conv.f  *= ( threehalfs - ( x2 * conv.f * conv.f ) );
    return conv.f;
}
```

Introduction of 3D Accelerator Cards

- Much of 3D processing is short algorithms repeated on a lot of data
  - pixels, polygons, textures, …
- Dedicated accelerators with simple, massively parallel computation



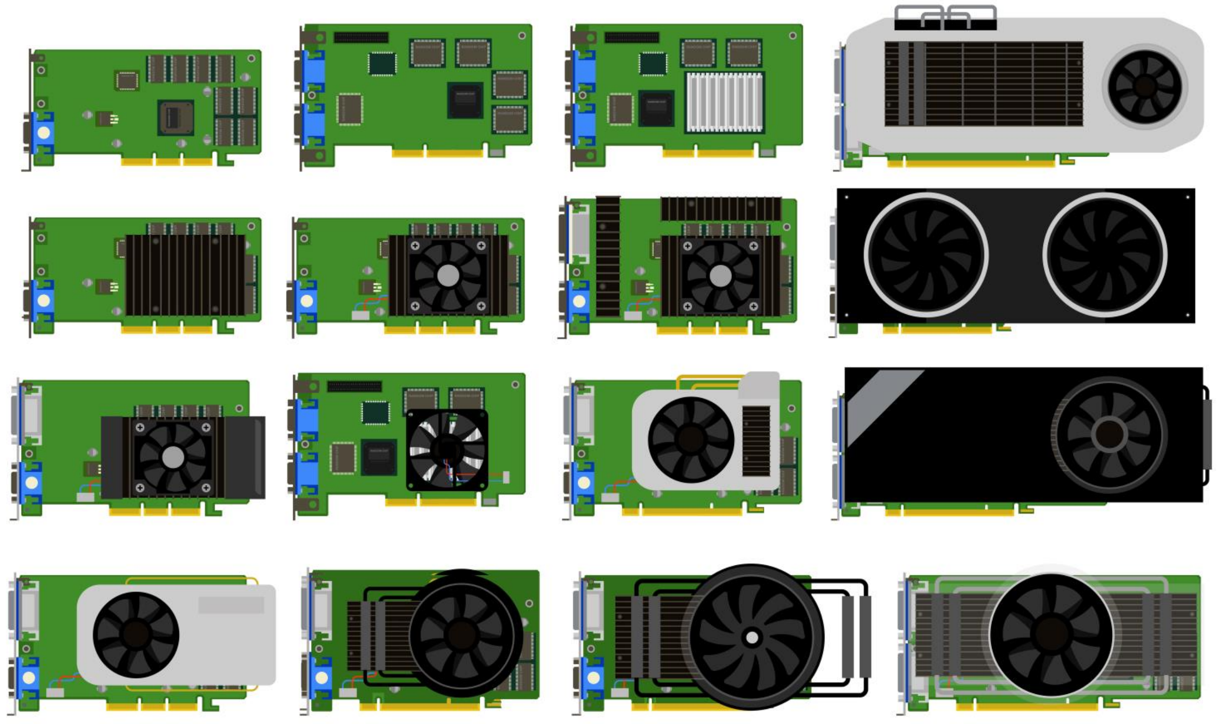


A Diamond Monster 3D, using the Voodoo chipset (1997)
(Konstantin Lanzet, Wikipedia)

General-Purpose GPUs (GP-GPUs)

- A new hardware always need to have supporting software
- In 2006, Nvidia introduced GeForce 8800 GPU, which supported a new programming language CUDA (in 2007)
  - Compute Unified Device Architecture”
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution.
- Migrating data into graphical form and then using the GPU to scan and analyze it can create a large speedup.

Evolution of GPUs



- GPUs originally developed as specialized hardware for **graphics computation**
  - Now used as programmable accelerators for highly **data-parallel** workloads
- GPU hardware evolution closely tied to evolution in usage patterns
  - Desire for improved **visual effects driven by games**
  - More realism, more effects, more screen resolution, more frames per second
- Originally a fixed-function pipeline, programmability was added gradually to many stages.
  - Now the bulk of the GPU is a programmable data-parallel architecture.
  - Some fixed-function hardware remains for graphics.
  - New hardware being added to cope with modern workloads, e.g., machine learning

https://dribbble.com/shots/8142893-Video-card-collection-Evolution-of-video-cards/attachments/562391?mode=media

GPU Categories

	Type	Vendors/Families	
Integration	integrated	NVIDIA (Geforce) AMD (Radeon)	
	dedicated	Intel (HD) AMD (APU)	
Applications	PC GPU	Intel, NVIDIA, AMD	
	Server GPU	NVIDIA (Tesla) AMD (FireStream)	
	Mobile GPU	Imagination (PowerVR) Qualcomm (Adreno) ARM (Mali) Apple A Series	

GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

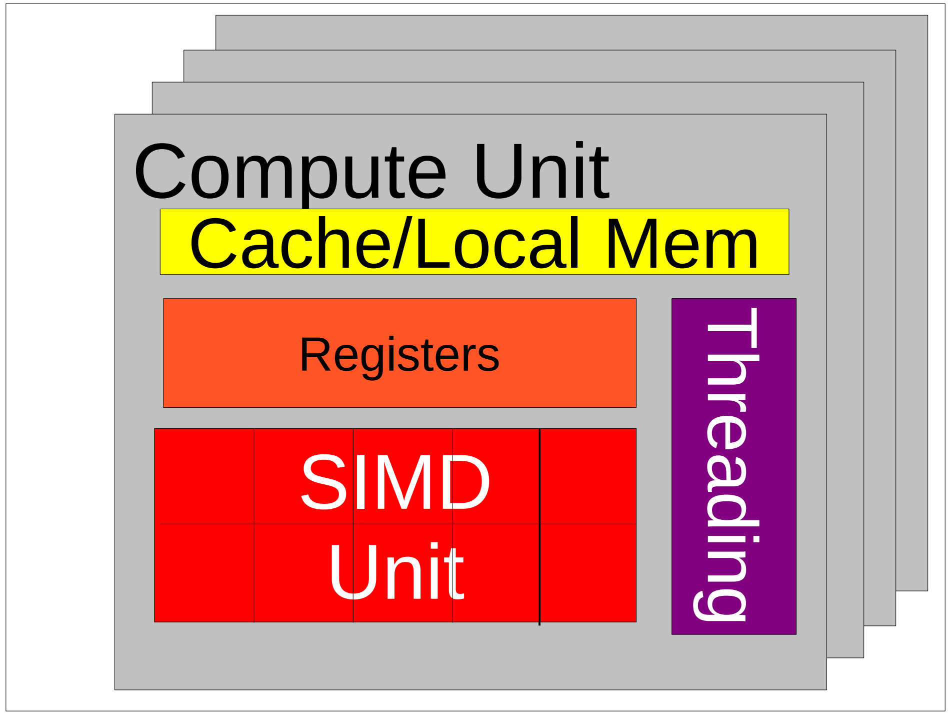
CPU and GPU have very different design philosophy

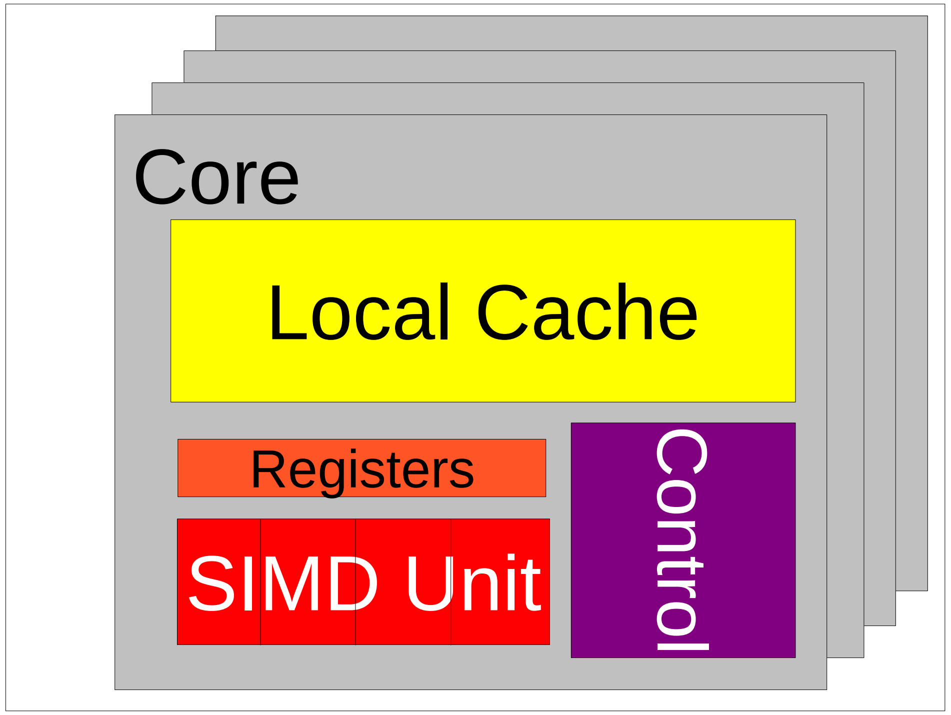
GPU

Throughput Oriented Cores

CPU

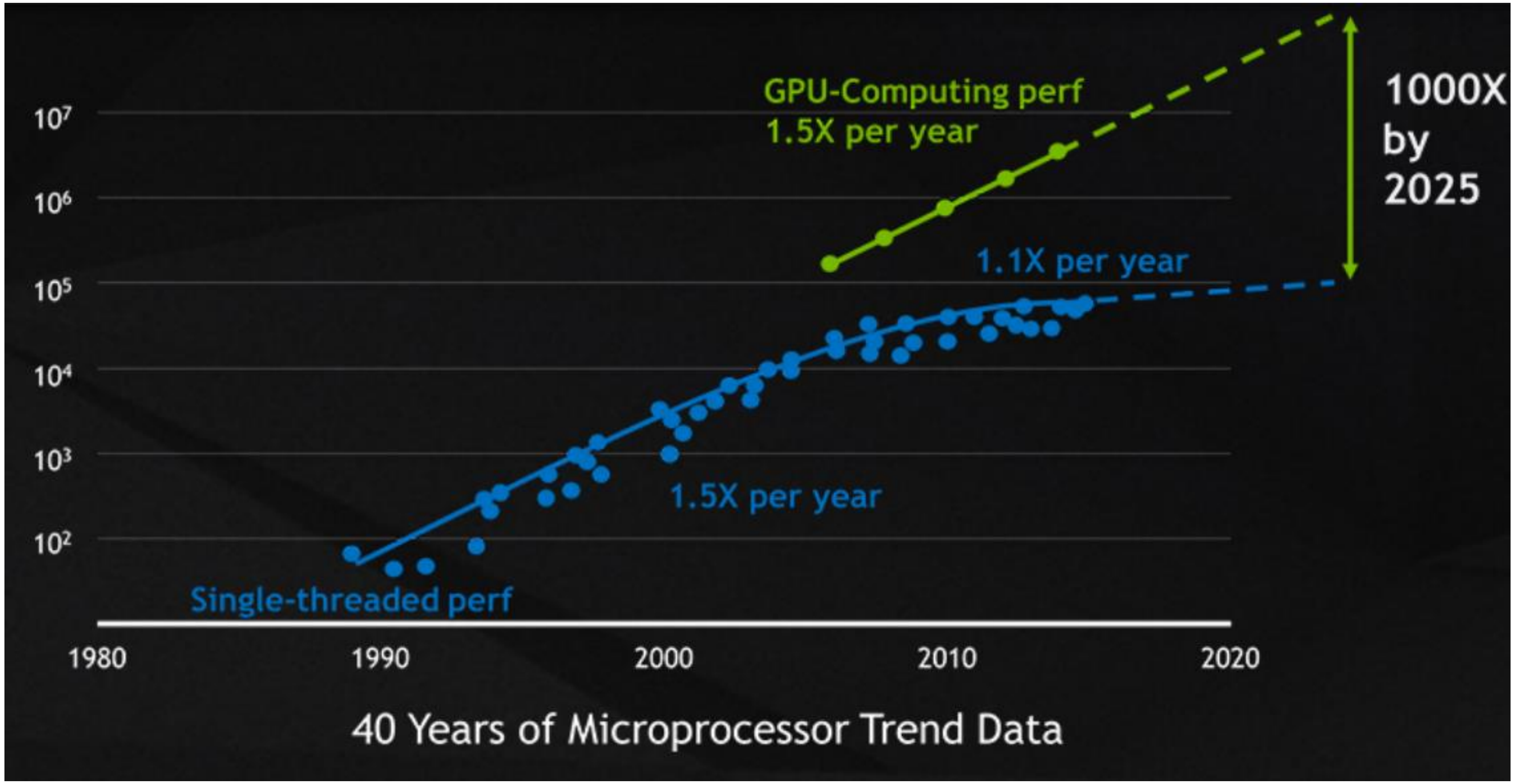
Latency Oriented Cores





Which is better depends on your needs…

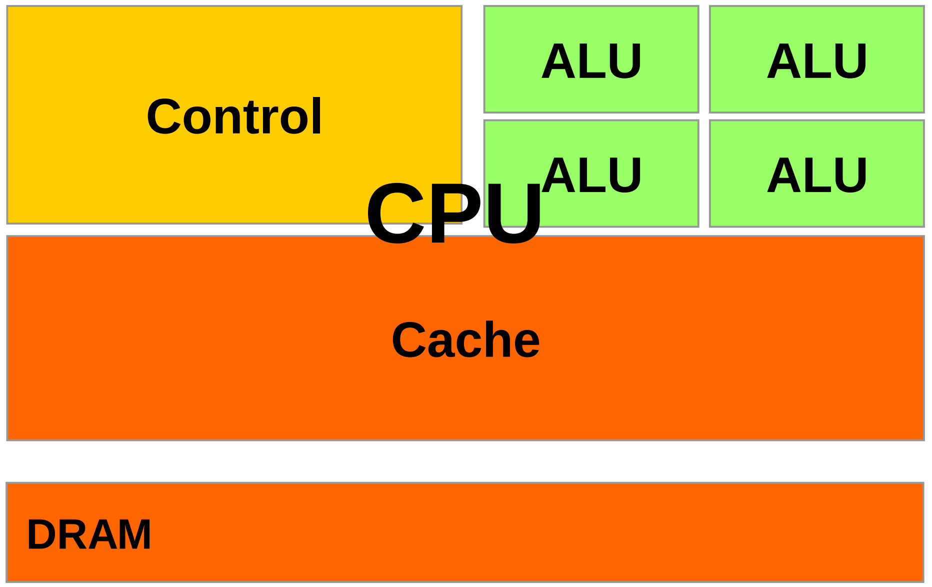
CPU vs. GPU



source: UCI CS295

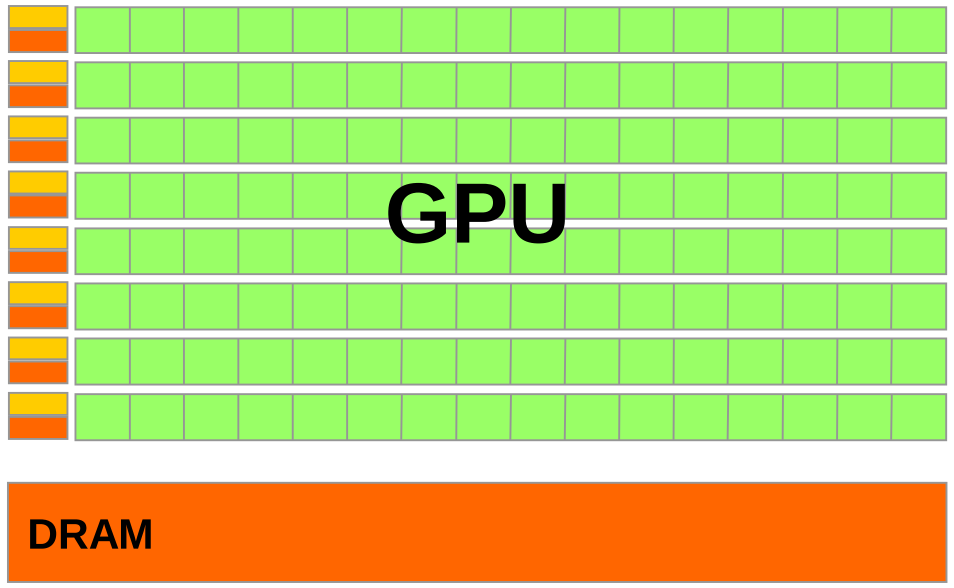
CPUs: Latency Oriented Design

- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency



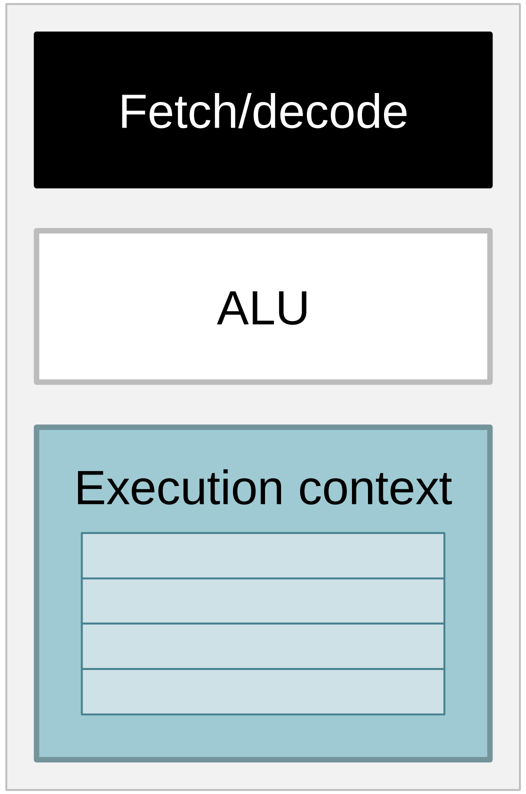
GPUs: Throughput Oriented Design

- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



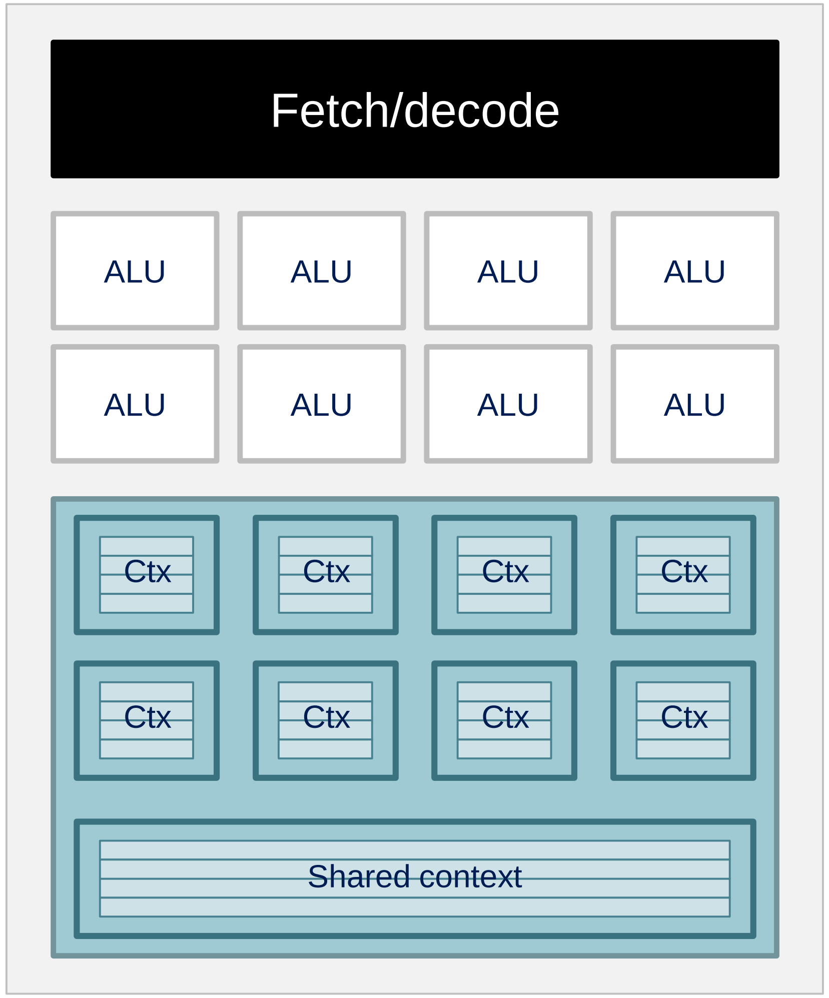
One Processing Element

- A single thread runs on a simple processing element.
- Short pipeline
- The execution context consists of the thread’s state.
  - E.g., registers and local memory
- But fetch and decode are costly.



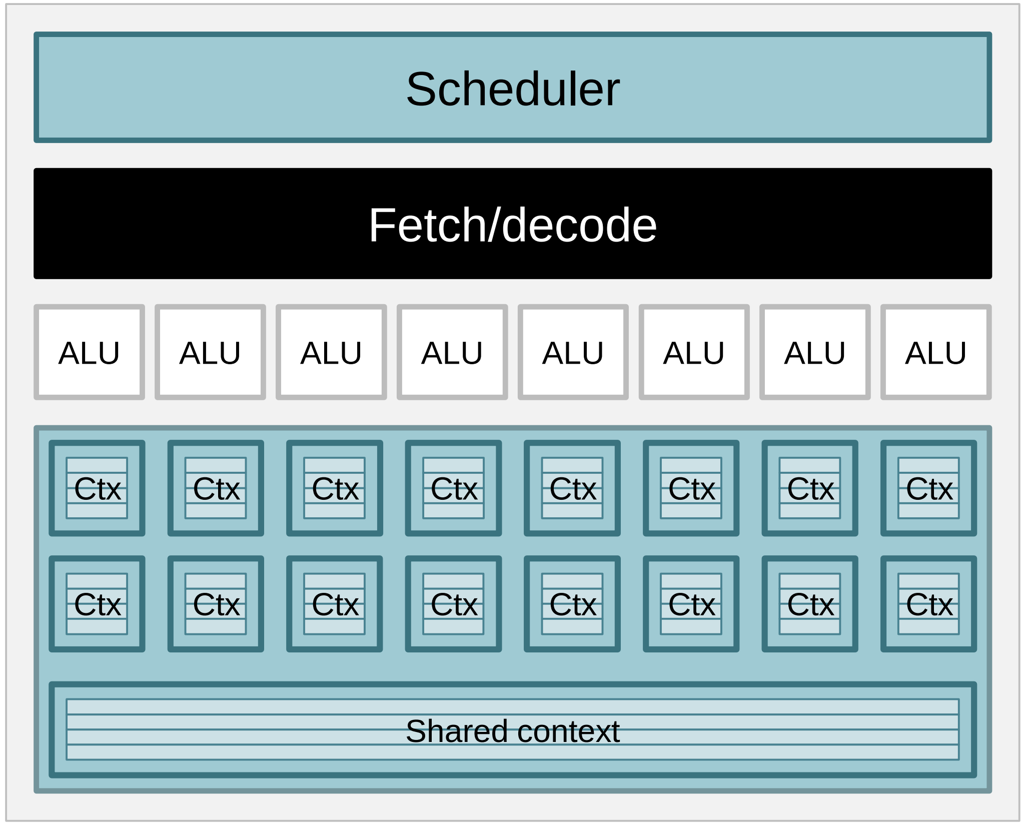
Multiple Processing Elements

- SIMT execution of threads
- Cost of fetch and decode spread across all threads in a work group (OpenCL terminology)
- Each thread has its own context.
  - And some shared context
- Multiple functional units for parallelism
  - One per thread for cheap units (e.g., simple ALU)
  - Fewer expensive units than threads (e.g., sqrt)
- However, stalls are costly.



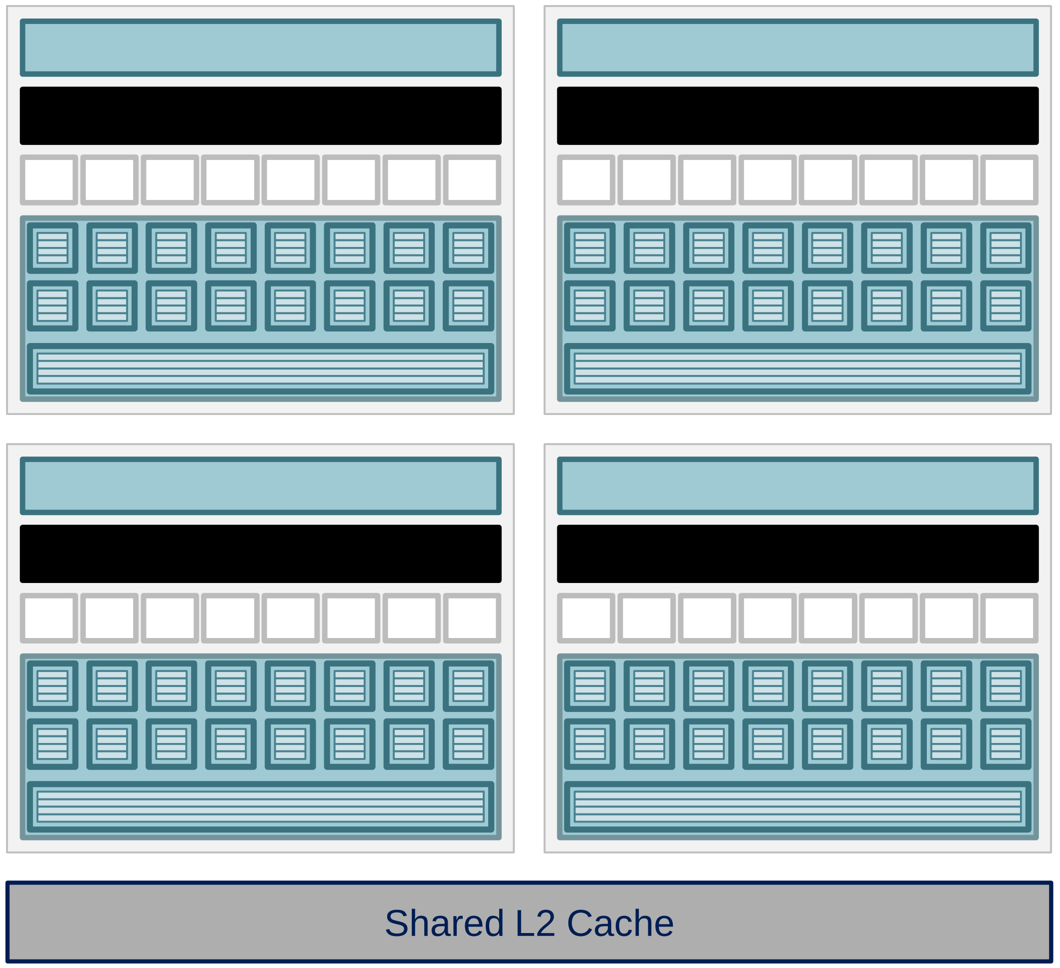
Minimizing Stalls

- Include support for multiple work groups
  - Each is independent of all others.
  - And this is guaranteed by the compiler.
  - Which means there is no fixed ordering of groups.
- A scheduler chooses work groups to run.
  - Maintains a list of ready work groups
  - Makes a choice each cycle
  - Some GPUs can schedule more than one work group per cycle.
  - Hides latency when a work group stalls
- This system is sometimes called a shader core.

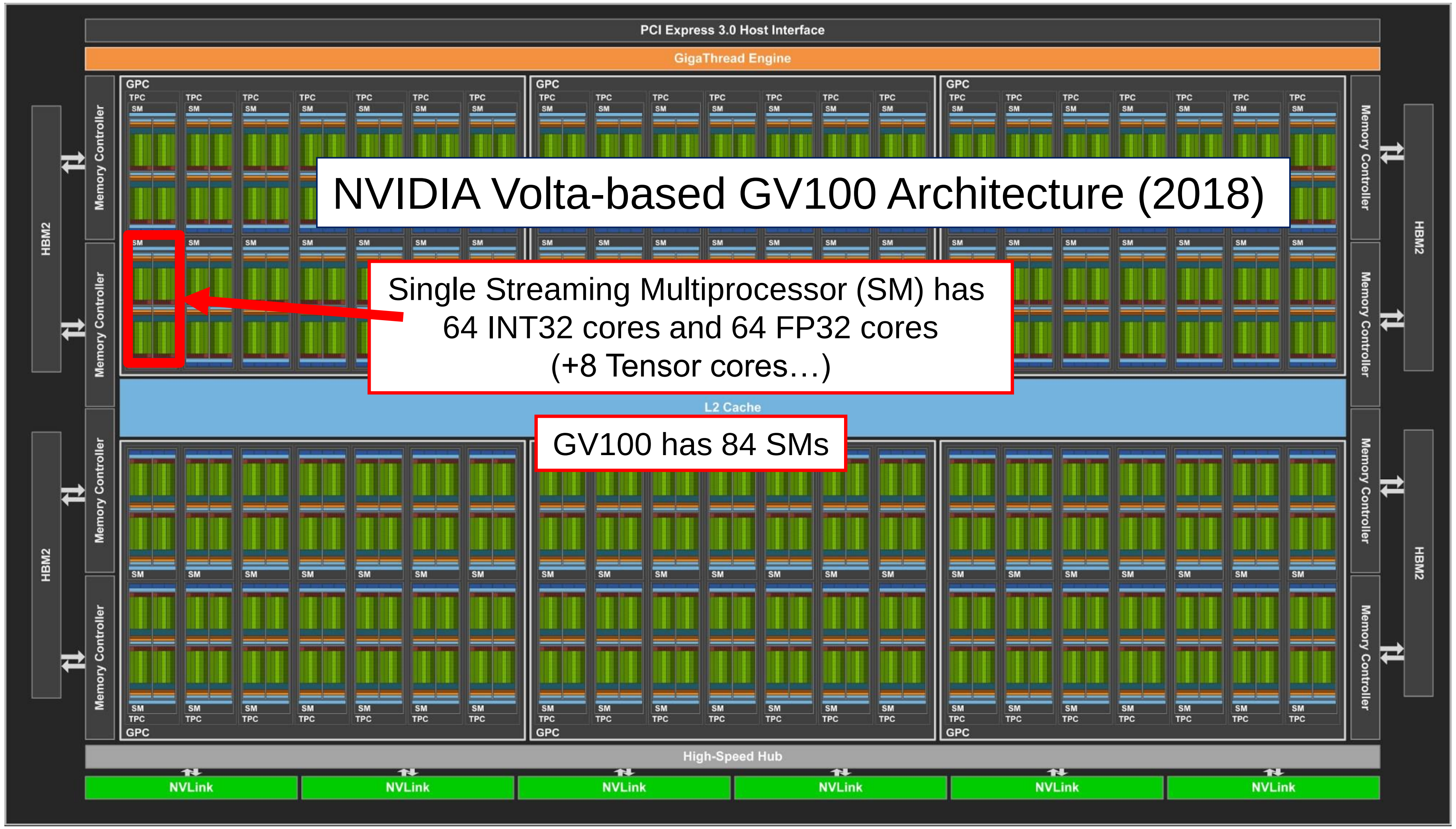


Scaling Out

- Multiple instances of each shader core provided together
  - Each independent of the others
  - Each processes a subset of the work groups
- Massively increases parallelism
- Memory hierarchy provided, too
  - Shared L2 cache reduces memory bandwidth requirements.
  - Smaller caches local to each multithreaded core

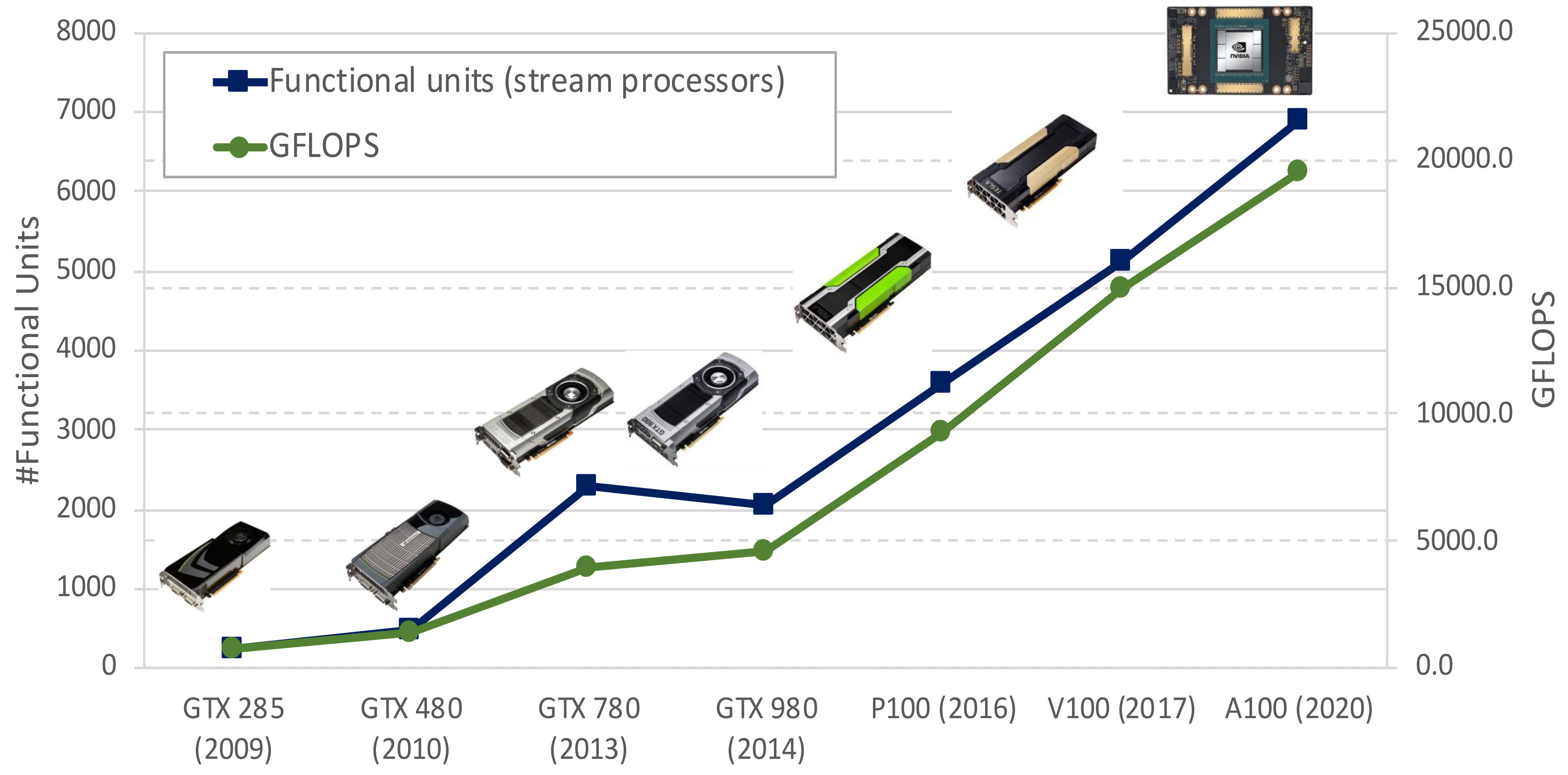


NVIDIA GV100 (2018)


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink
NVIDIA Volta-based GV100 Architecture (2018)
Single Streaming Multiprocessor (SM) has 64 INT32 cores and 64 FP32 cores (+8 Tensor cores…)
GV100 has 84 SMs

source: Nvidia

Evolution of NVIDIA GPUs



Slide Credit: Prof. Onur Mutlu, ETH

GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using threads, NOT SIMD instructions
- Before we understand that, let’s distinguish between
  - Programming Model (Software)

    vs.

  - Execution Model (Hardware)

Programming Model vs. Hardware Execution Model

- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), …
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, …
- Execution Model can be very different from the Programming Model
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (a GPU)

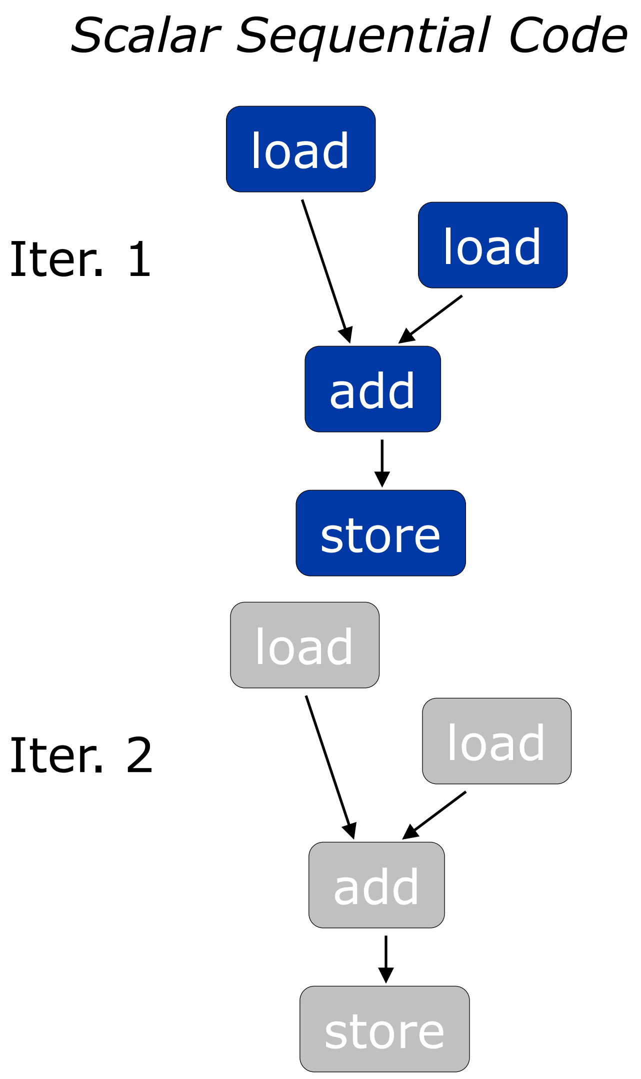
A GPU is a SIMD (SIMT) Machine

- Except it is **not** programmed using SIMD instructions

- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a **warp (wavefront)** by the hardware
  - A warp is essentially a SIMD operation formed by hardware!

How Can You Exploit Parallelism Here?



```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Let's examine three programming options to exploit instruction-level parallelism present in this sequential code:

1. Sequential (SISD)

2. Data-Parallel (SIMD)

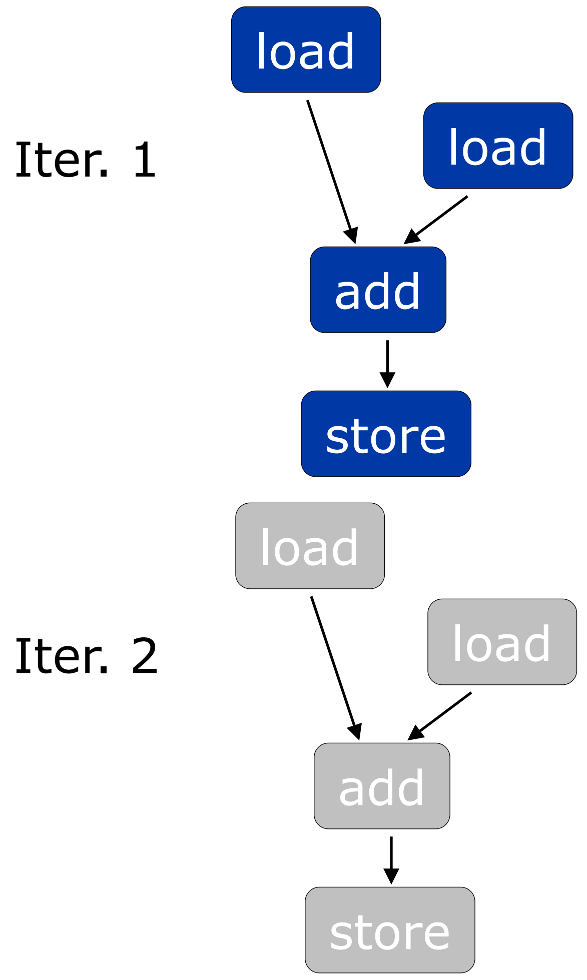
3. Multithreaded (MIMD/SPMD)

Slide Credit: Prof. Onur Mutlu, ETH

Prog. Model 1: Sequential (SISD)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Scalar Sequential Code



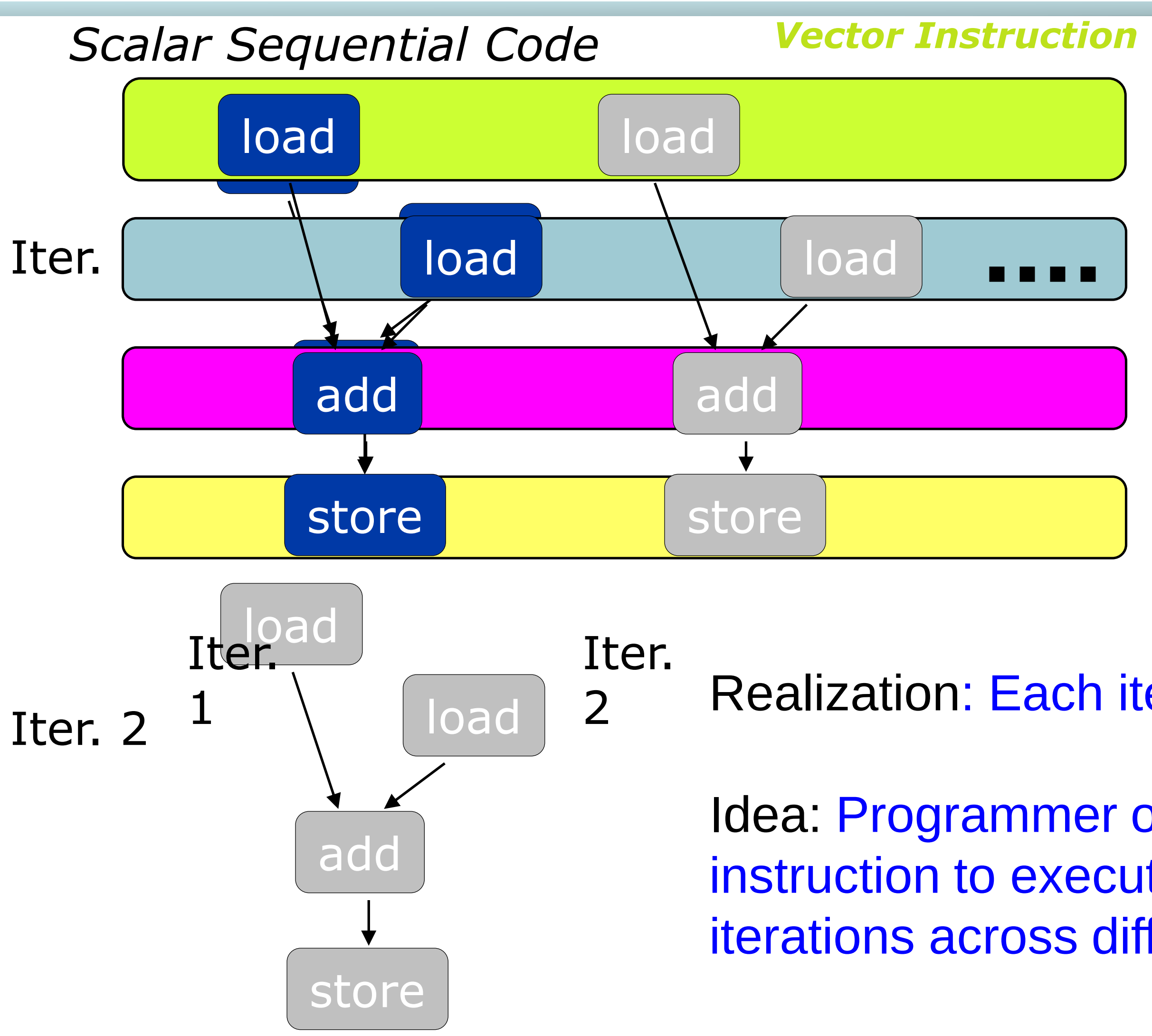
- Can be executed on a:

- Pipelined processor
- Out-of-order execution processor
  - Independent instructions executed when ready
  - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
  - In other words, the loop is dynamically unrolled by the hardware
- Superscalar or VLIW processor
  - Can fetch and execute multiple instructions per cycle

Slide Credit: Prof. Onur Mutlu, ETH

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 2: Data Parallel (SIMD)



Vectorized Code

VLD	A → V1
VLD	B → V2
VADD	V1 + V2 → V3
VST	V3 → C

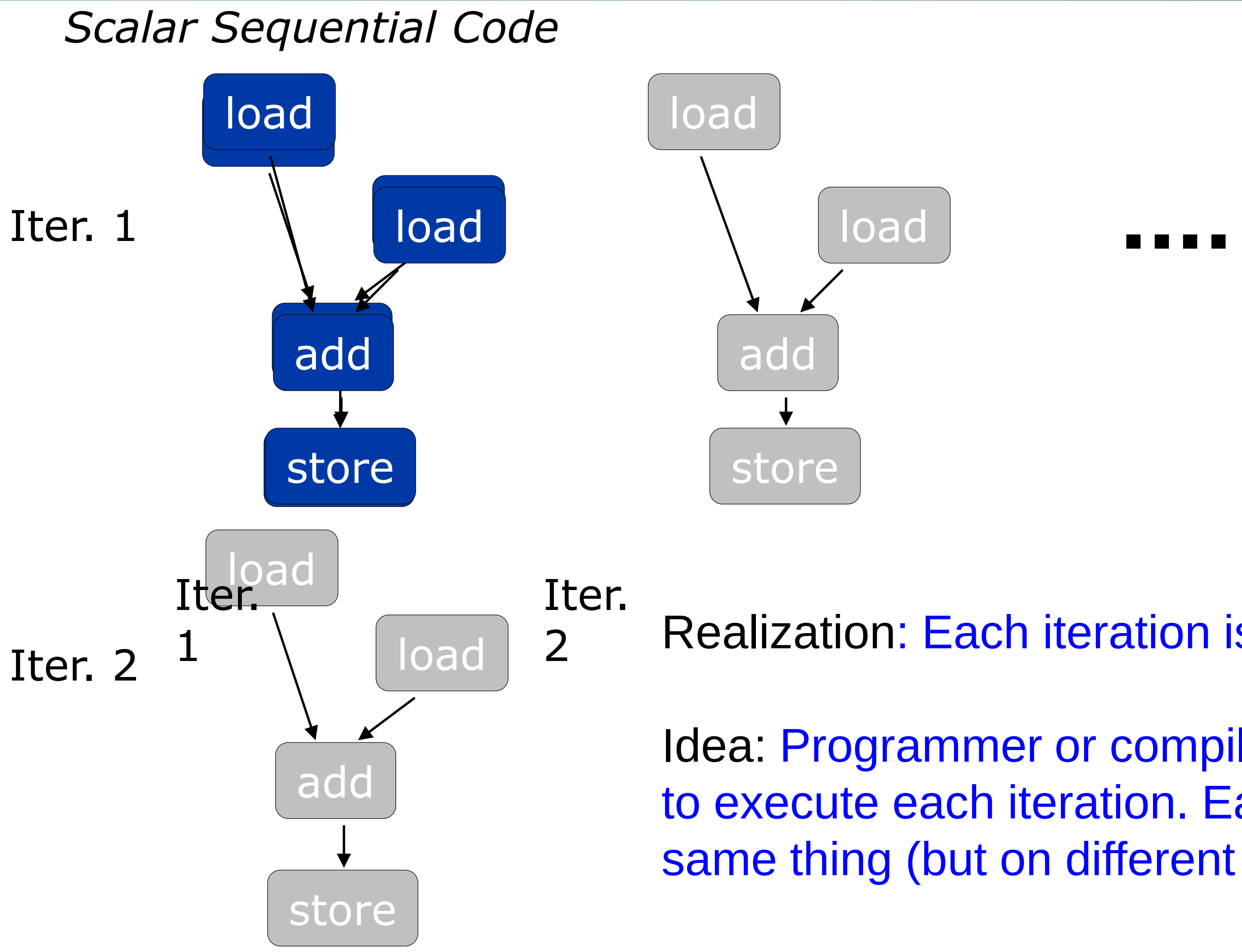
Realization: Each iteration is independent

Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)

Slide Credit: Prof. Onur Mutlu, ETH

Prog. Model 3: Multithreaded



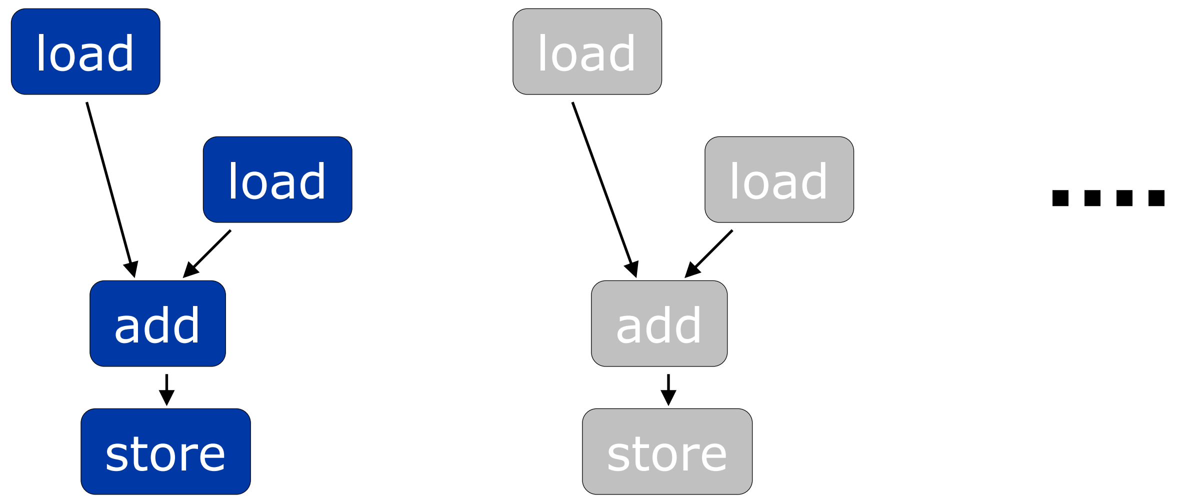
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine

Prog. Model 3: Multithreaded



```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Iter. 1

Iter. 2

Realization: Each iteration is independent

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

Slide Credit: Prof. Onur Mutlu, ETH

A GPU is a SIMD (SIMT) Machine

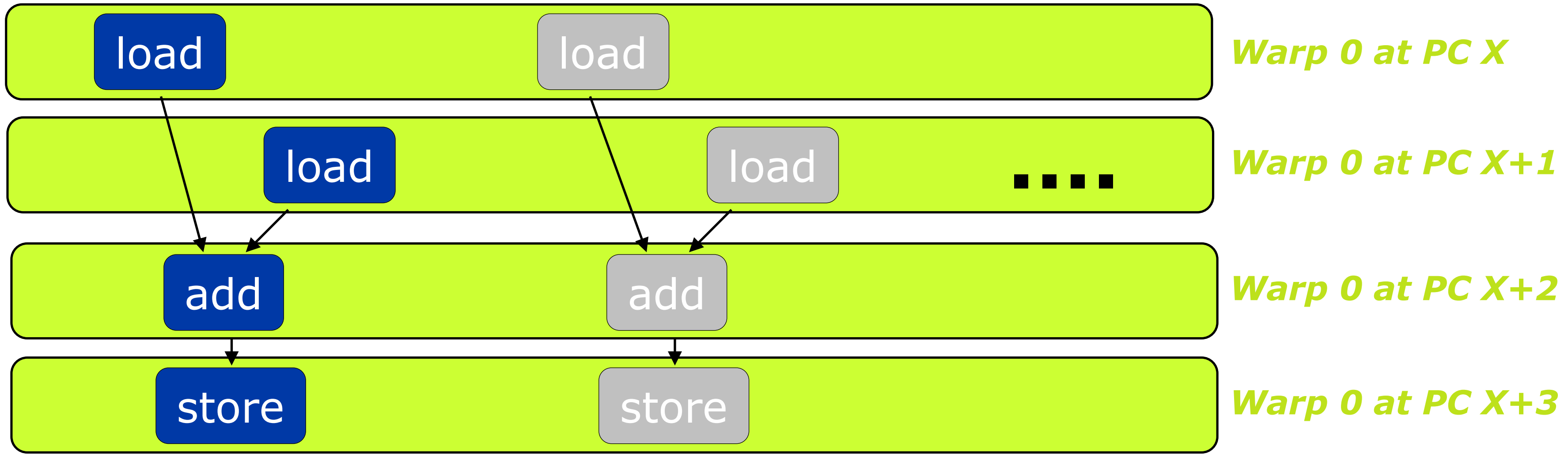
- Except it is **not** programmed using SIMD instructions

- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a **warp (wavefront)** by the hardware
  - A warp is essentially a SIMD operation formed by hardware!

SPMD on SIMT Machine

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Iter. 1

Iter. 2

Warp: A set of threads that execute the same instruction (i.e., at the same PC)

This particular model is also called:

SPMD: Single Program Multiple Data

A GPU executes it using the SIMT model: Single Instruction Multiple Thread

Slide Credit: Prof. Onur Mutlu, ETH

JOINT INSTITUTE 交大密西根学院

SIMD vs. SIMT Execution Model

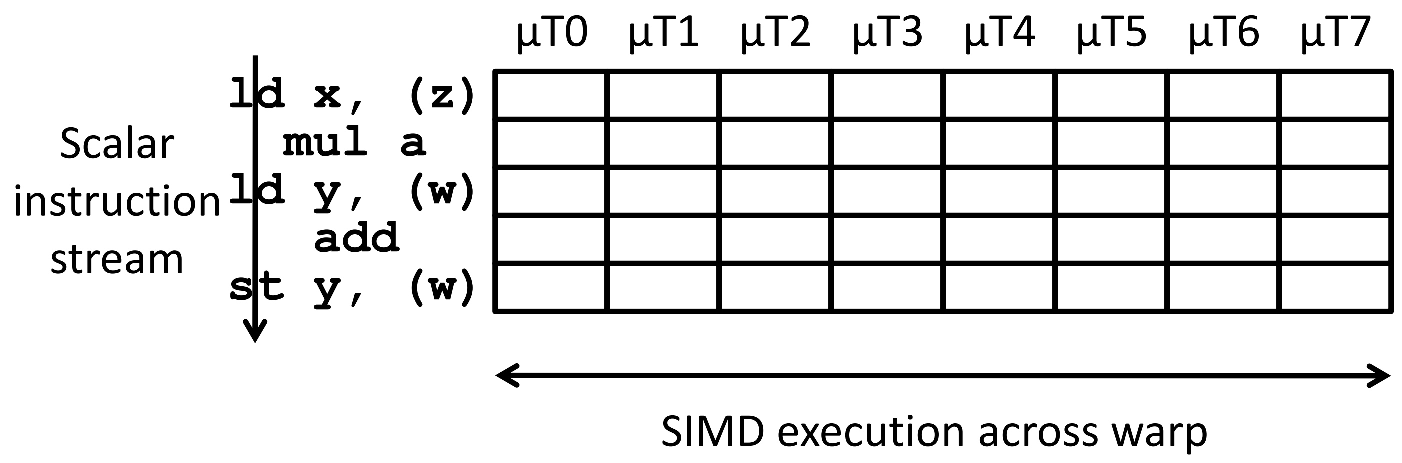
- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

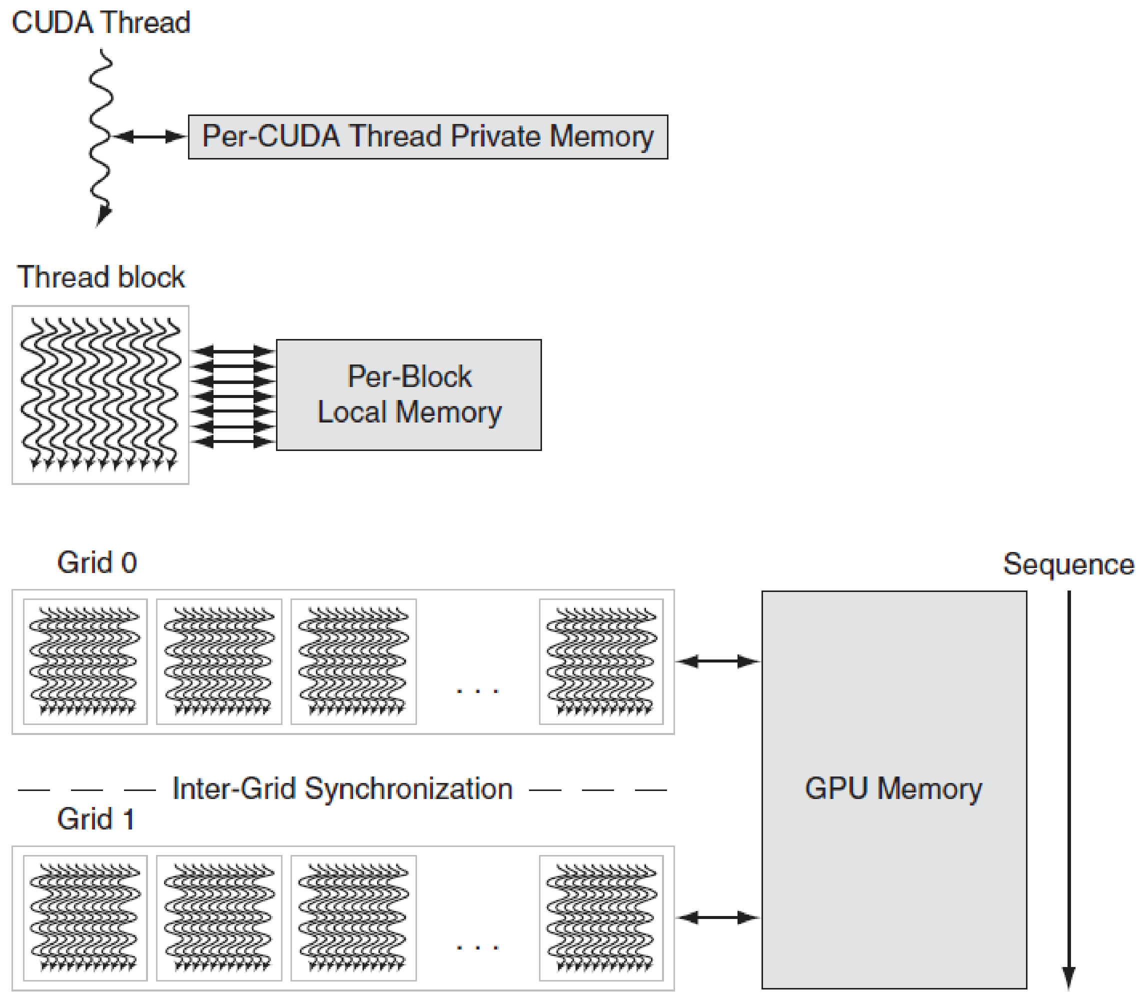
"Single Instruction, Multiple Thread" (SIMT)

- GPUs use a SIMT model, where individual scalar instruction streams for each **CUDA** thread are grouped together for SIMD execution on hardware (NVIDIA groups 32 CUDA threads into a *warp*)



GPU Memory Structures

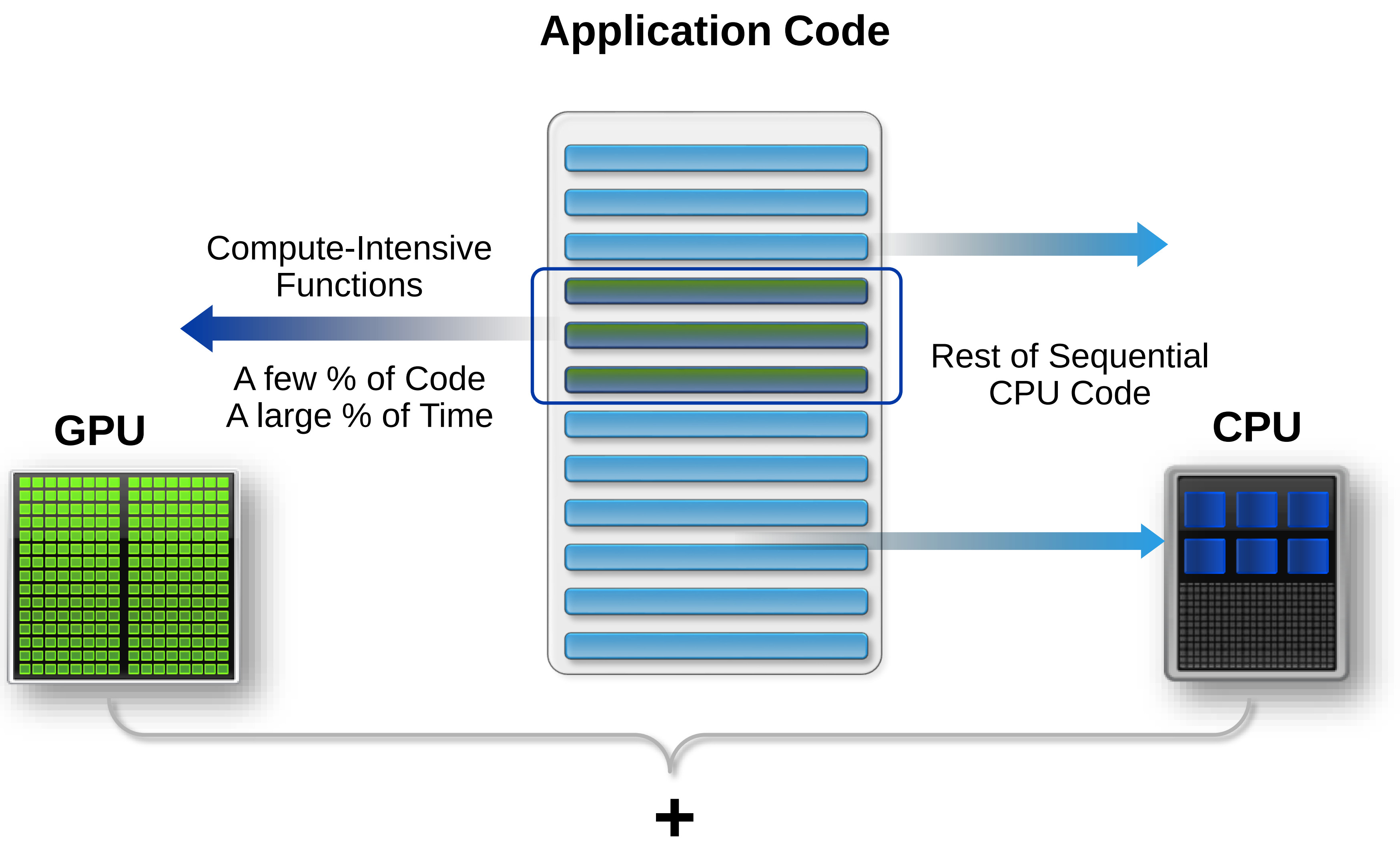
- GPUs traditionally use smaller streaming caches and rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM



CPU + GPU

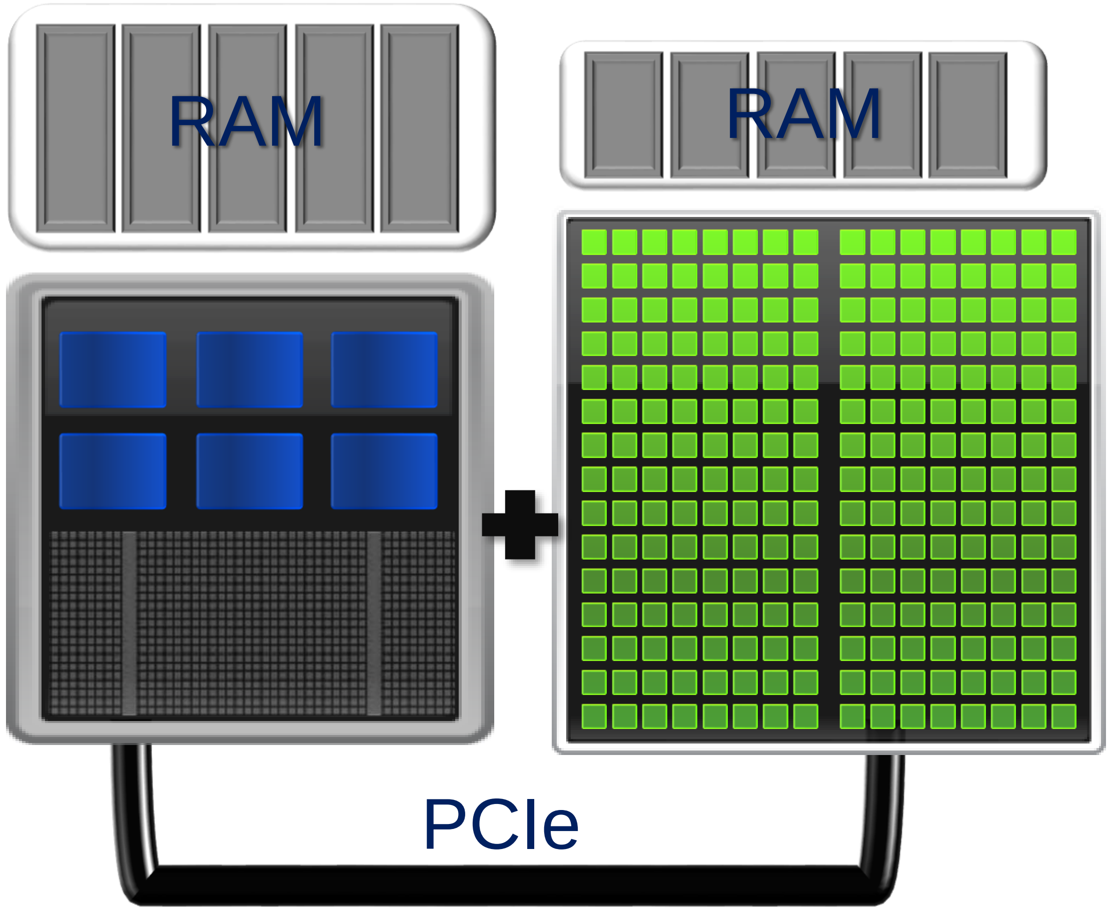
- GPU will not perform well on tasks on which CPUs are design to perform well. For program that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs.
- When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs. Many applications use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.

CPU+GPU (Heterogeneous Programming)



Slide adapted from Nvidia Accelerated Computing Teaching Kit

Accelerator Nodes



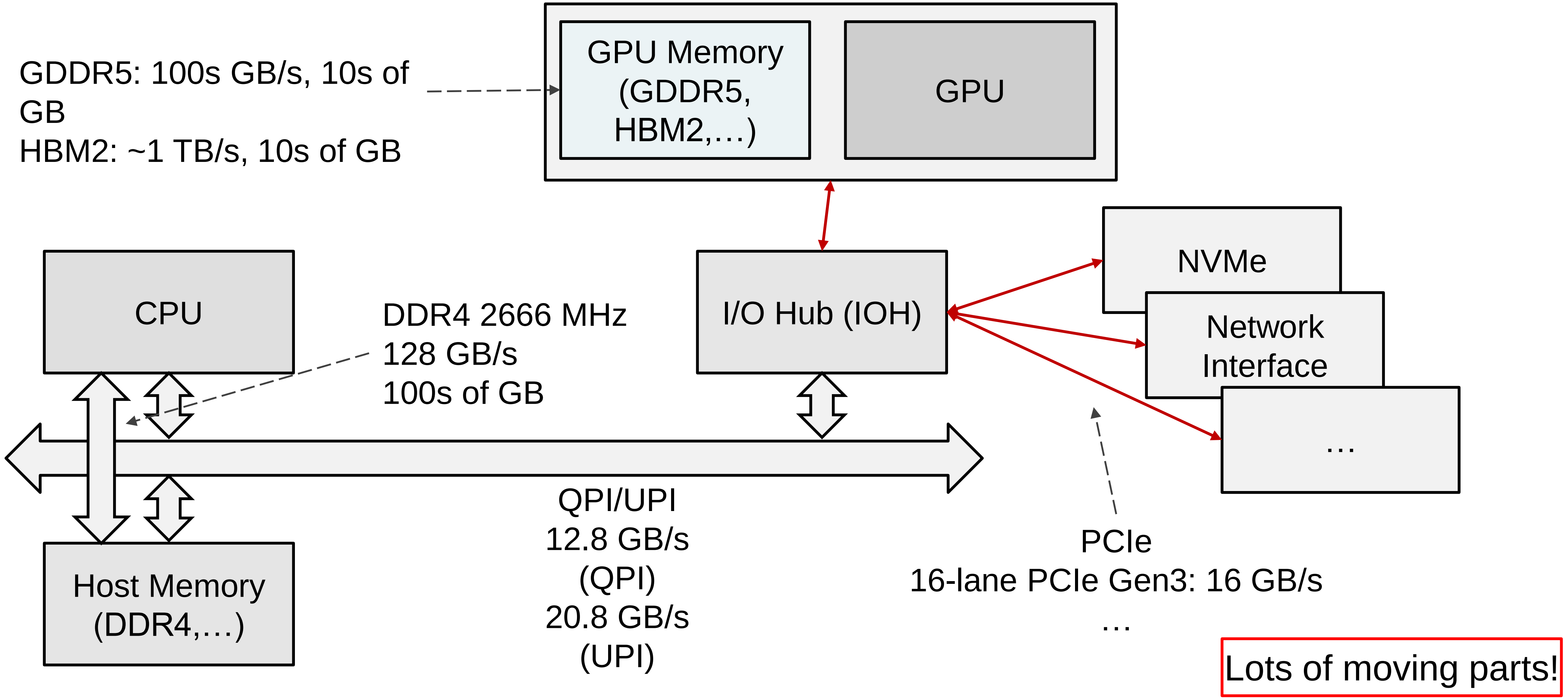
CPU and GPU have distinct memories

- CPU generally larger and slower
- GPU generally smaller and faster

CPU and GPU communicate via PCIe

- PCIe: Peripheral Component Interconnect Express
- Data must be copied between these memories over PCIe
- PCIe Bandwidth is much lower than either memories

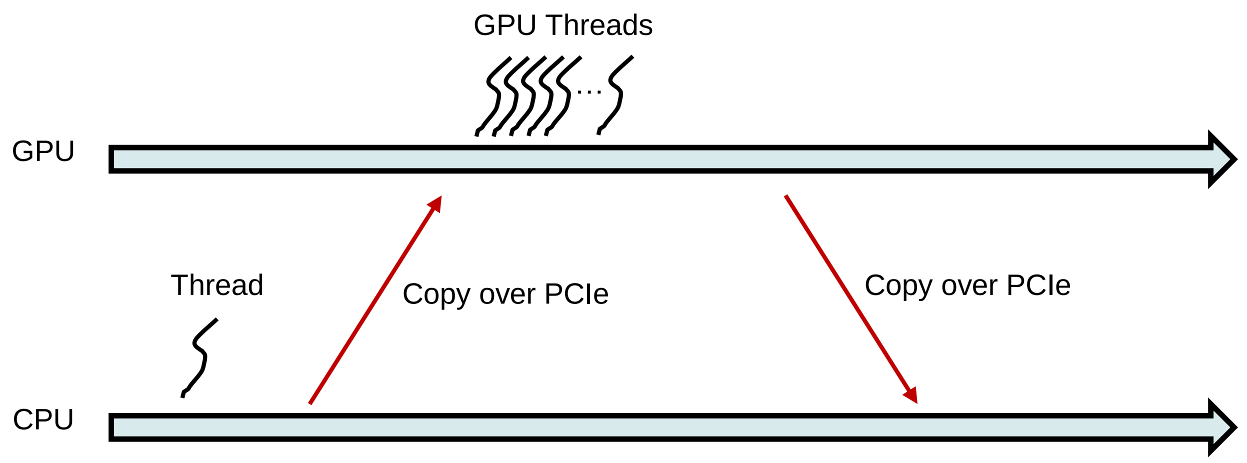
System Architecture Snapshot With a GPU (2019)



Adapted from Prof. Sang-Woo Jun@UCI, CS259

CUDA

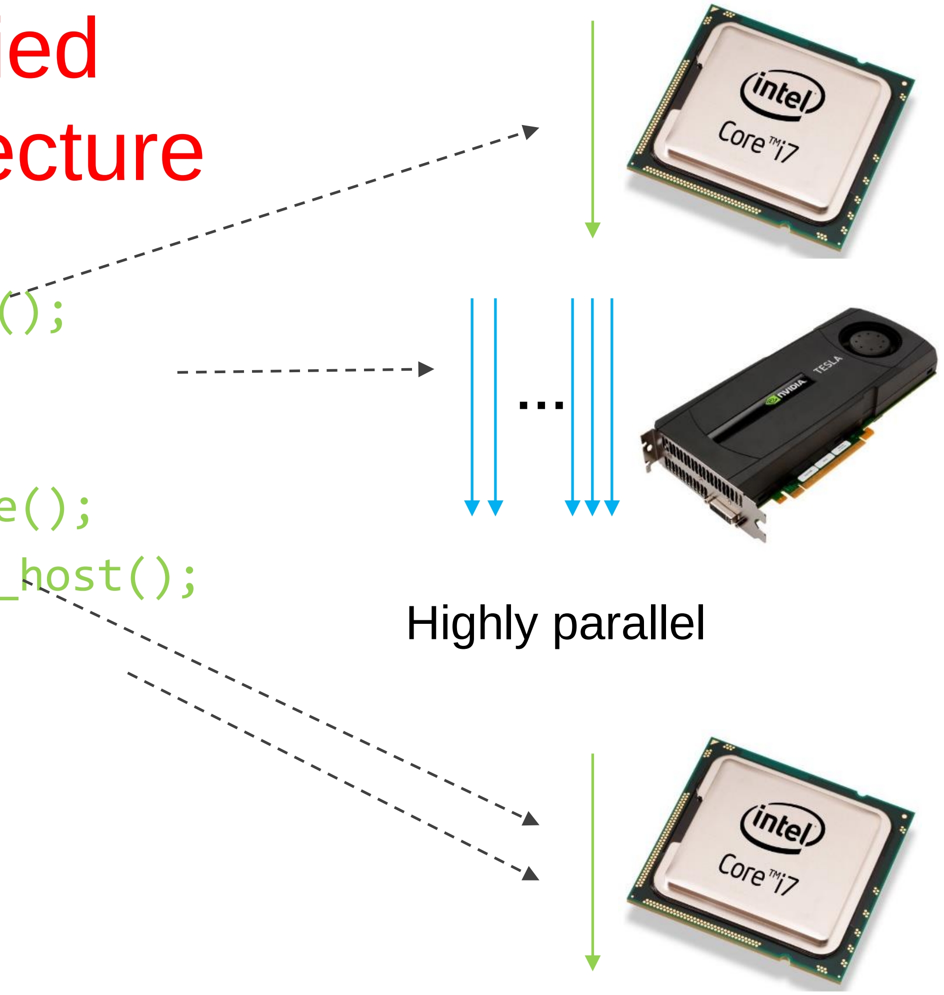
- NVIDIA **CUDA** (Compute Uniform Device Architecture) – 2007
  - A way to run custom programs on the massively parallel architecture!
- OpenCL specification released – 2008
- Both platforms expose synchronous execution of a massive number of threads



Programming using CUDA (high level)

- Compute Unified Device Architecture

```
do_something_on_host();
kernel<<<nBlk, nTid>>>(args);
cudaDeviceSynchronize();
do_something_else_on_host();
```



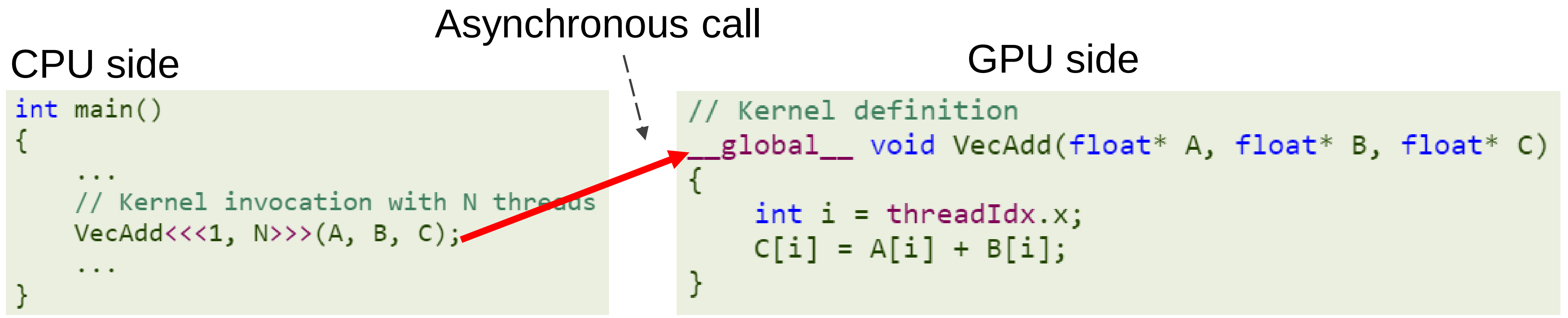
source: Shuang Zhao, Cornell University, 2014

Simplified CUDA Programming Model

- Computation performed by a very large number of independent small scalar threads (*CUDA threads* or *microthreads*) grouped into *thread blocks.*

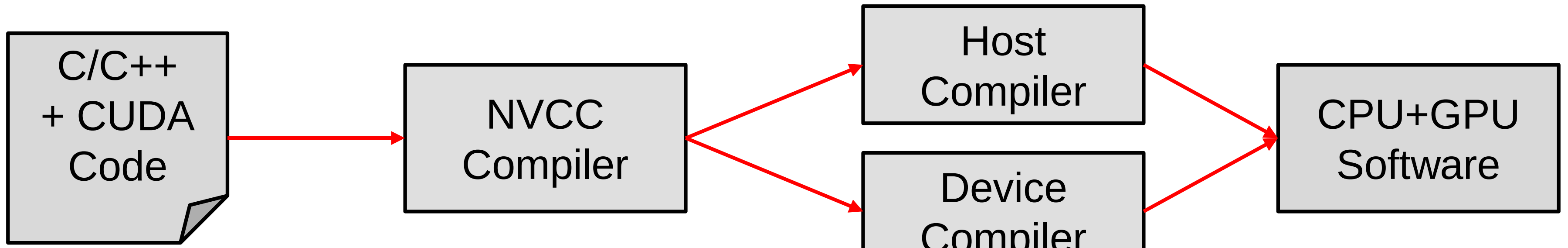
```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{  for (int i=0; i<n; i++)
      y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__  // Piece run on host processor.
int nblocks = (n+255)/256; //256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__  // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{  int i = blockIdx.x*blockDim.x + threadId.x;
   if (i<n) y[i]=a*x[i]+y[i]; }
```

Simple CUDA Example



```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```



Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100000; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

CUDA code

```
// there are 100000 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

Slide credit: Hyesoon Kim

Sample GPU Program (Less Simplified)

CPU Program

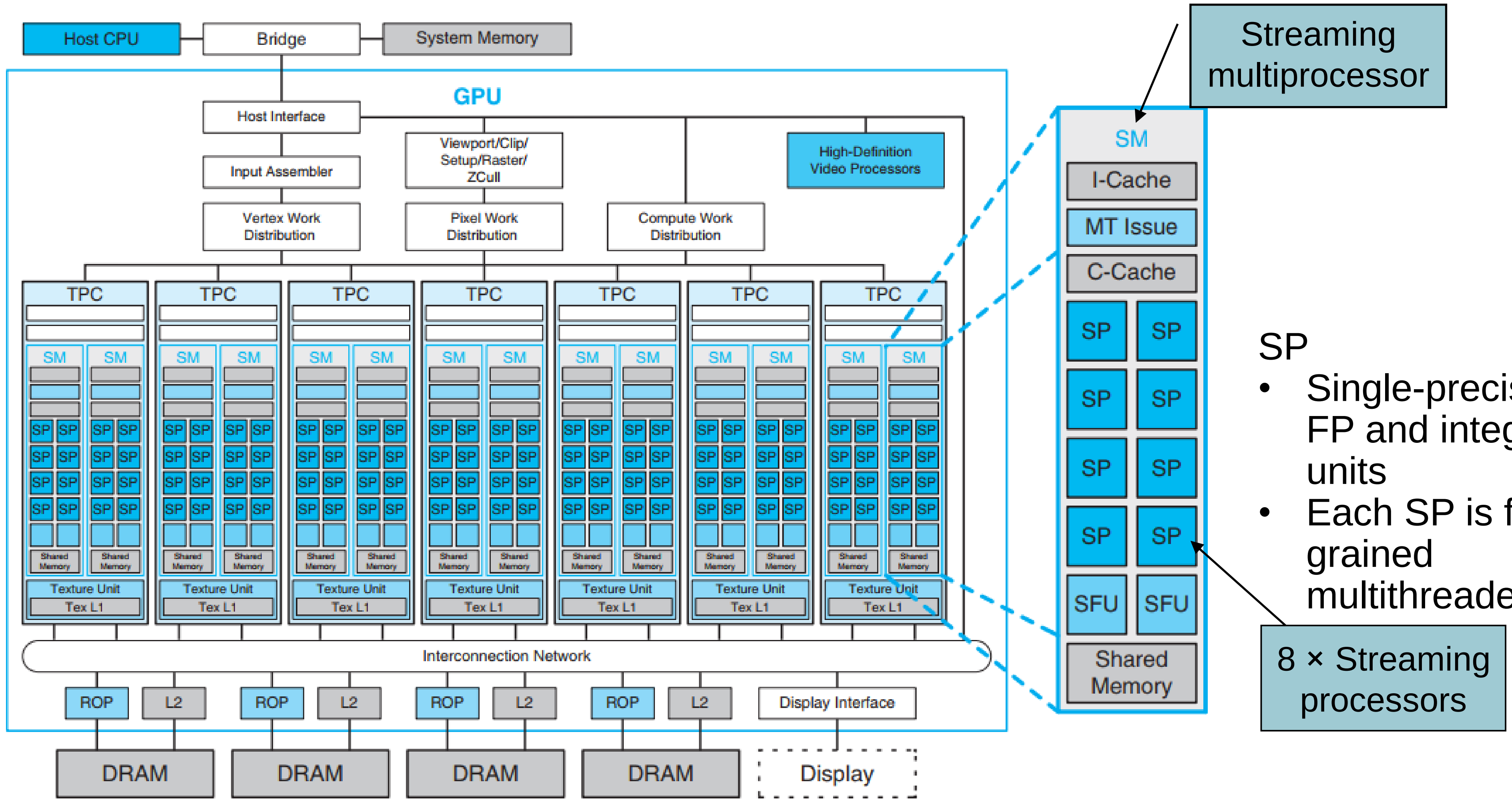
```
void add matrix
 ( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
        index = i + j*N;
        c[index] = a[index] + b[index];
       }
 }

int main () {

   add matrix (a, b, c, N);
}
```

GPU Program

```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

Int main() {
  dim3 dimBlock( blocksize, blocksize) ;
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

Slide credit: Hyesoon Kim

NVIDIA Tesla



SP
- Single-precision FP and integer units
- Each SP is fine-grained multithreaded

- 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs)
- Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory

Clarification of Some GPU Terms

Generic Term	NVIDIA Term	AMD Term	Comments
Vector length	Warp size	Wavefront size	Number of threads that run in parallel (lock-step) on a SIMD functional unit
Pipelined functional unit / Scalar pipeline	Streaming processor / CUDA core	-	Functional unit that executes instructions for one GPU thread
SIMD functional unit / SIMD pipeline	Group of N streaming processors (e.g., N=8 in GTX 285, N=16 in Fermi)	Vector ALU	SIMD functional unit that executes instructions for an entire warp
GPU core	Streaming multiprocessor	Compute unit	It contains one or more warp schedulers and one or several SIMD pipelines

Summary: GPU Design Principles

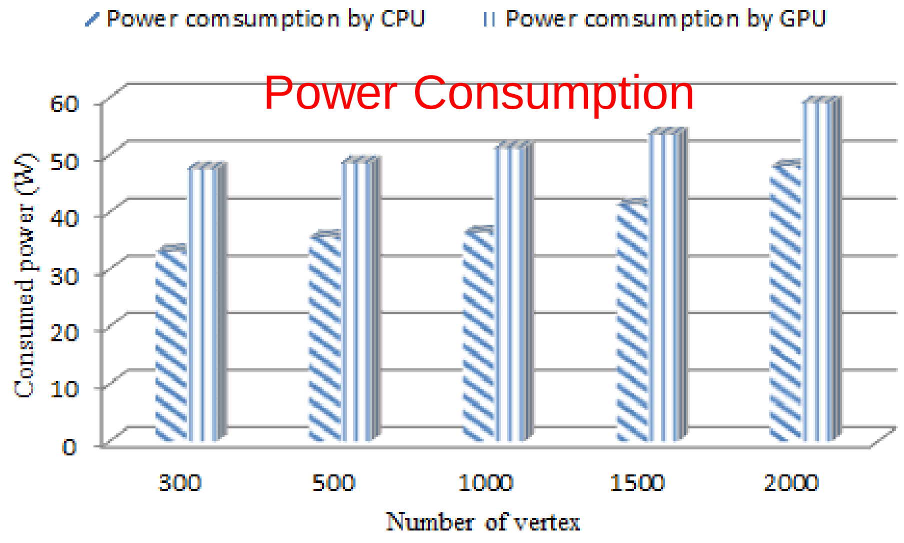
- CPU design is about making a single thread run as fast as possible.
  - Pipeline stalls and memory accesses are expensive in terms of latency.
  - So increased logic was added to reduce the probability/cost of stalls.
  - Use of large cache memories to avoid memory misses
- GPU design is about maximizing computation throughput.
  - Individual thread latency not considered important
  - GPUs avoid much of the complex CPU pipeline logic for extracting ILP.
  - Instead, each thread executes on a relatively simple core with performance obtained through parallelism.
    - Single instruction, multiple threads
- Computation hides memory and pipeline latencies.
- Wide and fast bandwidth-optimized memory systems

GPU from embedded to cloud

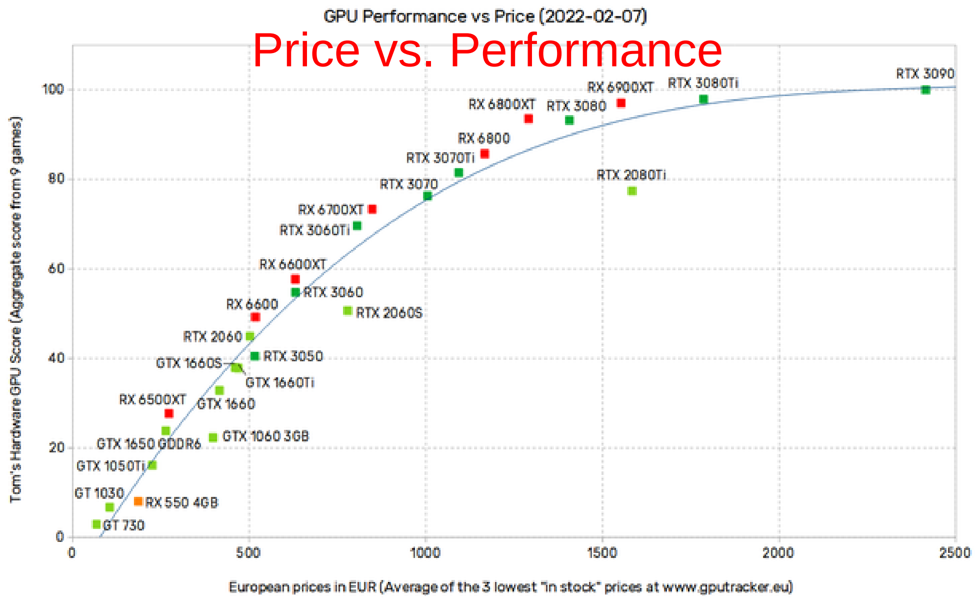
Edge

Cloud

Power Consumption



Price vs. Performance



Performance

Price

Photo Credit: ARM, Nvidia, "High performance computing for large graphs of internet applications using GPU"

Where are we Heading?

- T7: Multicore

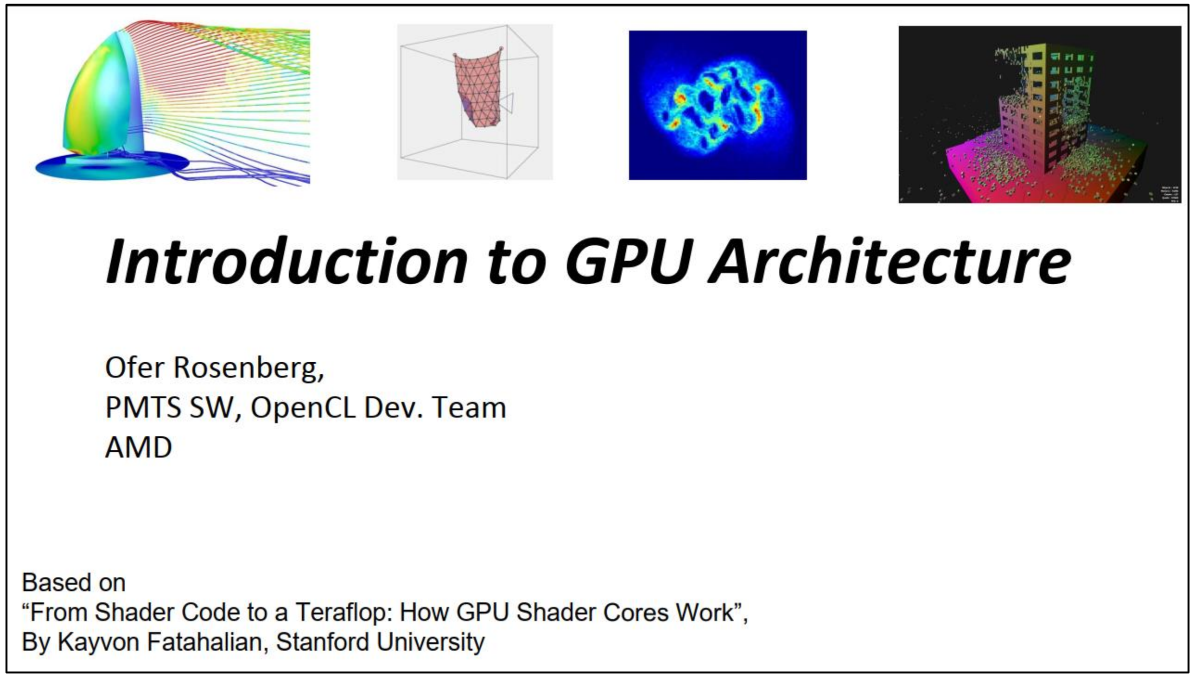
Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Nvidia Courseware
- Prof. Onur Mutlu @ ETH
- Prof. Joe Devietti @ Upenn, CIS 571
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

Action Items

- Final project
- Reading Materials
  - Ch. 4.4
  - “Introduction to GPU Architecture” Slides (available on canvas reading)



Introduction to GPU Architecture

Ofer Rosenberg,
PMTS SW, OpenCL Dev. Team
AMD

Based on
“From Shader Code to a Teraflop: How GPU Shader Cores Work”,
By Kayvon Fatahalian, Stanford University