GRAPHICS PROCESSING UNIT

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah

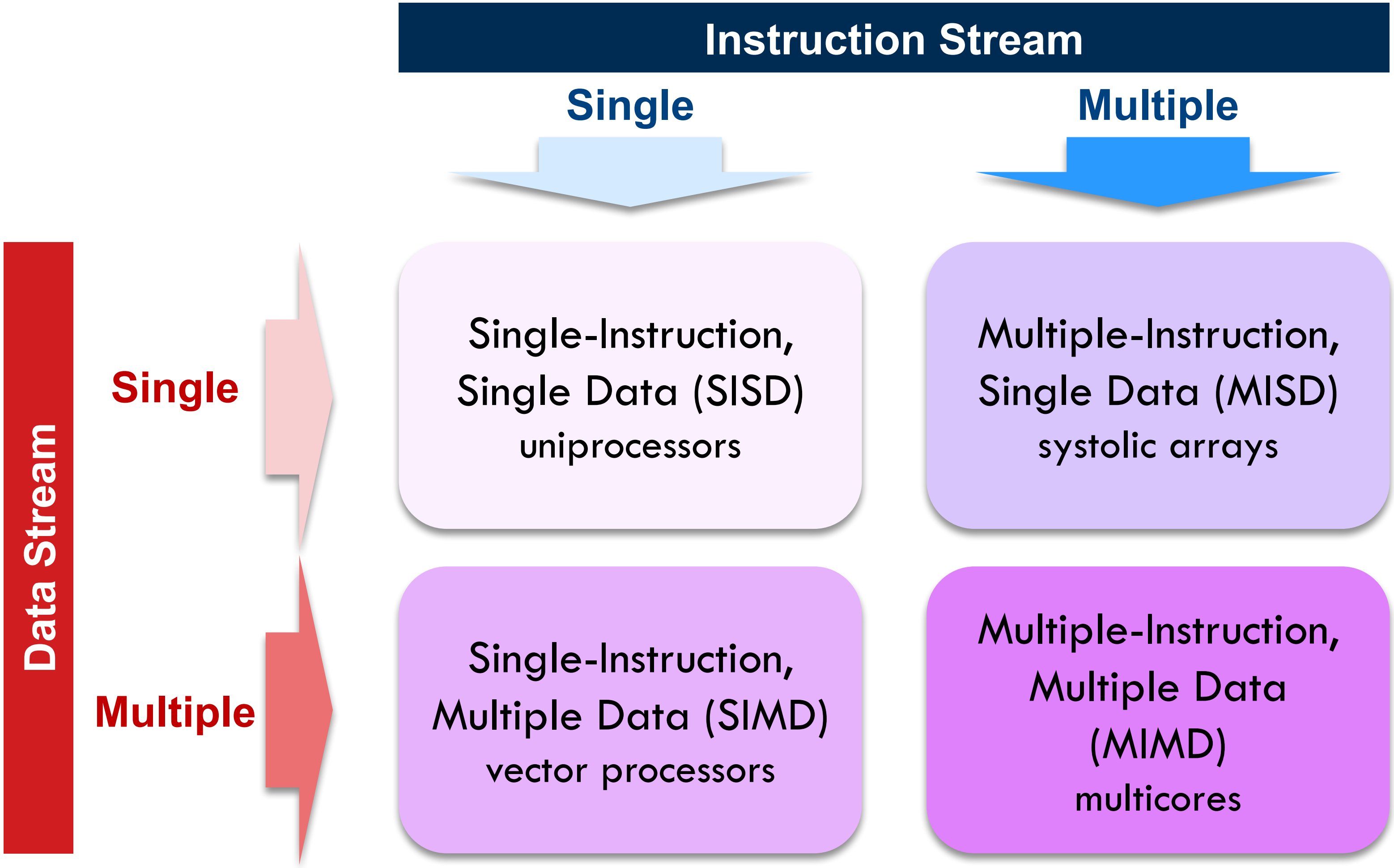
Overview

- Announcement
  - Homework 7 will be available tonight (due on 12/02)

- This lecture
  - Classification of parallel computers
  - Graphics processing
  - GPU architecture
  - CUDA programming model

Flynn’s Taxonomy

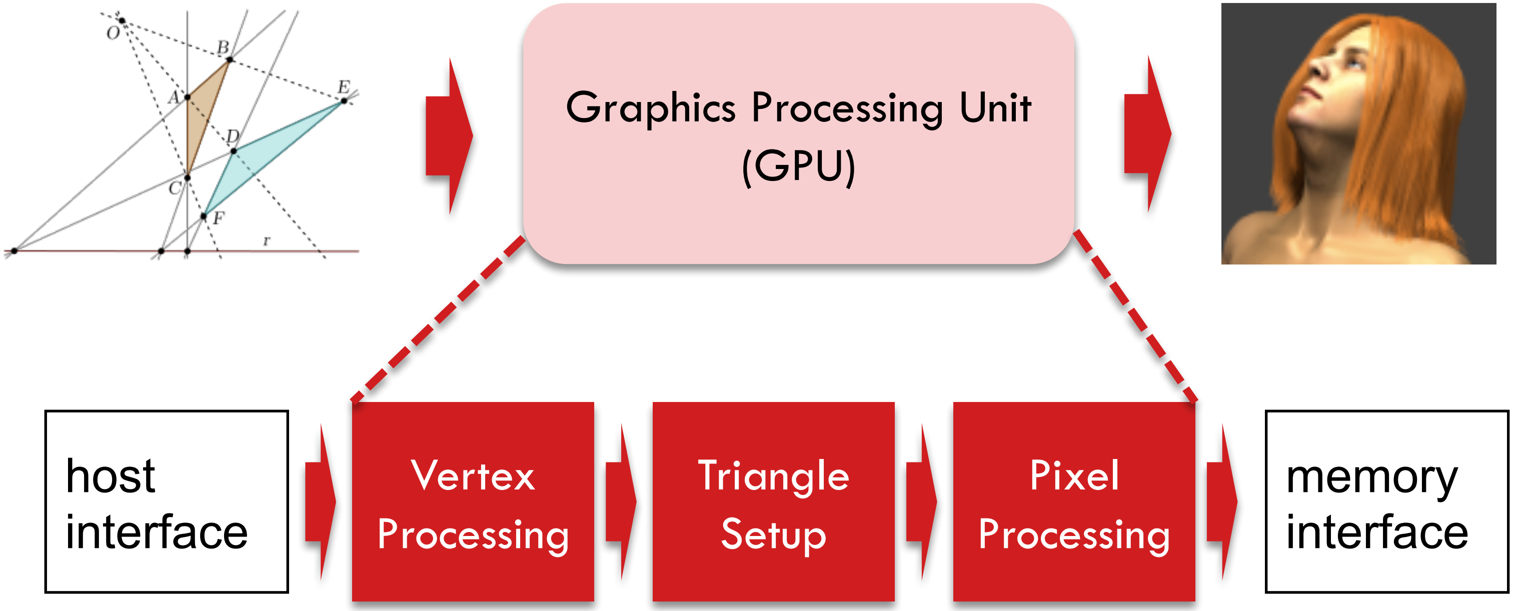
□ Data vs. instruction streams



		Instruction Stream	
		Single	Multiple
Data Stream	Single	Single-Instruction, Single Data (SISD) uniprocessors	Multiple-Instruction, Single Data (MISD) systolic arrays
	Multiple	Single-Instruction, Multiple Data (SIMD) vector processors	Multiple-Instruction, Multiple Data (MIMD) multicores

Graphics Processing Unit

- Initially developed as graphics accelerator
  - It receives geometry information from the CPU as an input and provides a picture as an output



Graphics Processing Unit (GPU)

host interface

Vertex Processing

Triangle Setup

Pixel Processing

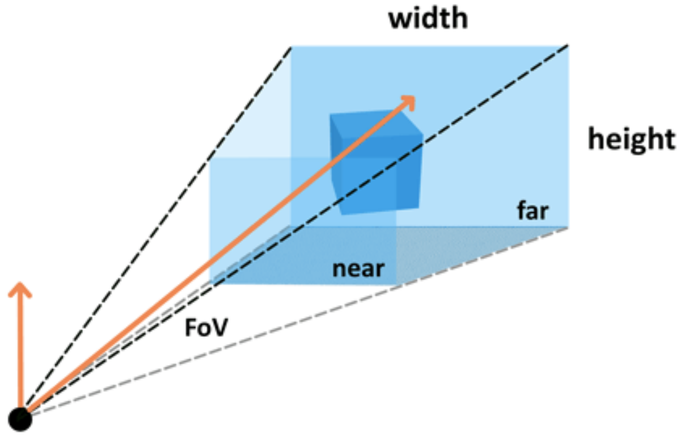
memory interface

Host Interface

- The host interface is the communication bridge between the CPU and the GPU
- It receives commands from the CPU and also pulls geometry information from system memory
- It outputs a *stream* of vertices in object space with all their associated information

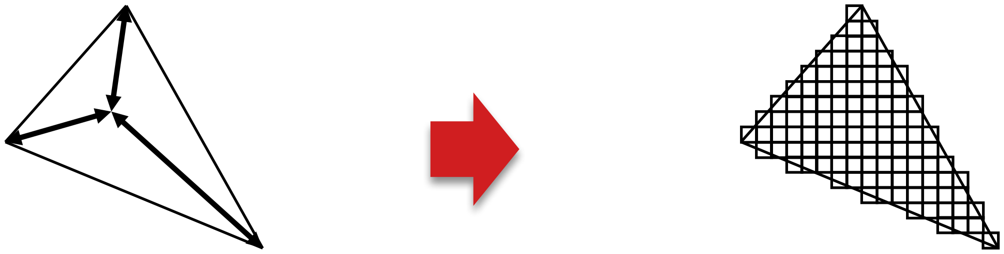
Vertex Processing

- The vertex processing stage receives vertices from the host interface in object space and outputs them in screen space
- This may be a simple linear transformation, or a complex operation involving morphing effects



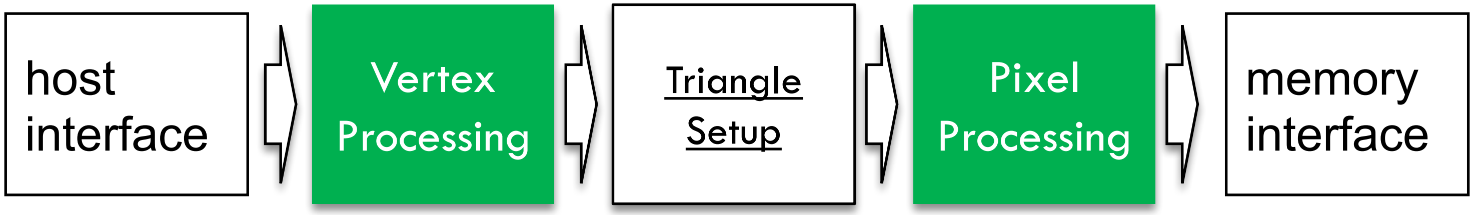
Pixel Processing

- Rasterize triangles to pixels
- Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, texcoord etc), which are used to compute the final color for this pixel
- The computations taking place here include texture mapping and math operations



Programming GPUs

- The programmer can write programs that are executed for every vertex as well as for every fragment
- This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications

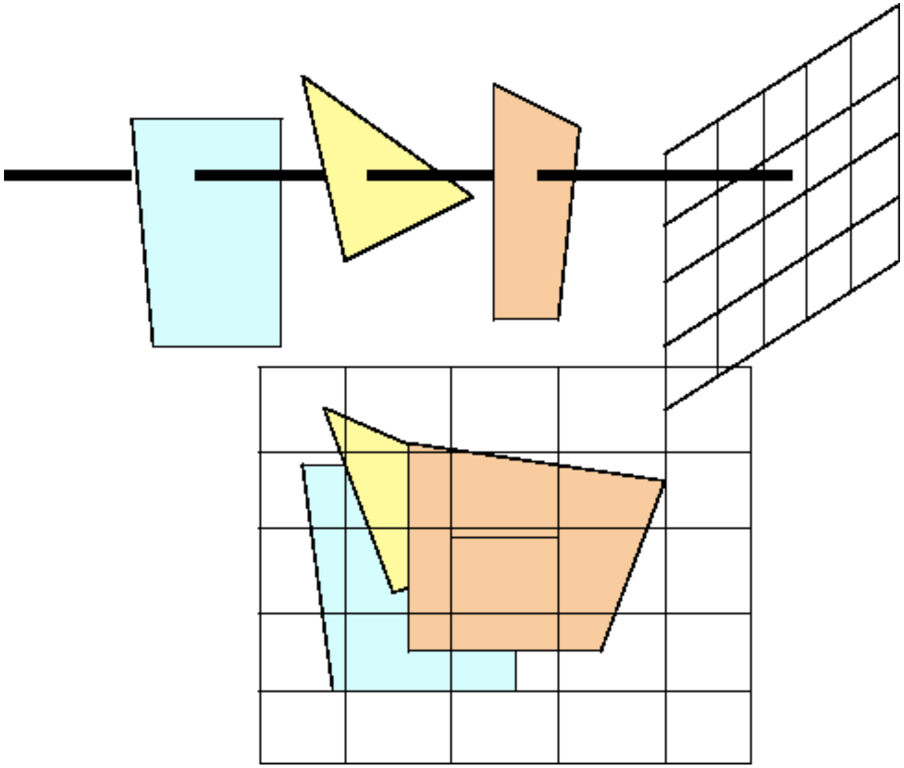


Memory Interface

- Fragment colors provided by the previous stage are written to the framebuffer
- Used to be the biggest bottleneck before fragment processing took over
- Before the final write occurs, some fragments are rejected by the zbuffer, stencil and alpha tests
- On modern GPUs, z and color are compressed to reduce framebuffer bandwidth (but not size)

Z-Buffer

- Example of 3 objects

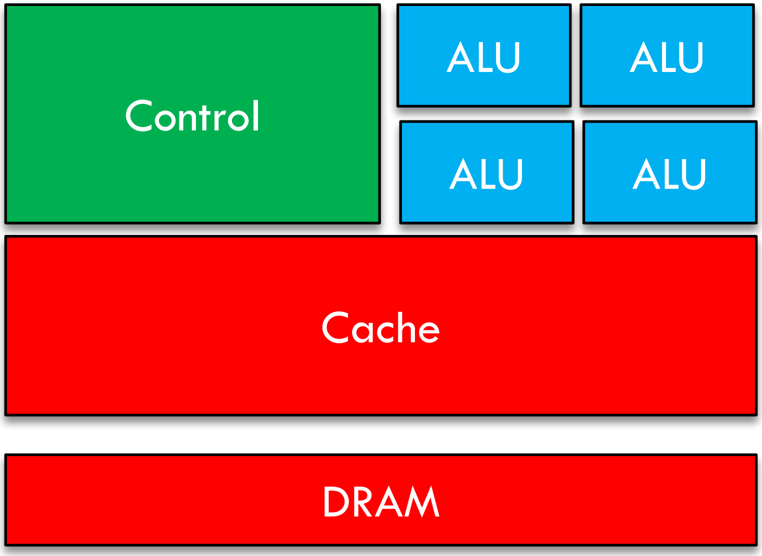


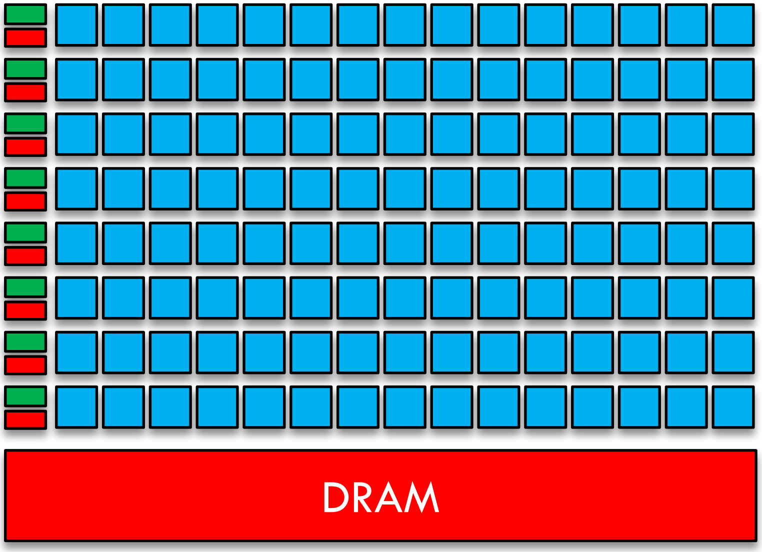
Graphics Processing Unit

- Initially developed as graphics accelerators
  - one of the densest compute engines available now
- Many efforts to run non-graphics workloads on GPUs
  - general-purpose GPUs (GPGPUs)
- C/C++ based programming platforms
  - CUDA from NVidia and OpenCL from an industry consortium
- A heterogeneous system
  - a regular host CPU
  - a GPU that handles CUDA (may be on the same CPU chip)

Graphics Processing Unit

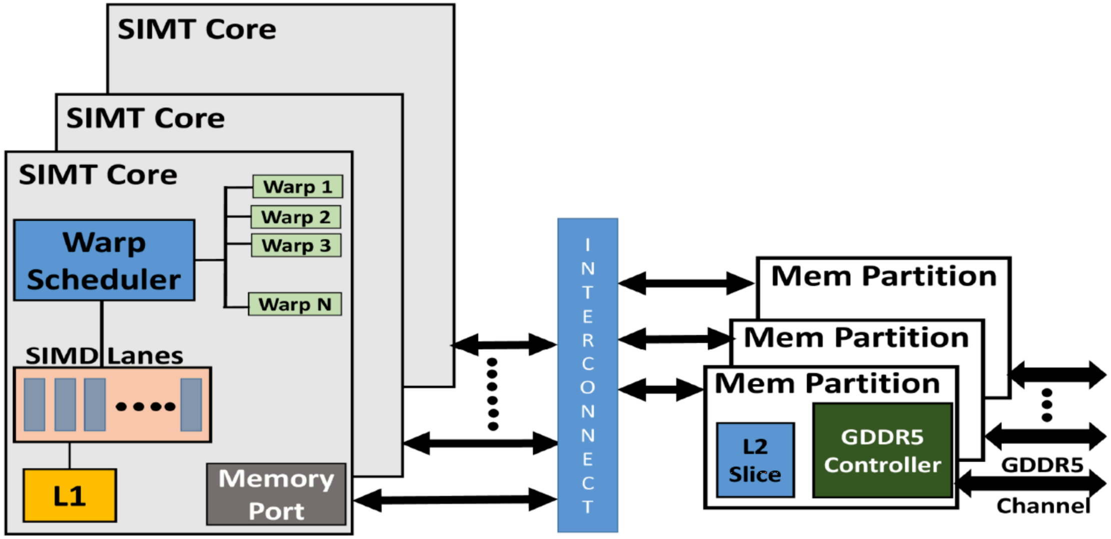
- Simple in-order pipelines that rely on thread-level parallelism to hide long latencies
- Many registers (~1K) per in-order pipeline (lane) to support many active warps



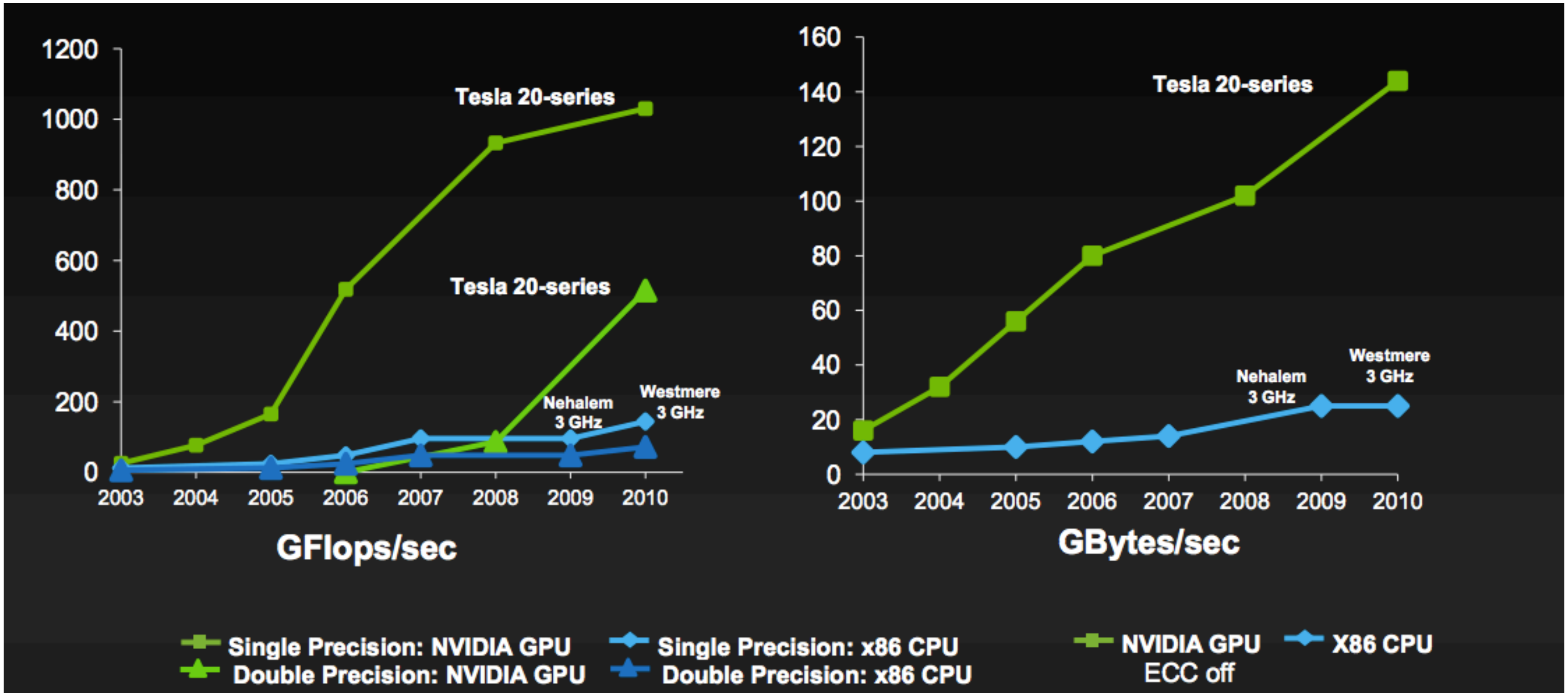


The GPU Architecture

- SIMT – single instruction, multiple threads
  - GPU has many SIMT cores
- Application → many thread blocks (1 per SIMT core)
- Thread block → many warps (1 warp per SIMT core)
- Warp → many in-order pipelines (SIMD lanes)



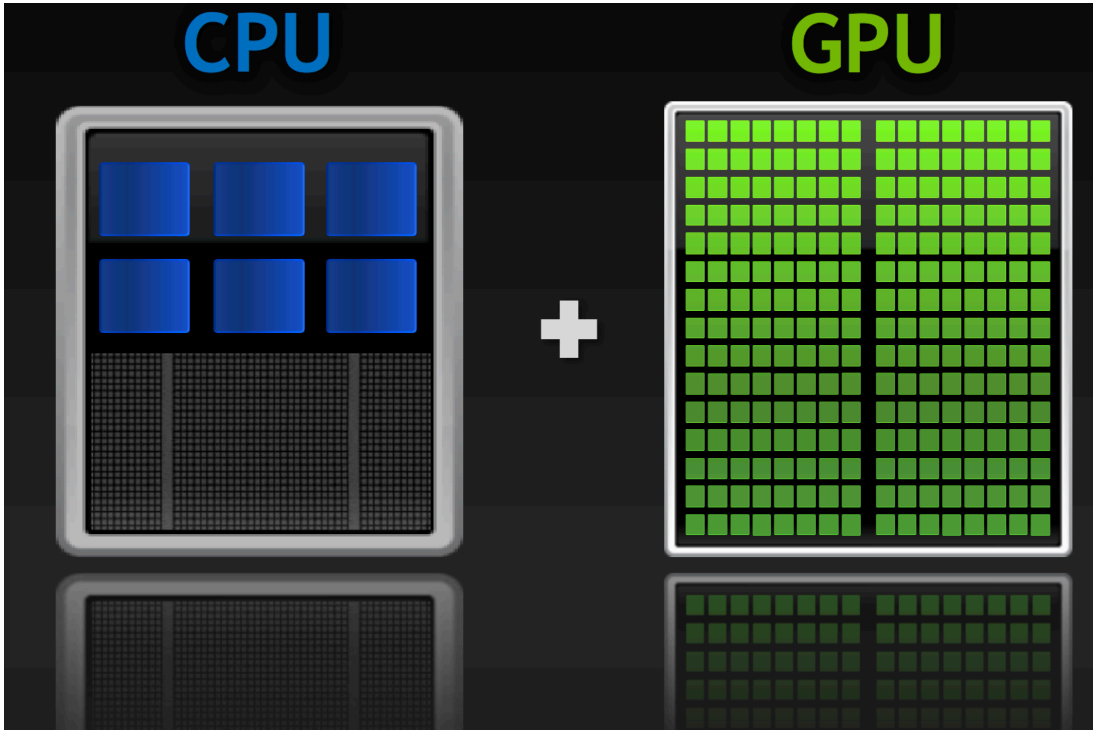
Why GPU Computing?



Source: NVIDIA

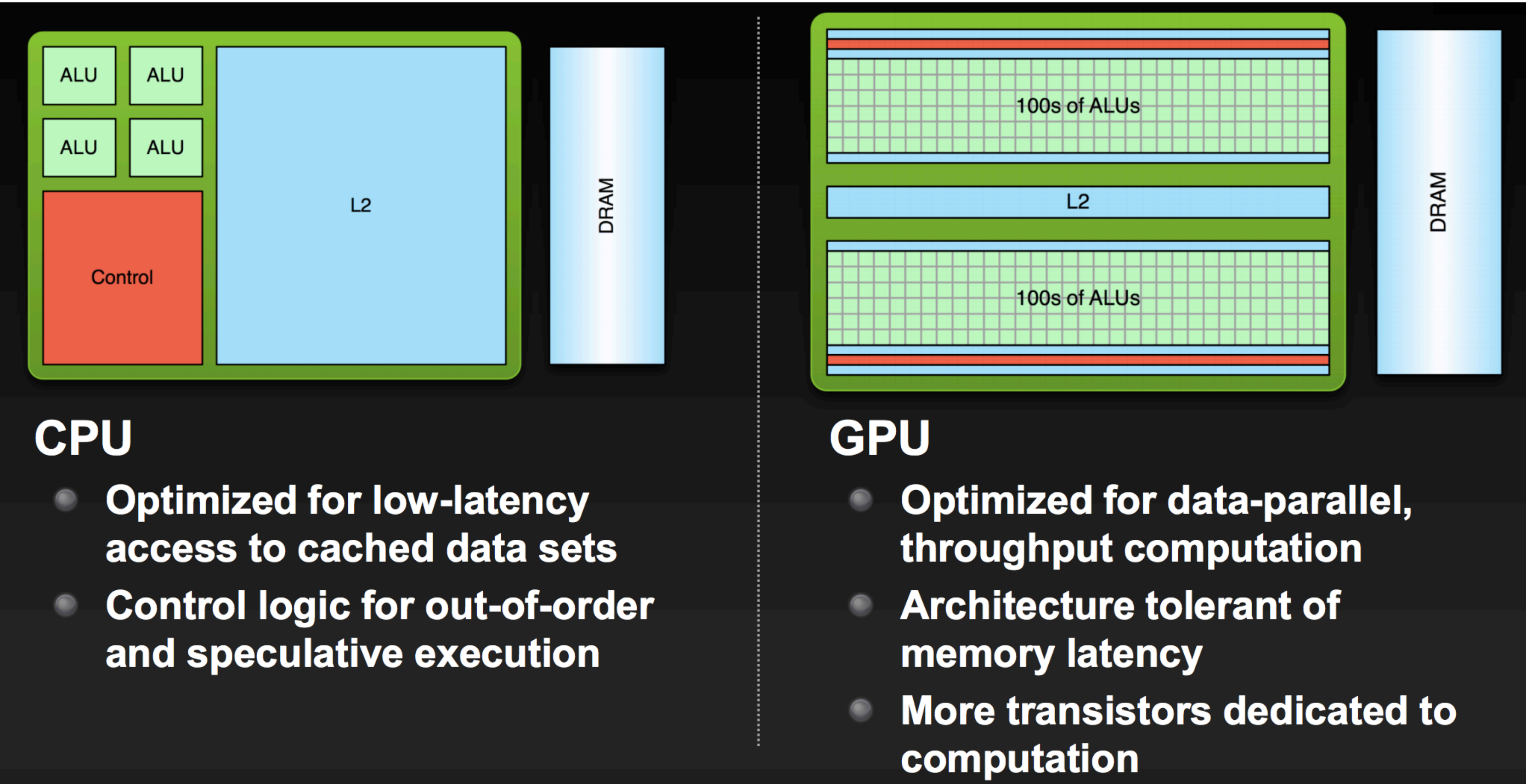
GPU Computing

- GPU as an accelerator in scientific applications



GPU Computing

□ Low latency or high throughput?



CPU

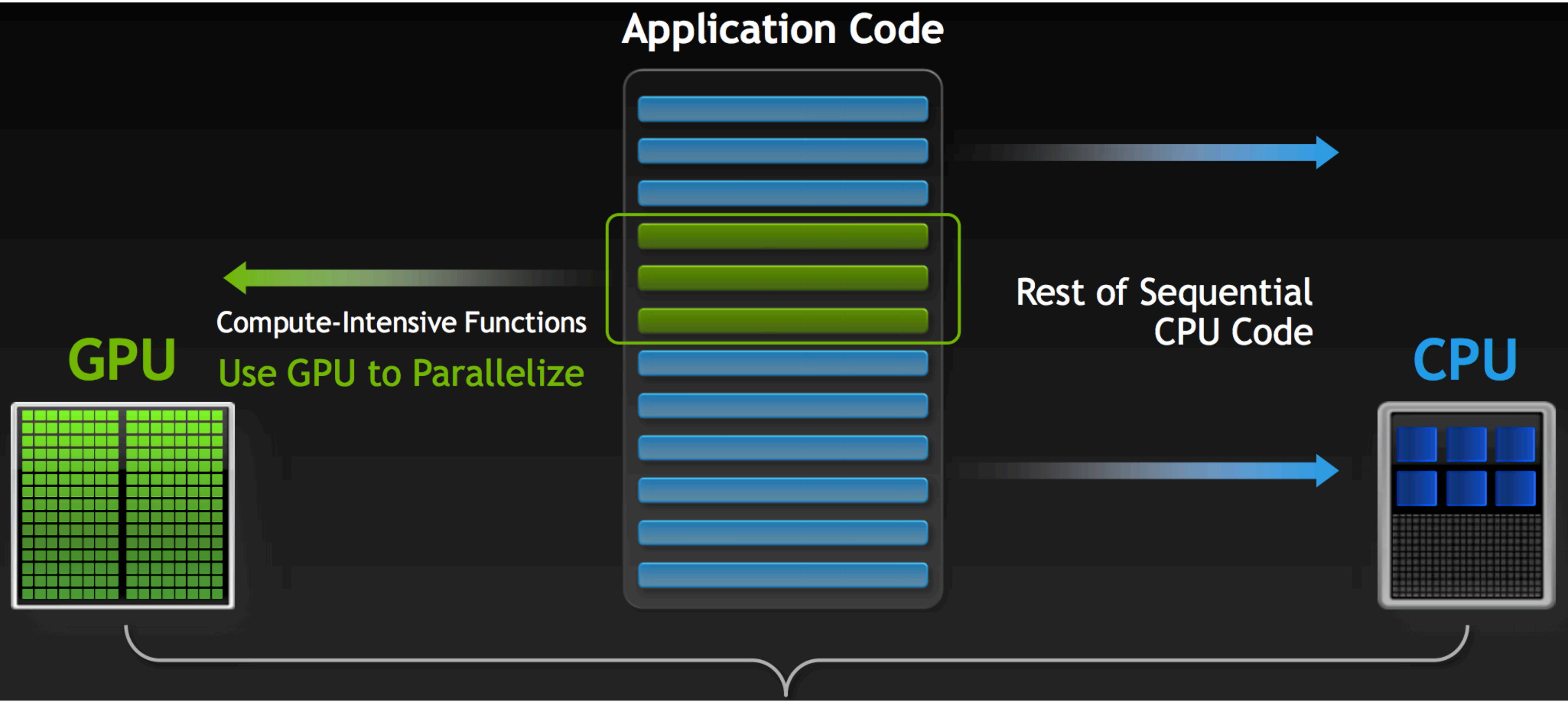
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

GPU Computing

- Low latency or high throughput



CUDA Programming Model

- Step 1: substitute library calls with equivalent CUDA library calls
  - saxpy (…) → cublasSaxpy (…)
    - single precision alpha x plus y ($z = \alpha x + y$)
- Step 2: manage data locality
  - cudaMalloc(), cudaMemcpy(), etc.
- Step 3: transfer data between CPU and GPU
  - get and set functions
- rebuild and link the CUDA-accelerated library
  - nvcc myobj.o –l cublas

Example: SAXPY Code

```
int N = 1 << 20;


// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);
```

Example: CUDA Lib Calls

```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

Example: Initialize CUDA Lib

```
int N = 1 << 20;

cublasInit();



// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);



cublasShutdown();
```

Example: Allocate Memory

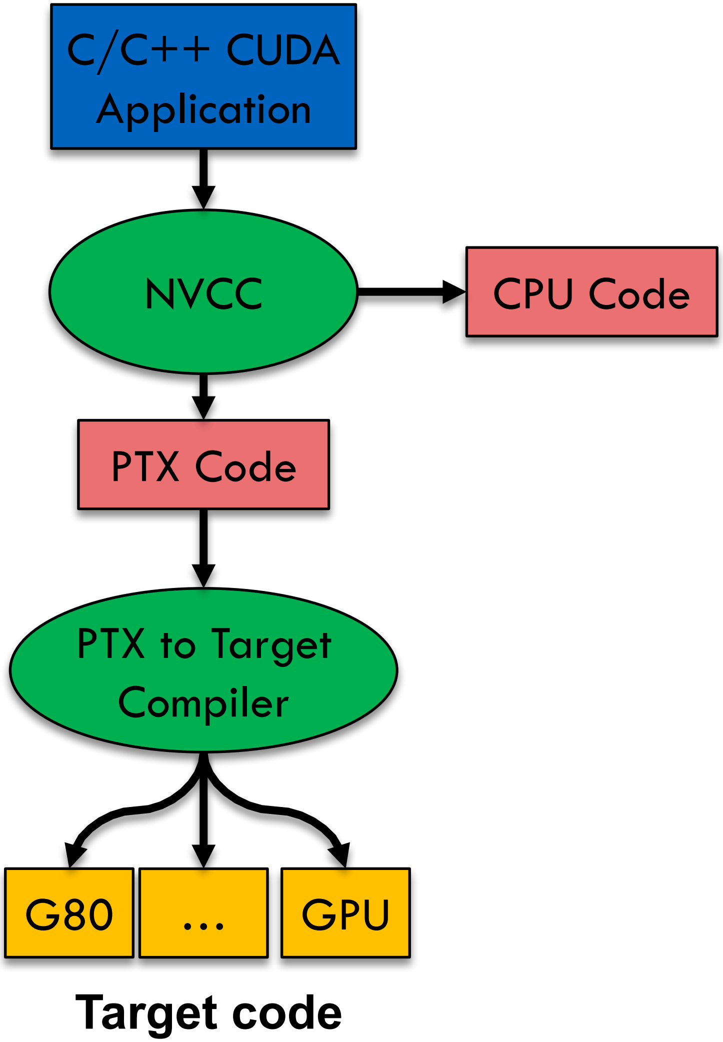
```
int N = 1 << 20;

cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);




// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);



cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Example: Transfer Data

```
int N = 1 << 20;

cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

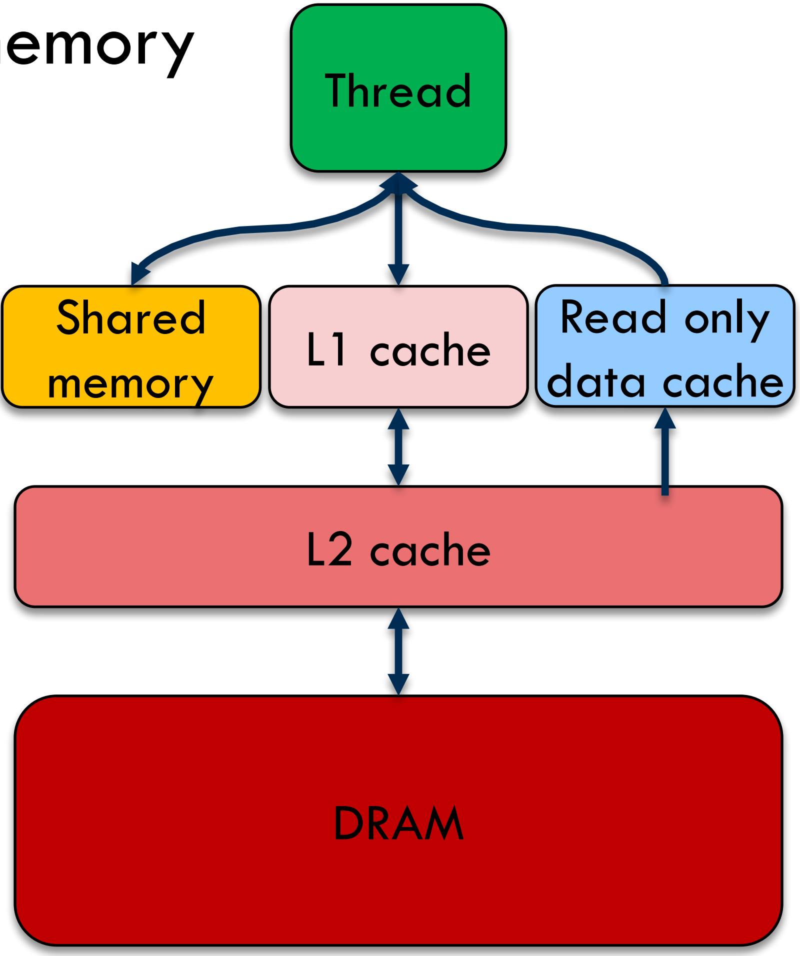
Compiling CUDA

- Call nvcc
- Parallel Threads eXecution (PTX)
  - Virtual machine and ISA
- Two stage
  - 1. PTX
  - 2. device-specific binary object



Memory Hierarchy

- Throughput-oriented main memory
  - Graphics DDR (GDDR)
    - Wide channels: 256 bit
    - Lower clock rate than DDR
  - 1.5MB shared L2
  - 48KB read-only data cache
    - Compiler controlled
  - Wide buses


Thread
Shared memory
L1 cache
Read only data cache
L2 cache
DRAM