
ACACES 2018 Summer School

GPU Architectures: Basic to Advanced Concepts

Adwait Jog, Assistant Professor

College of William & Mary
(<http://adwaitjog.github.io/>)

William & Mary

- Second oldest-institution of higher education in the USA
- Located in Williamsburg, VA, USA. *Recently hosted ASPLOS conference – one of the top venues for computer architecture research.*
- I am affiliated with Computer Science Department
 - Graduate Program (~65-70 Ph.D. students)
 - 25 Faculty Members
- Many graduated Ph.D. students have successfully established careers in academia & industry.



Brief Introduction



Adwait Jog
(Assistant Professor)

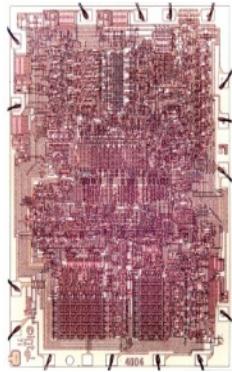
Interested in developing high-performance, energy-efficient and scalable systems that are low cost, reliable, and secure.

Special focus on GPU architectures and accelerators.

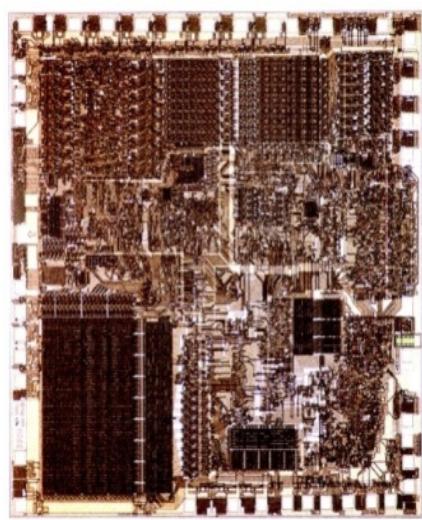
I lead Insight Computer Architecture
Lab at College of William and Mary
(<http://insight-archlab.github.io/>)

Our lab is funded by
US National Science Foundation (NSF) and
always looking to hire bright students at all levels.

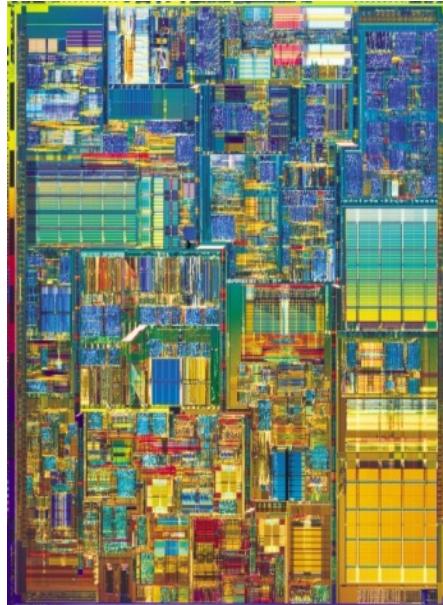
Journey of CMPs: Scaling and Heterogeneity Trends



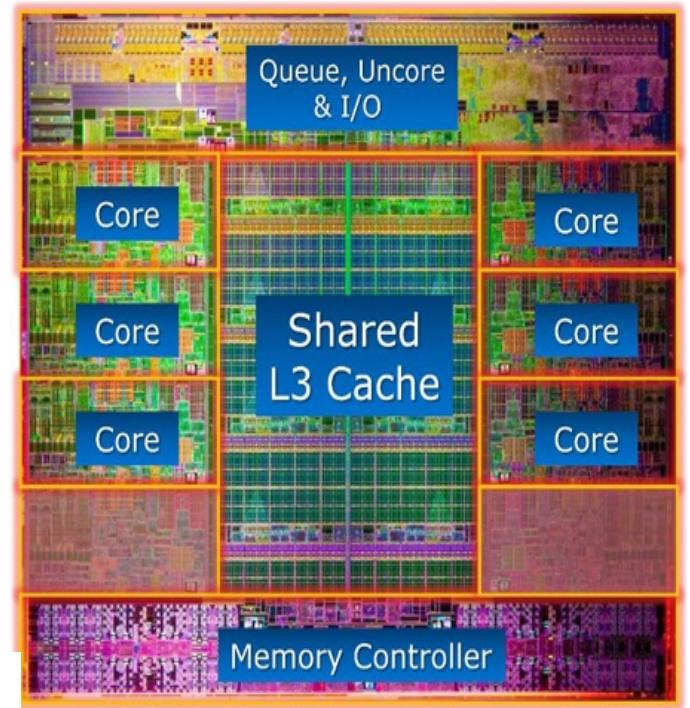
Intel 4004,
1971
1 core,
no cache
23K transistors



Intel 8088,
1978
1 core,
no cache
29K transistors



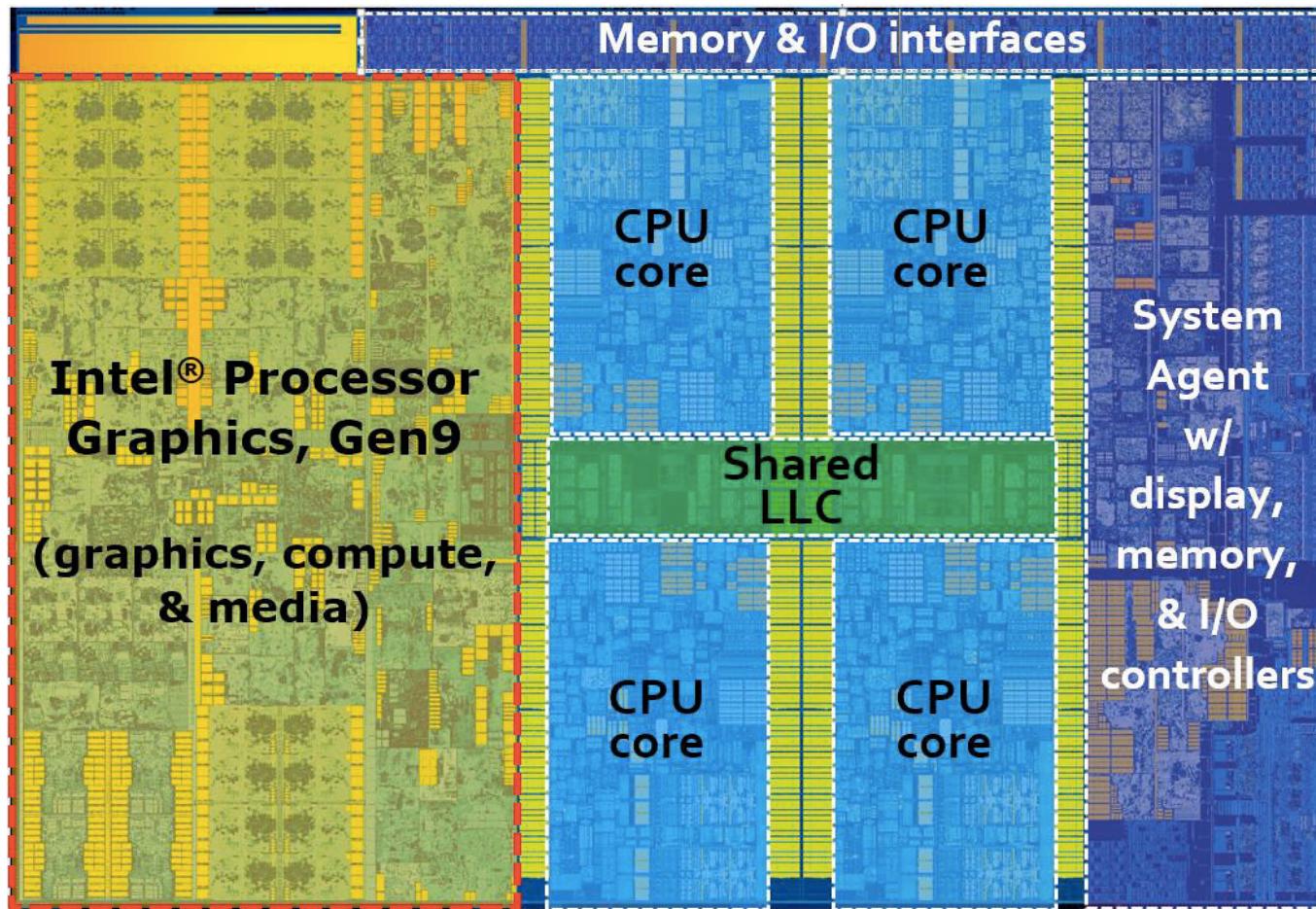
Intel Pentium 4,
2000
1 core
256 KB L2 cache
42M transistors



Intel Sandy Bridge,
2011
6 cores
15 MB L3 cache
2270M transistors

What's now?

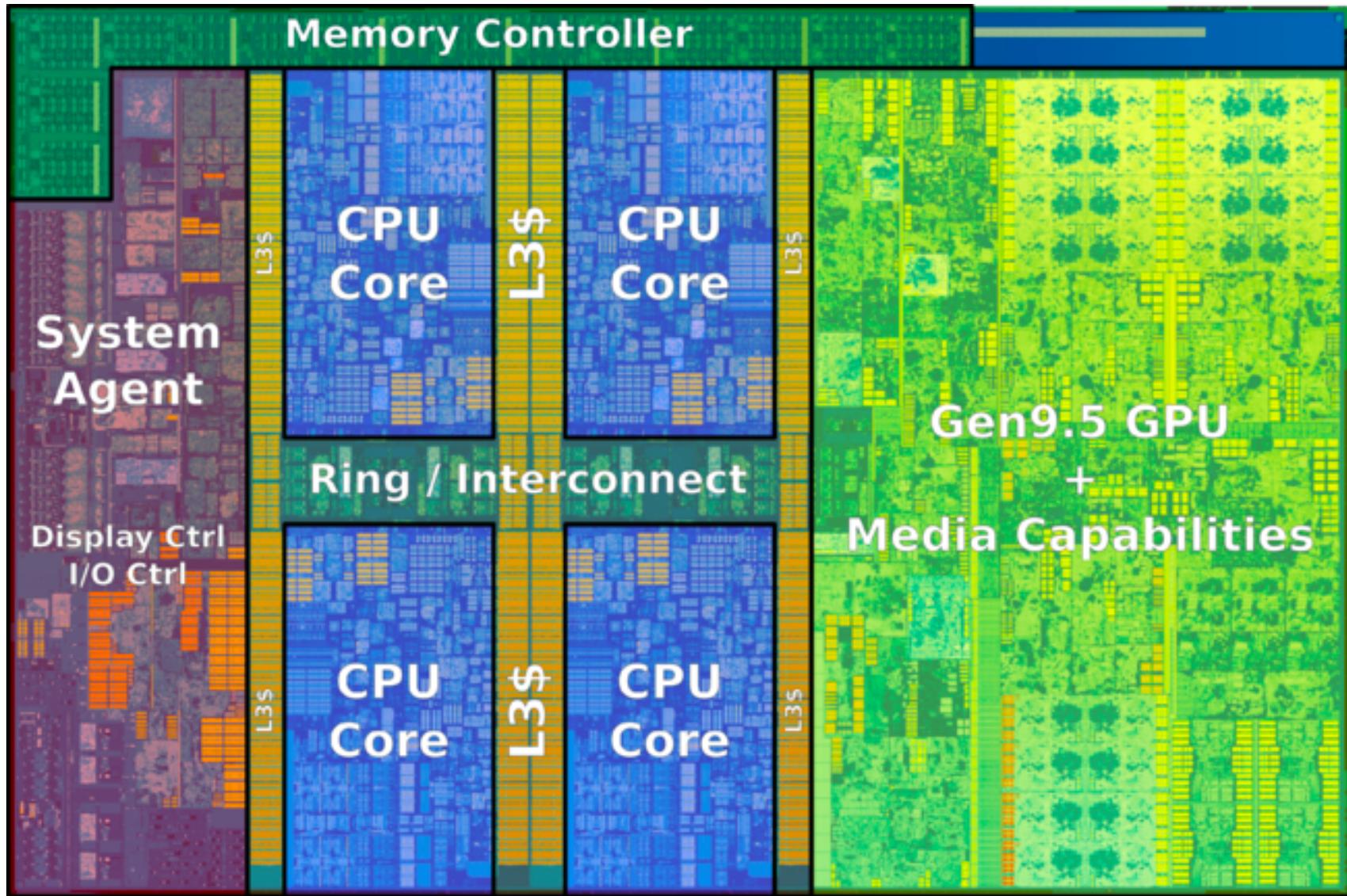
Intel Core i7-6700K Processor, 2016 (SKYLAKE)



1.7 billion transistors, 14 nm process, die size 122 mm²

“Skylake”

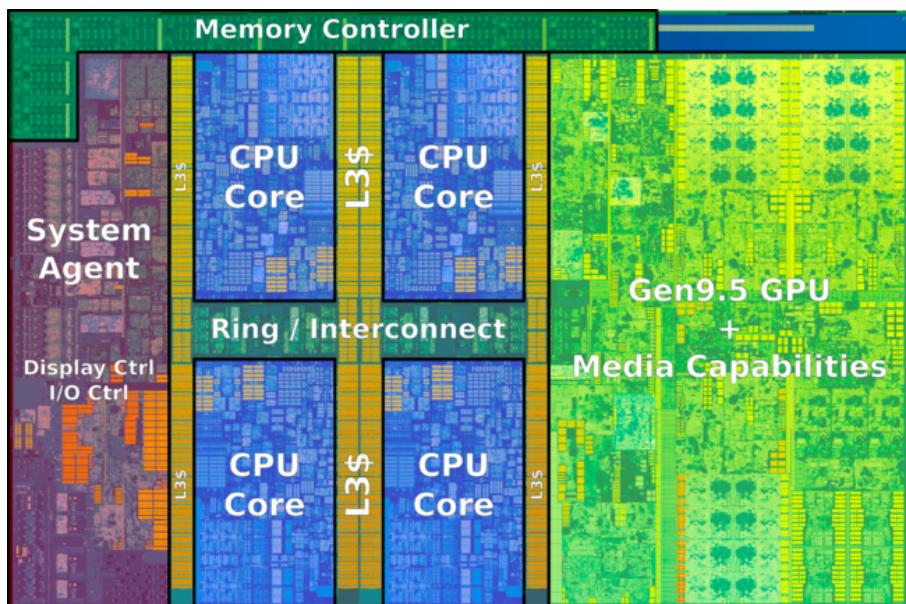
Intel Quad Core GT2, 2017 (Kaby Lake)



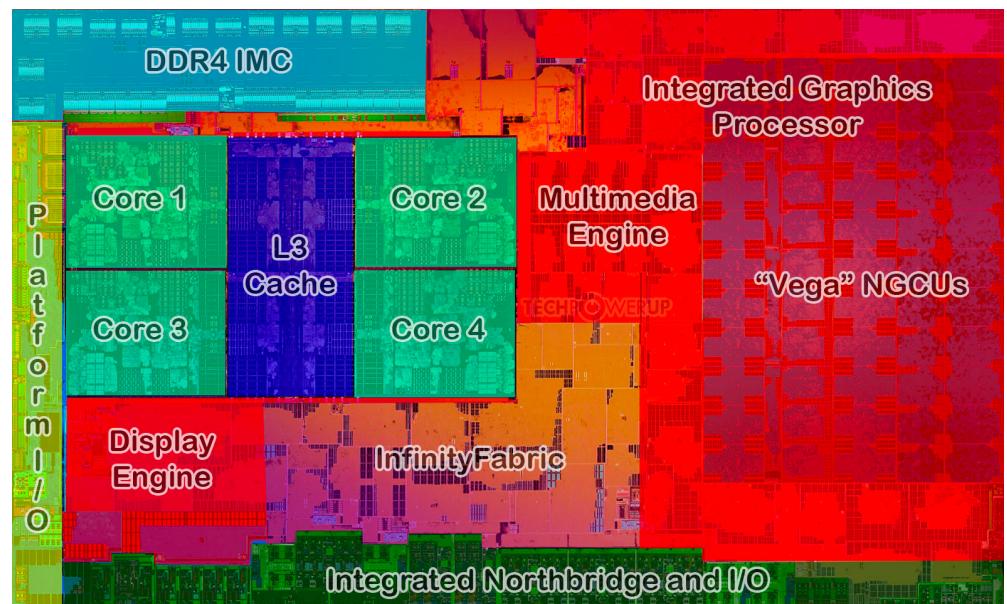
14 nm process, die size 126 mm²

Heterogeneous Architectures (2018)

Intel Coffee Lake and Kaby Lake



AMD Raven Ridge



Discrete GPU Cards --- Scaling Trends

2008



2010



2012



2014



2016



2018



GTX 275
(Tesla)
240

GTX 480
(Fermi)
448

GTX 680
(Kepler)
1536

GTX 980
(Maxwell)
2048

GP 100
(Pascal)
3584

GV 100
(Volta)
5120

CUDA
Cores
(127
GB/sec)

CUDA
Cores
(139
GB/sec)

CUDA
Cores
(192
GB/sec)

CUDA
Cores
(224
GB/sec)

CUDA
Cores
(720
GB/sec)

CUDA
Cores
(900
GB/sec)

GPUs are Ubiquitous (1)



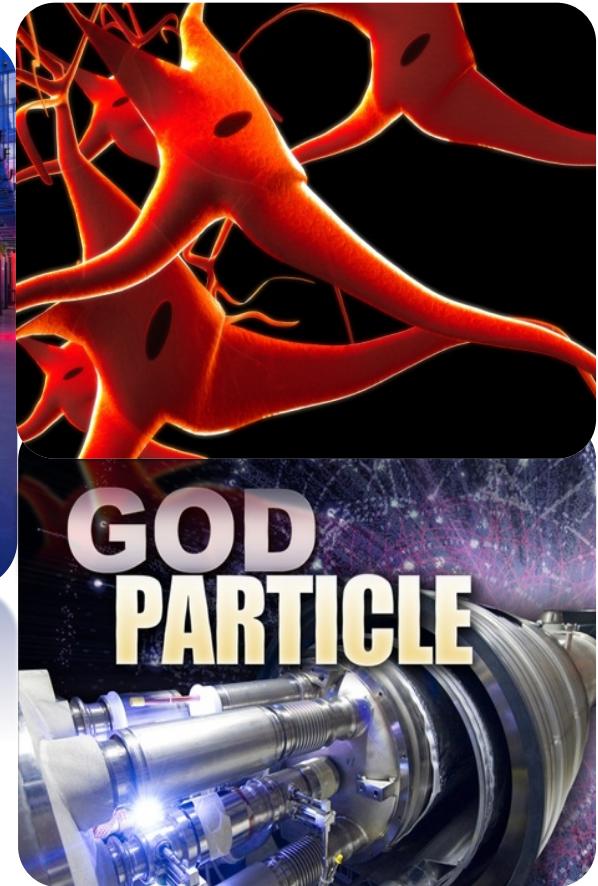
GPUs are Ubiquitous (2)



**HELPS
UNDERSTAND
UNIVERSE**



**BACKEND OF
“GOOGLE”
&
MANY OTHERS**

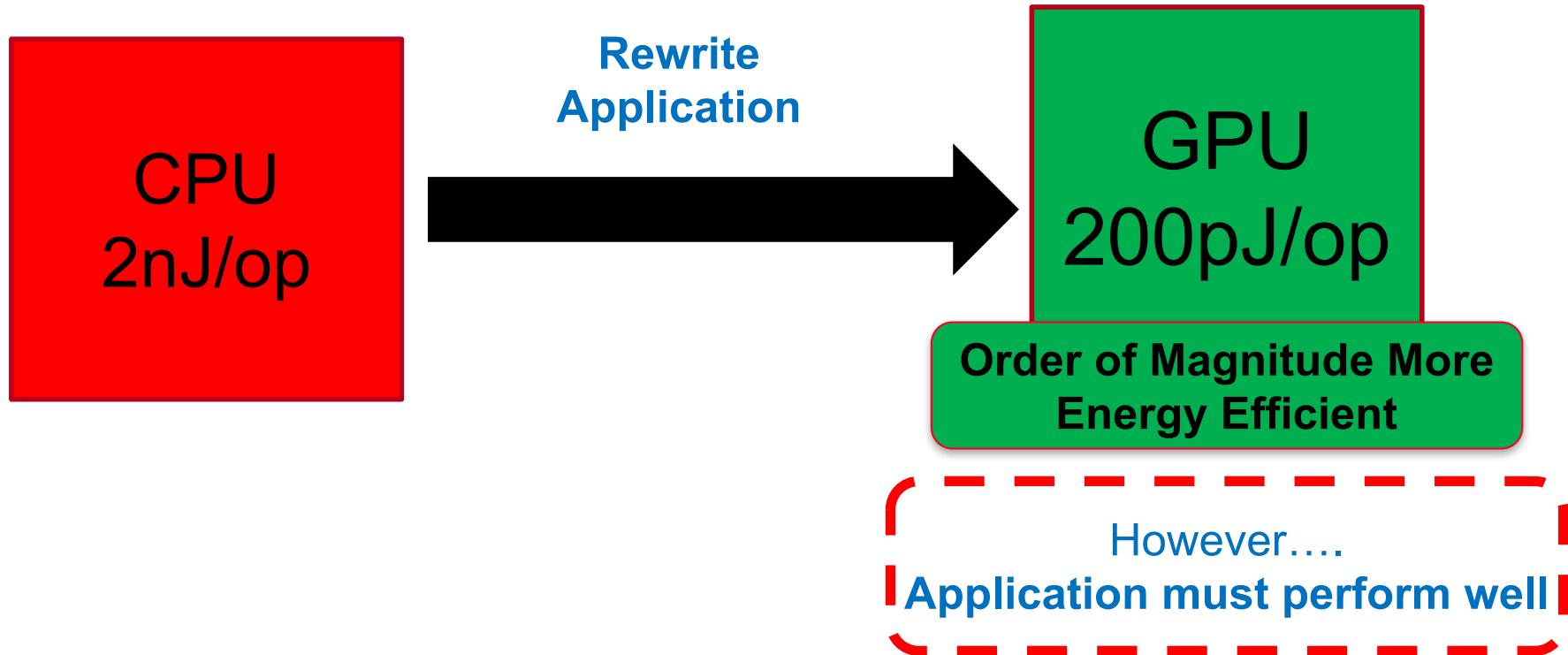


**GOD
PARTICLE**

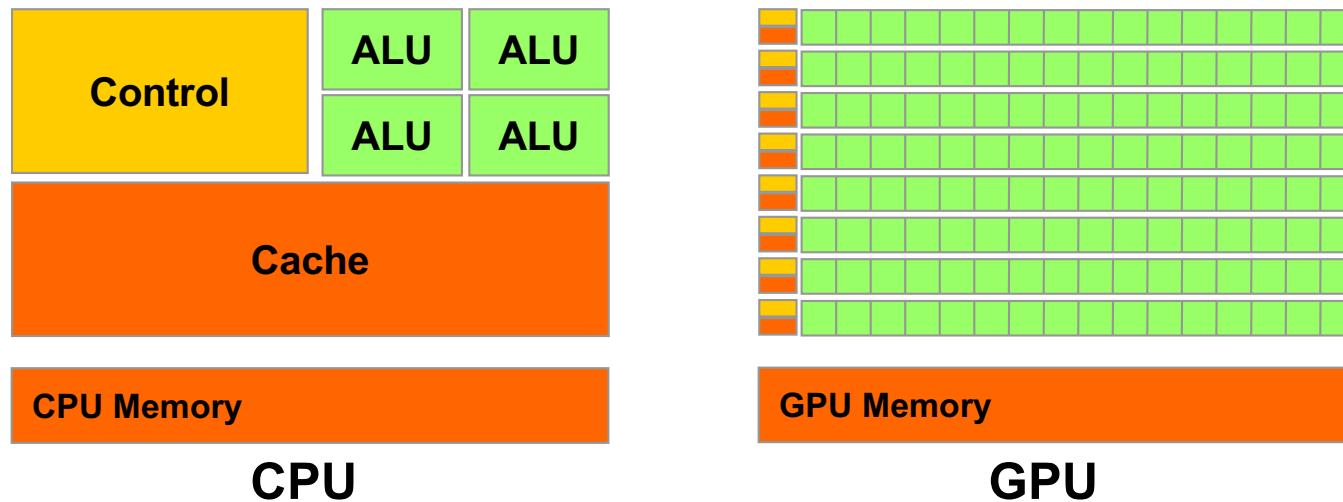
**ENABLING
SCIENTIFIC
COMPUTING
&
BREAKTHROUGHS**

Why use a GPU for computing?

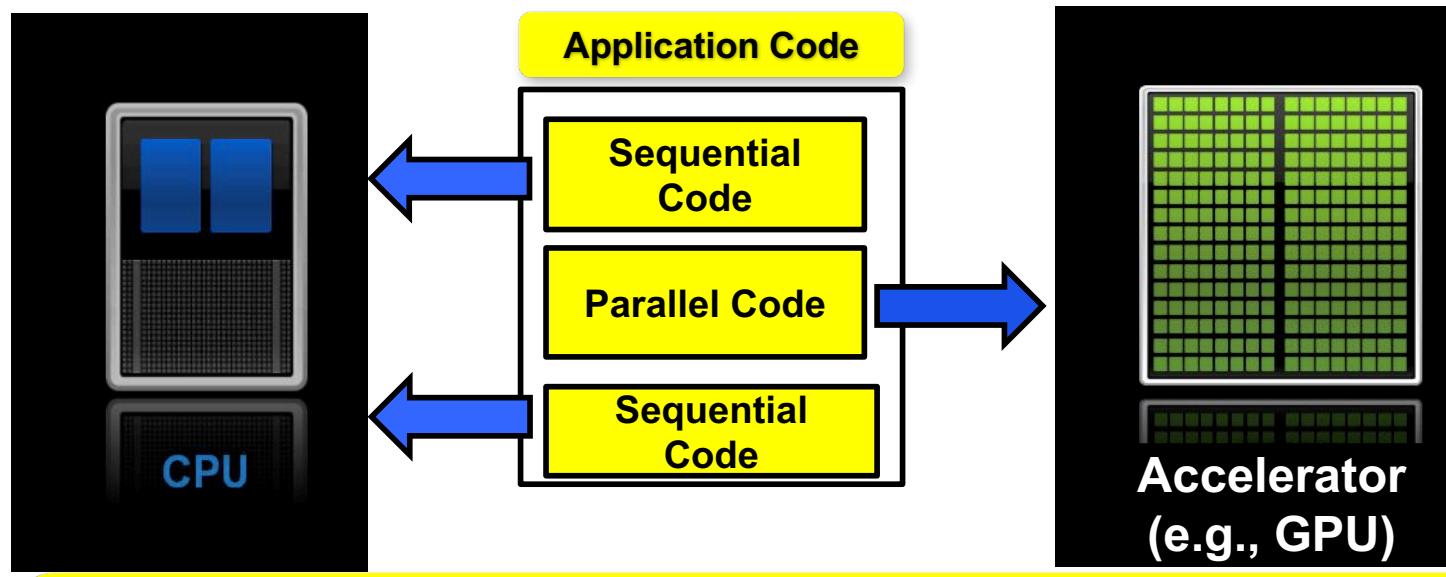
- ❑ GPU uses larger fraction of silicon for computation than CPU.
- ❑ At peak performance GPU uses order of magnitude less energy per operation than CPU.



GPU vs. CPU

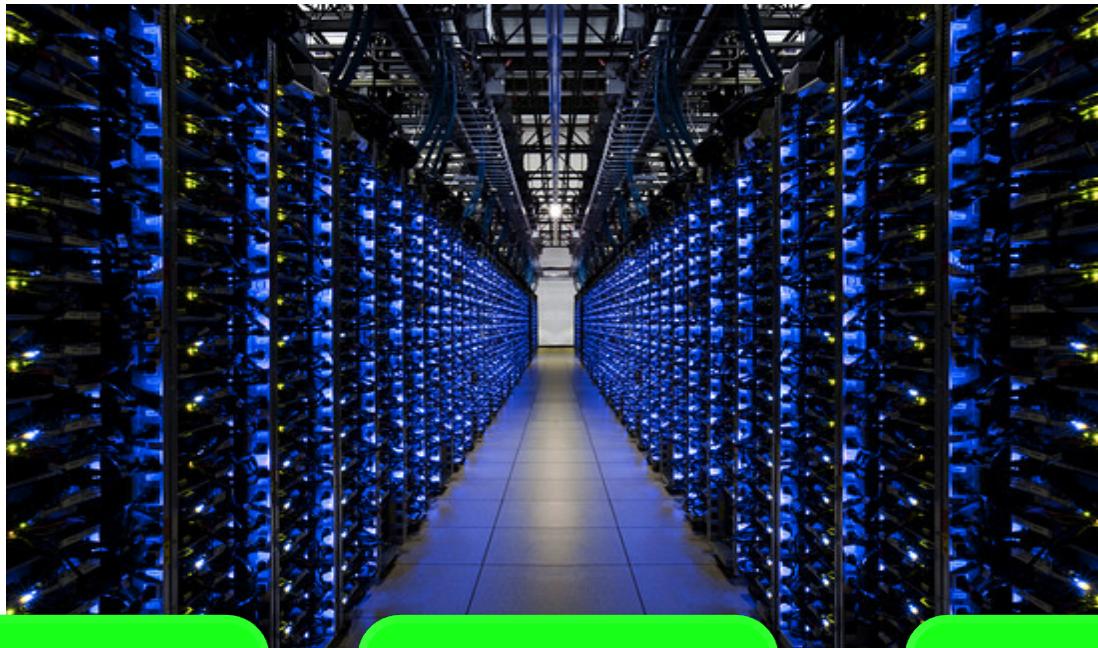


How Acceleration Works



**Many Top 20 supercomputers
in the green500 list employ accelerators.**

Fastest Super Computer* -- SUMMIT @ Oak Ridge



**Multiple
Volta GPUs**

NVLink

**HBM +
DDR4**

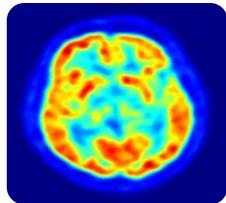
<https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/>

* As of June 2018

Making Good Use of Resources (1)



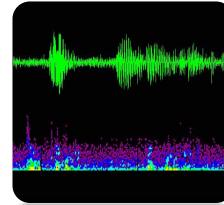
Astronomy



Medical Imaging



Physics Simulation



Audio Processing



Machine Learning

Data-Level Parallelism

Large Data Sets



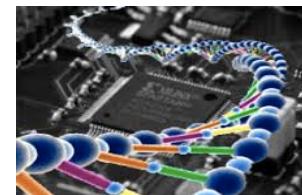
Games



Image Processing



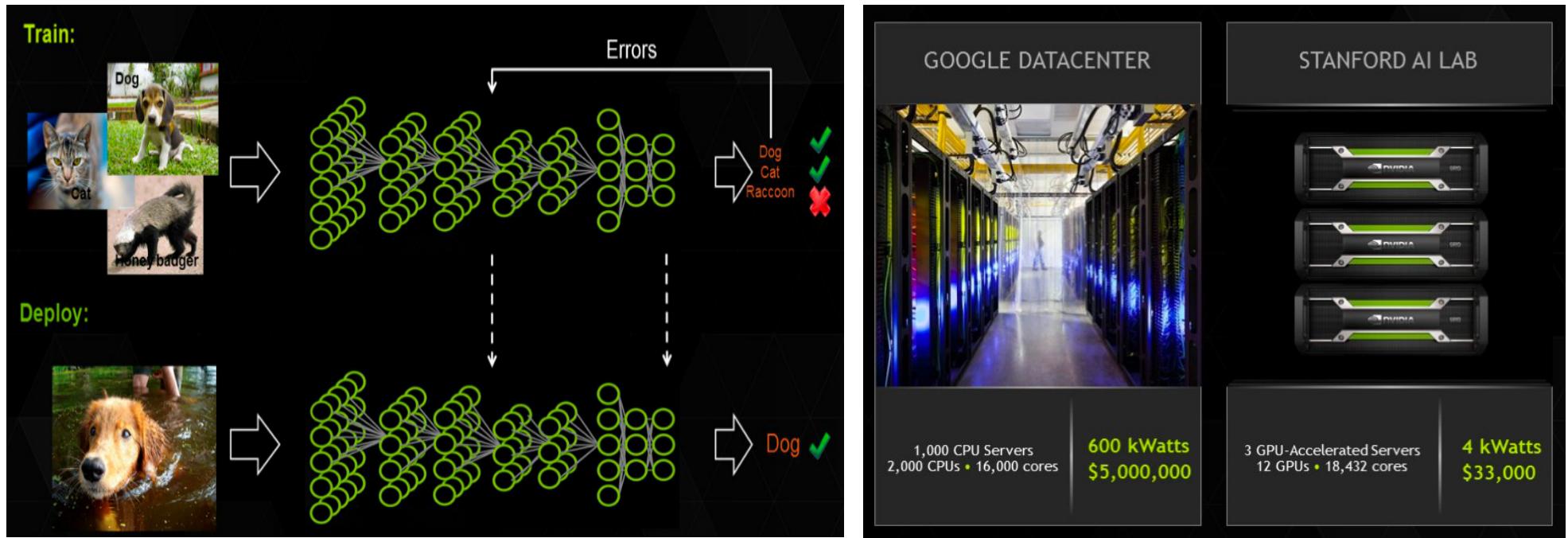
Financial Computing



Genomics

Making Good Use of Resources (2)

❑ Deep Learning and Artificial Intelligence



Credit: NVIDIA AI

❑ However, there are several compute and memory bottlenecks in GPU-based systems that need to be addressed via software- and/or hardware-based solutions

Course Outline

❑ Lectures 1 and 2: Basics Concepts

- Basics of GPU Programming
- Basics of GPU Architecture

❑ Lecture 3: GPU Performance Bottlenecks

- Memory Bottlenecks
- Compute Bottlenecks
- Possible Software and Hardware Solutions

❑ Lecture 4: GPU Security Concerns

- Timing channels
- Possible Software and Hardware Solutions

Course Objectives

- ❑ By the end of this (short) course, I hope you can appreciate
 - the benefits of GPUs
 - the architectural differences between CPU and GPU
 - the key research challenges in the context of GPUs
 - some of the existing research directions
- ❑ I encourage questions during and after the class
 - Ample time for discussions during the week

Background

- My assumption is that students have some background on basic computer organization and design.
- Question 1: How many of you have taken undergraduate-level course on computer architecture?
- Question 2: How many of you have taken graduate-level course on computer architecture?
- Question 3: How many of you have taken a GPU course before?

Reading Material (Books & Docs)

- ❑ D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach, 3rd Edition”
- ❑ Patterson and Hennesy, *Computer Organization and Design*, 5th Edition, Appendix C-2 on GPUs
- ❑ Aamodt, Fung, Rogers, “General-Purpose Graphics Processing Architectures” – Morgan & Claypool Publishers, 1st Edition (New book!)
- ❑ Nvidia CUDA C Programming Guide
 - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

Course Outline

□ Lectures 1 and 2: Basics Concepts

- **Basics of GPU Programming**
- Basics of GPU Architecture

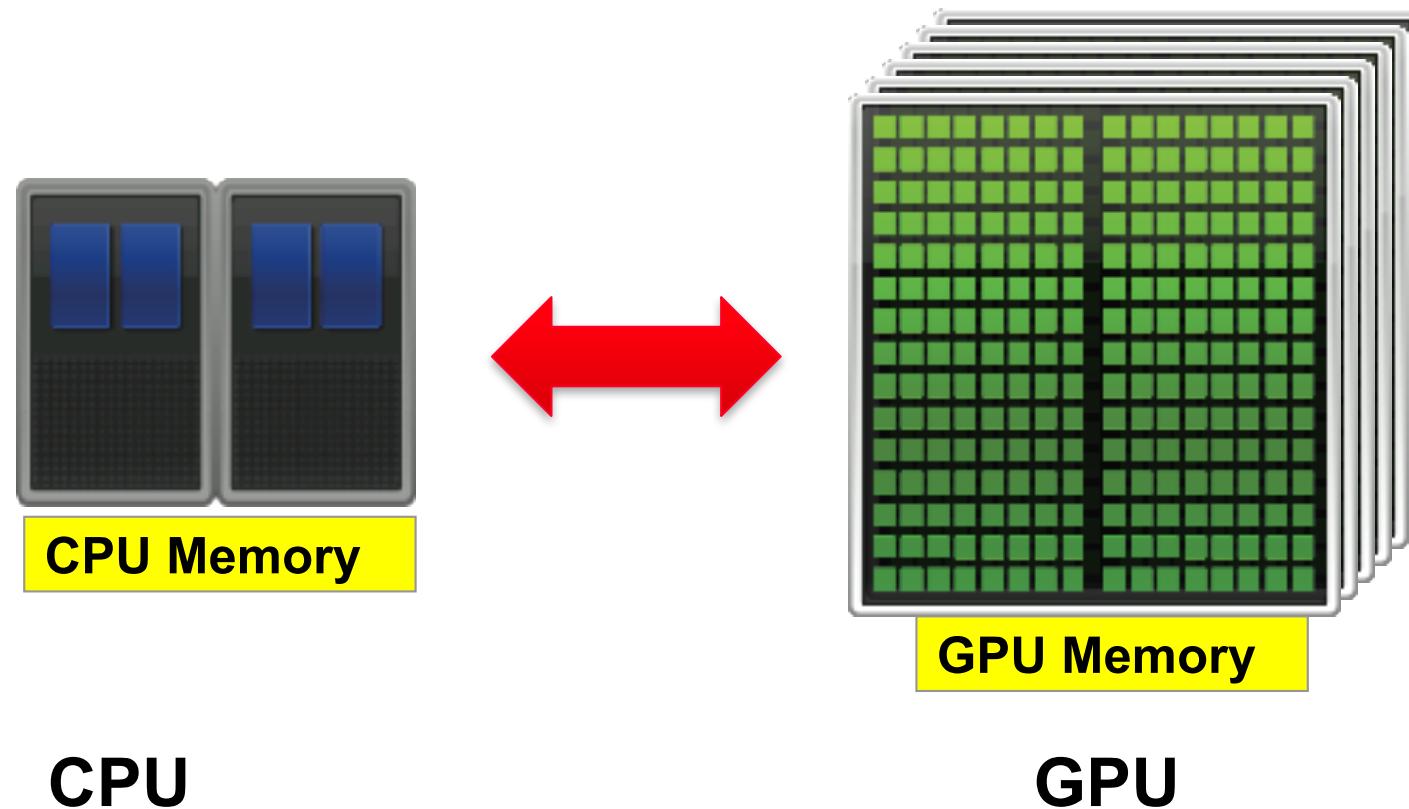
□ Lecture 3: GPU Performance Bottlenecks

- Memory Bottlenecks
- Compute Bottlenecks
- Possible Software and Hardware Solutions

□ Lecture 4: GPU Security Concerns

- Timing channels
- Possible Software and Hardware Solutions

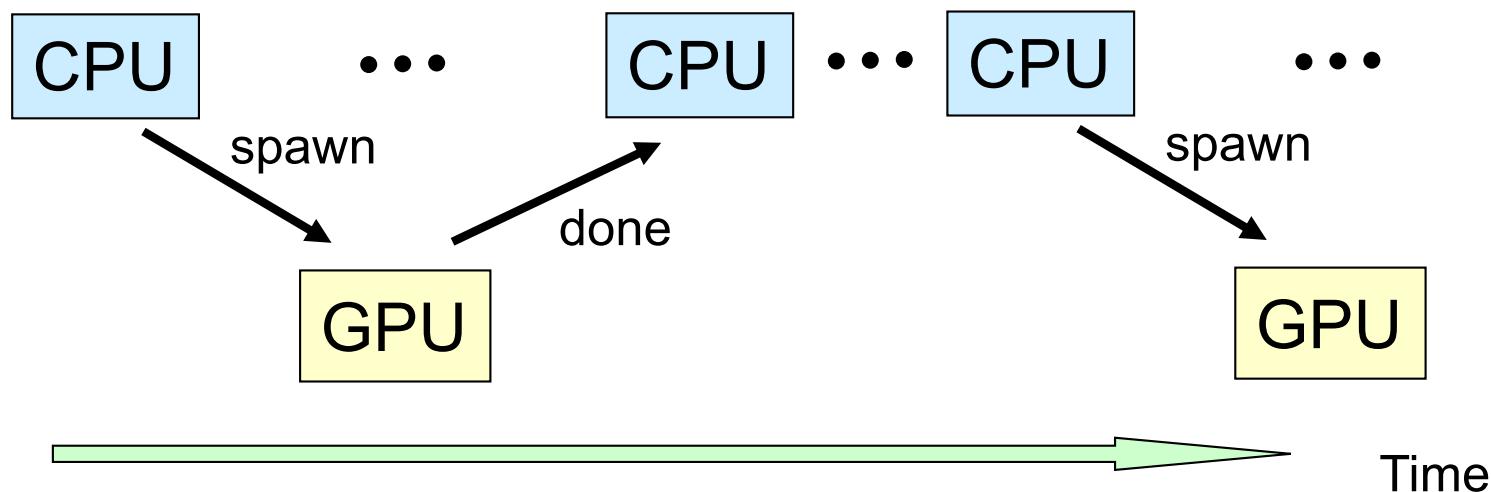
How is this system programmed (today)?



GPU Programming Model

+

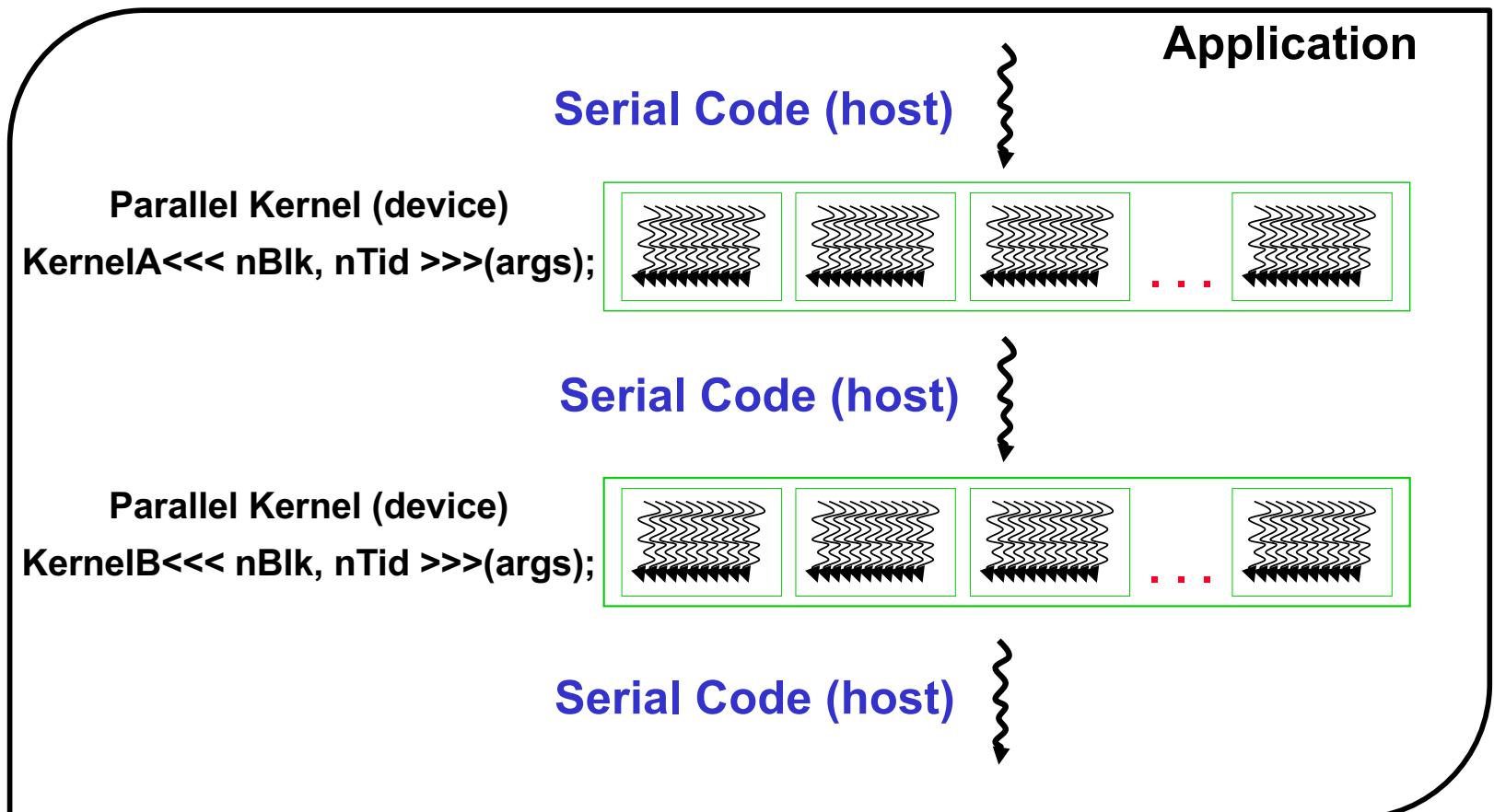
- CPU (host) “off-load” parallel kernels to GPU (device)



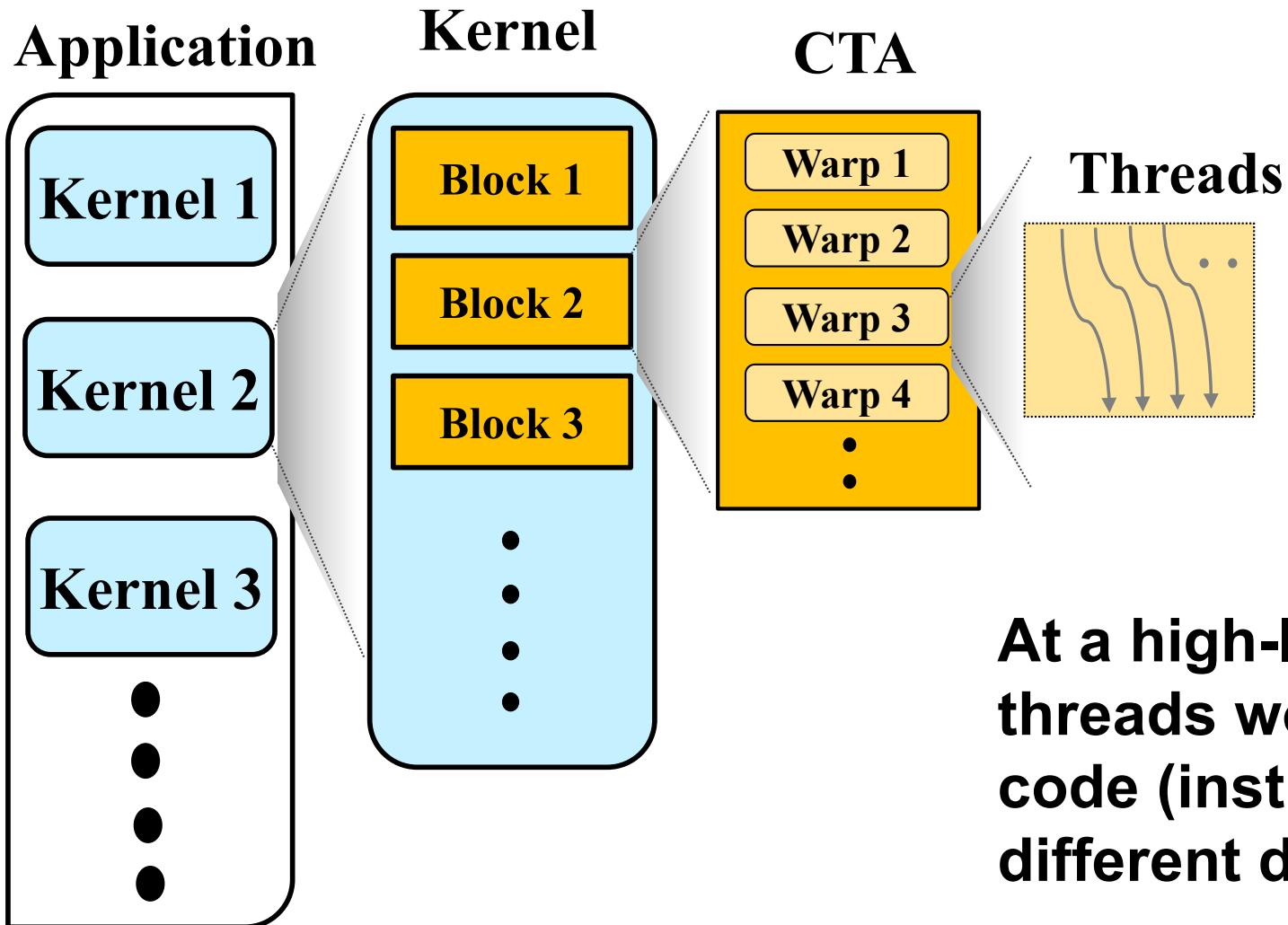
- Transfer data to GPU memory
- GPU spawns threads
- Need to transfer result data back to CPU main memory

CUDA Execution Model

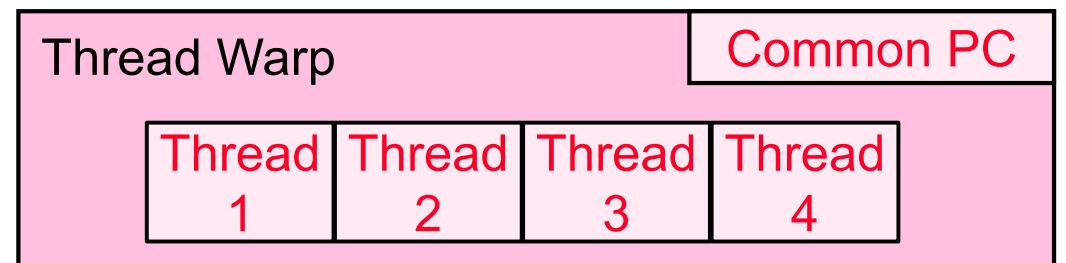
- Application Code
 - Serial parts (C code) in CPU (host)
 - Parallel parts (Kernel code) in GPU (device)



GPU as SIMD machine

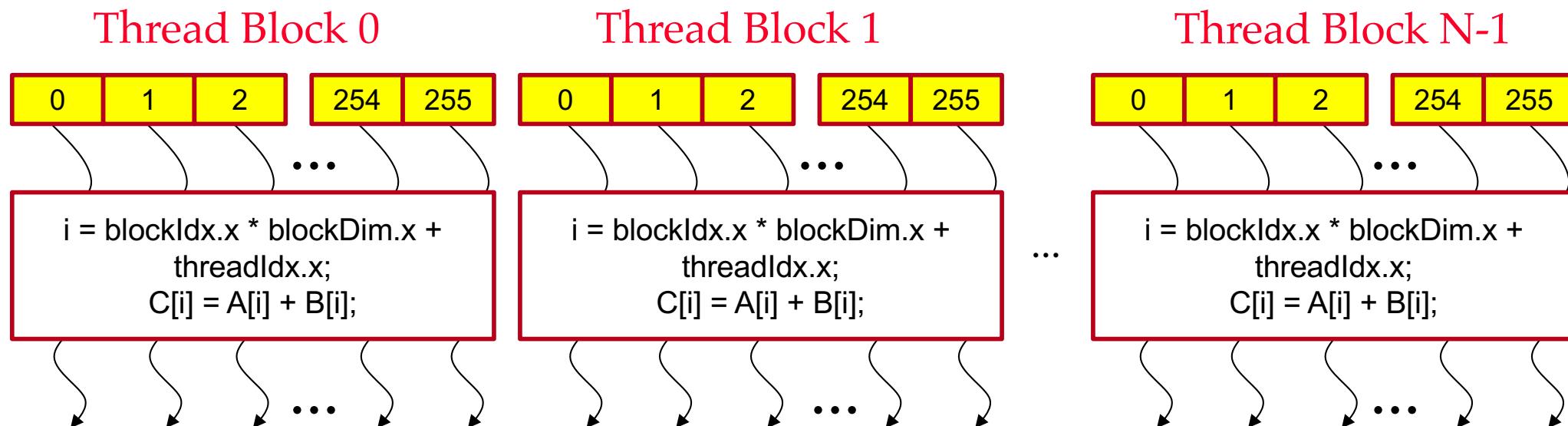


At a high-level, multiple threads work on same code (instructions) but different data

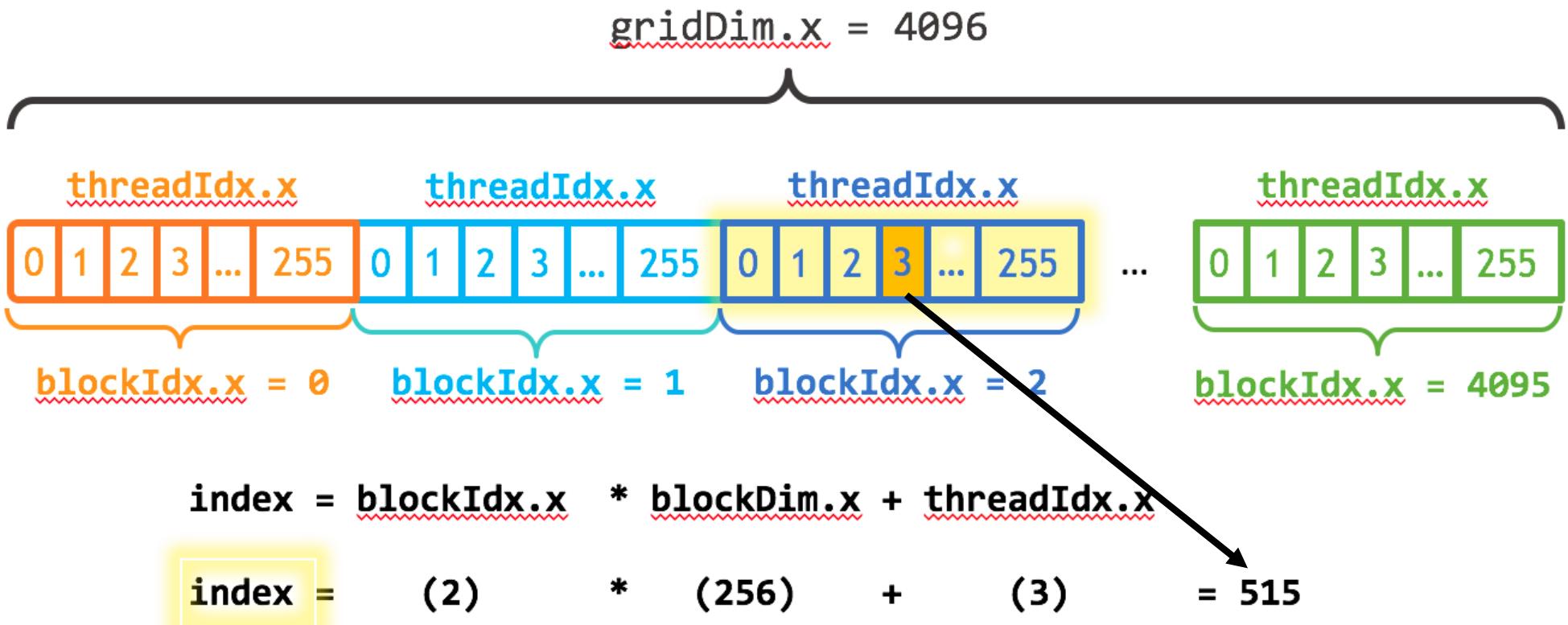


Kernel: Arrays of Parallel Threads

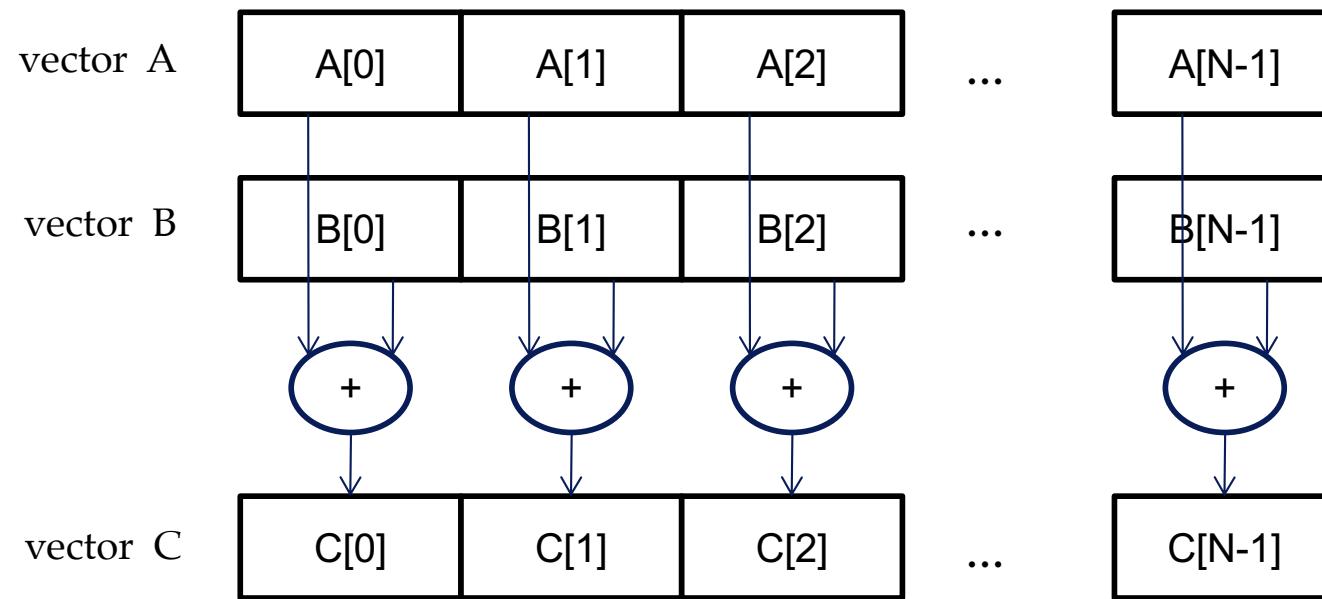
- A CUDA kernel is executed by a grid of threads
 - All threads in a grid run the same kernel code (Single Program Multiple Data)
 - Each thread has indexes that it uses to compute memory addresses and make control decisions



Kernel, Blocks, Threads



Vector Addition Example



Vector Addition – Traditional C Code

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

vecAdd CUDA Host Code

```
#include <cuda.h>

void vecAdd(float *h_A, float *h_B, float *h_C, int n)

{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to GPU (device) memory
    // Part 2
    // Kernel launch code – the device performs the vector addition
    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

Vector Addition (Host Side)

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Kernel invocation code – to be shown later

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// do processing of results
cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

Kernel Invocation code (Host Side)

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    ..... Preparation code (See previous slide)

    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;

    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, n);

    ... Post-processing (See previous slide)
}
```

Kernel Code (Device Side)

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

Same code is run by several threads

Important CUDA Syntax Extensions

- ❑ Declaration specifiers

```
__global__ void foo(...); // kernel entry point (runs on GPU)
```

- ❑ Syntax for kernel launch

```
foo<<<500, 128>>>(...); // 500 thread blocks, 128 threads each
```

- ❑ Built in variables for thread identification

```
dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
```

Example: Original C Code

```
void saxpy_serial(int n, float a, float *x, float
                  *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int main() {
    // omitted: allocate and initialize memory
    saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY
    // kernel
    // omitted: using result
}
```

CUDA Code

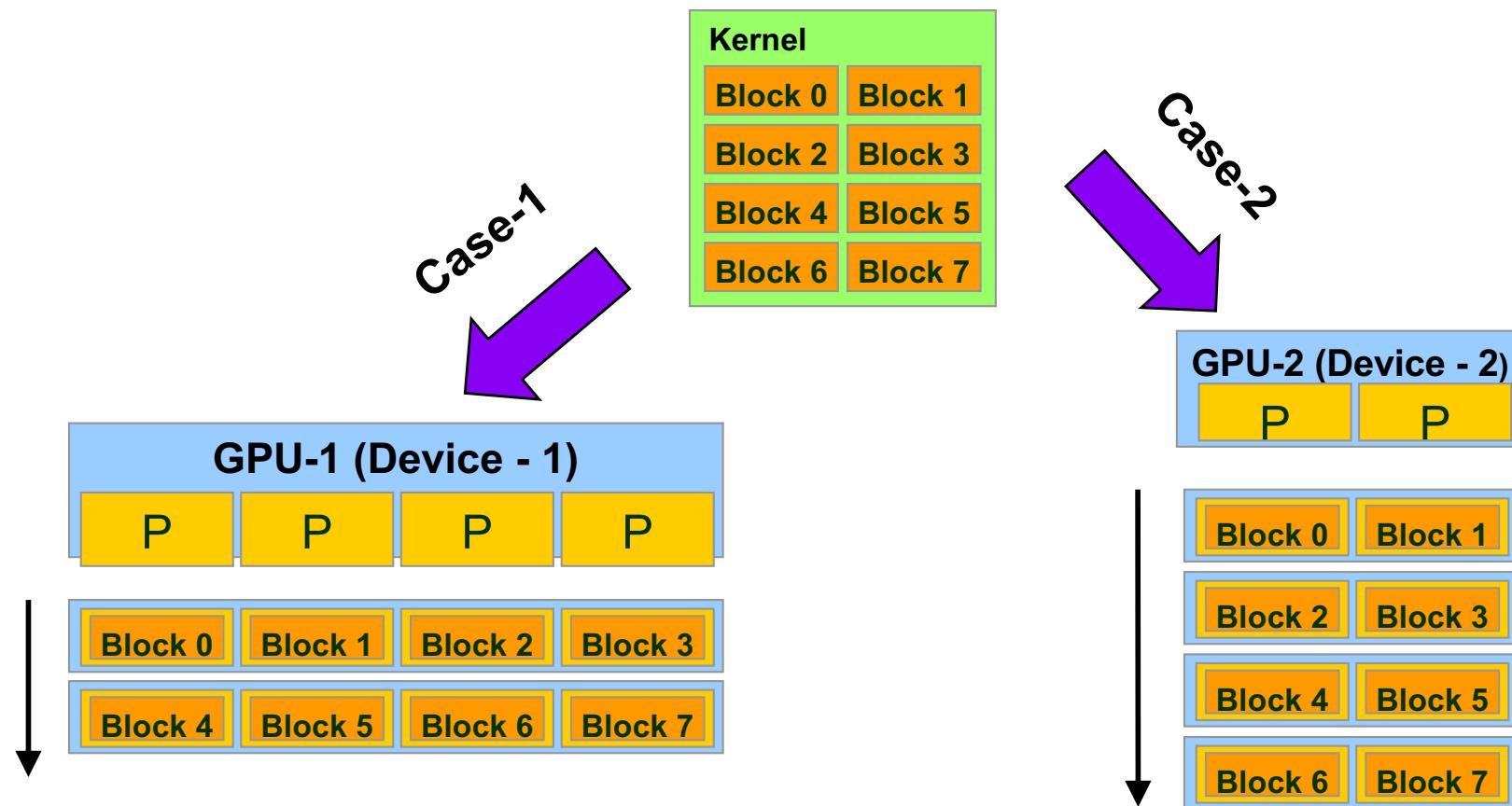
```
__global__ void saxpy(int n, float a, float *x, float *y) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if(i<n) y[i]=a*x[i]+y[i];  
}
```

Runs on GPU

```
int main() {  
  
    // omitted: allocate and initialize memory  
  
    int nblocks = (n + 255) / 256;  
  
    cudaMalloc((void**) &d_x, n);  
    cudaMalloc((void**) &d_y, n);  
    cudaMemcpy(d_x, h_x, n*sizeof(float), cudaMemcpyHostToDevice);  
    cudaMemcpy(d_y, h_y, n*sizeof(float), cudaMemcpyHostToDevice);  
    saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);  
    cudaMemcpy(h_y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);  
  
    // omitted: using result  
}
```

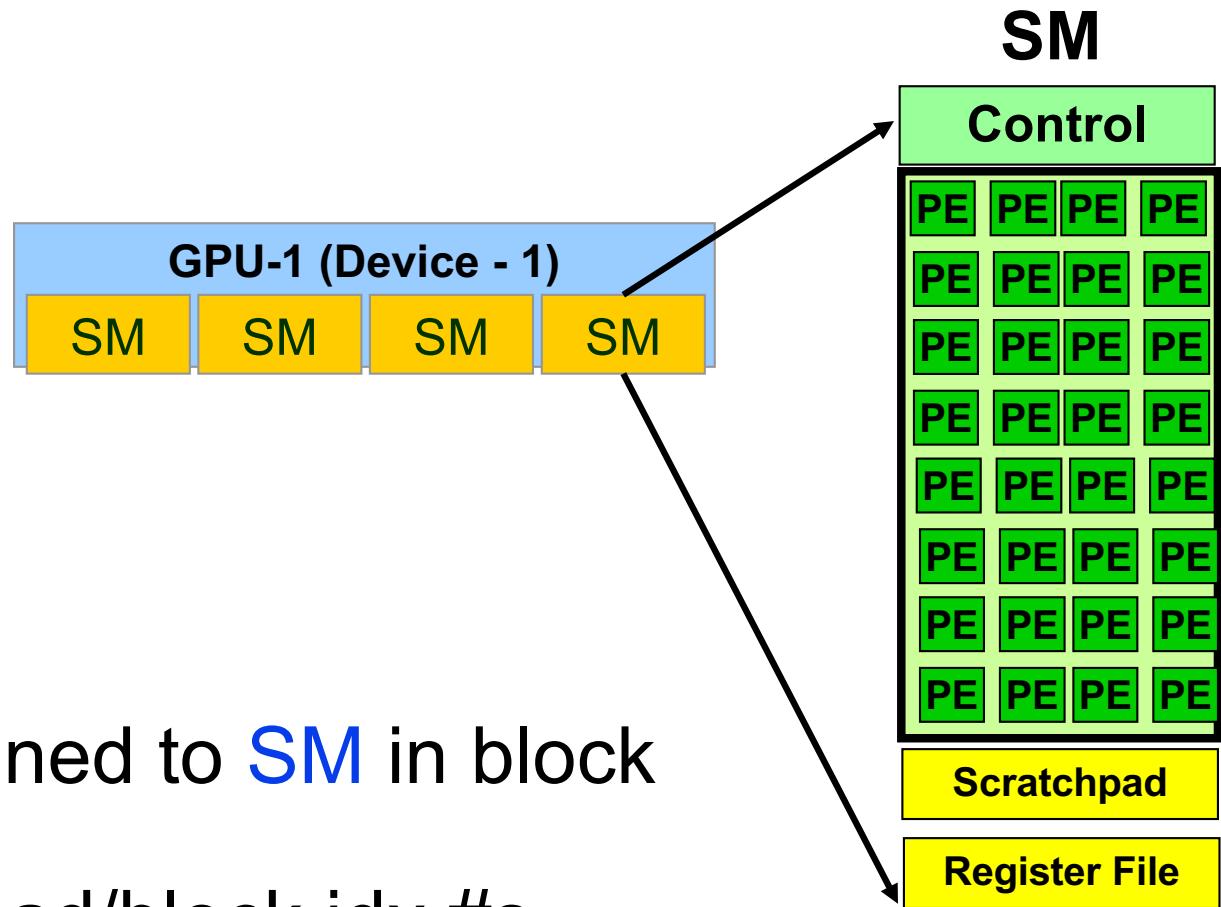
Code to Hardware Mapping

Code to Hardware Mapping: Transparent Scalability



- ❑ Each block can execute in any order relative to others.
- ❑ GPU Hardware is free to assign blocks to any processor at any time
 - A kernel scales to any number of parallel processors.

Streaming Multi-Processor (SM)



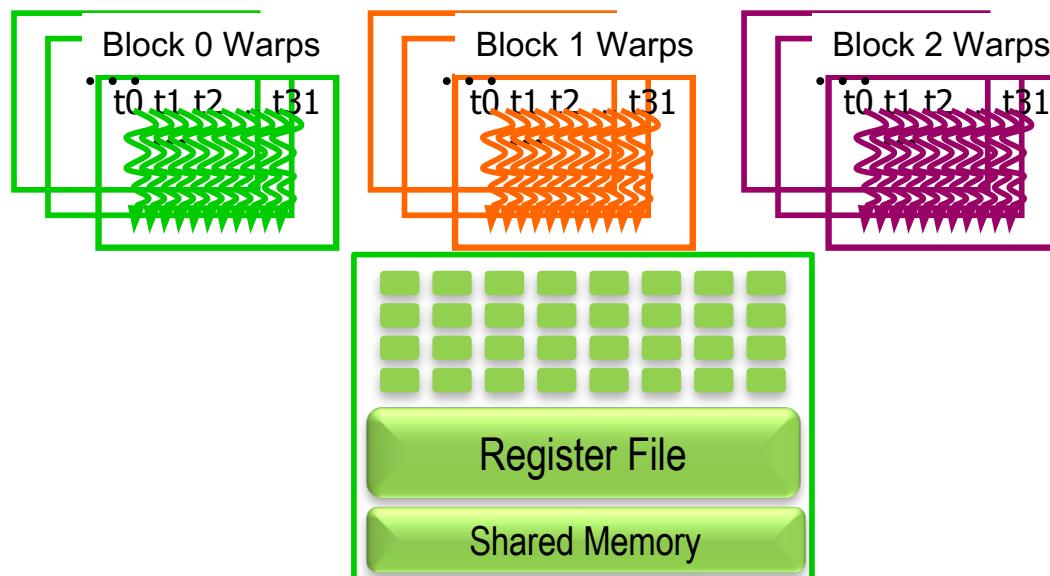
- Threads are assigned to **SM** in block granularity
- SM maintains thread/block idx #s
- SM manages/schedules thread execution
- Multiple blocks can be allocated to the SM
 - Based on the amount of resources (shared memory, register file etc.)

GPU Execution Model

- ❑ Blocks assigned to each SM are scheduled on the associated SIMD hardware (i.e., on the Processing Elements (PEs)).
- ❑ SM bundles threads (from various blocks) into **warps** (wavefronts) and runs them in lockstep on across PEs.
- ❑ An NVIDIA warp groups 32 consecutive threads together (AMD wave-fronts group 64 threads together)
- ❑ Warps are:
 - Scheduling units in SM
 - Scheduled in multiplexed and pipelined manner on the SM

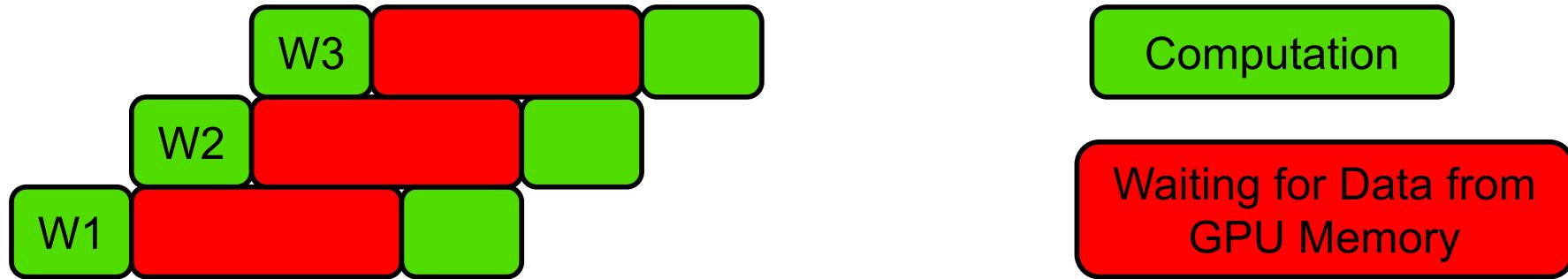
Warp Example

- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
 - Each Block is divided into $256/32 = 8$ Warps
 - There are $8 * 3 = 24$ Warps



Tolerating Long Latencies

- Execution in an SM



GPU attempts to hide long memory latency with computation from other warps

How SMs are able to context switch between warps so quickly?

Summary

- ❑ Spawns more threads than GPU can run (some may wait)
- ❑ Organize threads into “blocks” (up to 1024 threads per block)
- ❑ Motivation: Write parallel software once and run on future hardware
- ❑ Warps associated with blocks can help in tolerating long latencies.
- ❑ GPUs support large register files (for fast context switching) and high bandwidth memories (for providing data to large number of concurrent threads)

Reading Material

- D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach, 3rd Edition”
- *Patterson and Hennesy, Computer Organization and Design, 5th Edition, Appendix C-2 on GPUs*
- More background material: Jog et al., OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU performance, ASPLOS’13
- *Nvidia CUDA C Programming Guide*
 - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>