Технології графічного процесінгу

(Масивно-паралельні обчислення на графічних прискорювачах
…
Massively Parallel Computing on Graphic Processing Units - GPUs)

Lecture 3. CUDA –
Review + Current Trends

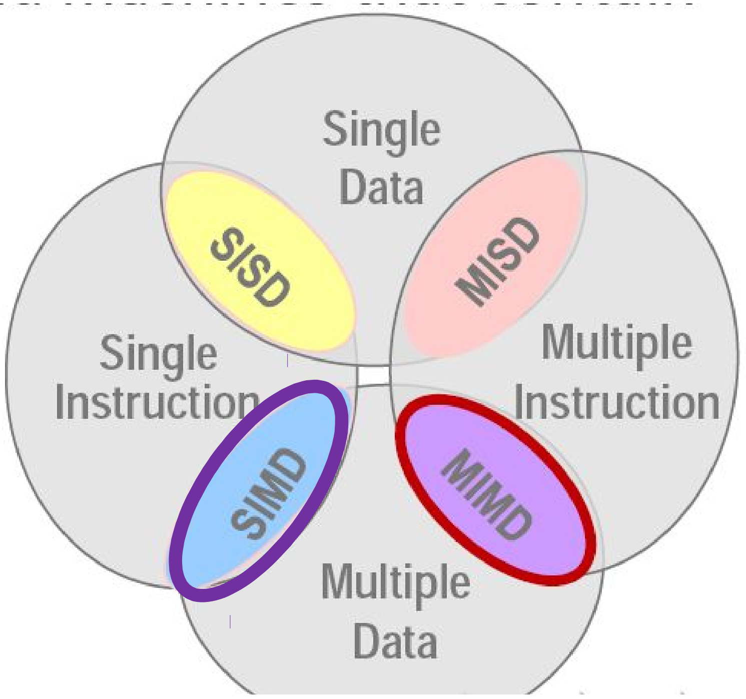
Yuri G. Gordienko
(NTUU-KPI, 2021)

(on the basis of materials by NVIDIA, T.Lanfear, P.Pomorski, W.Hwu, D.Kirk)

From the last lectures (01): Distributed Computing – Architectural Models

Flynn's Taxonomy:

- **SISD:** traditional uniprocessor computers
- **MISD:** Space Shuttle flight control computer
- SIMD: array processor, GPU.
- MIMD: parallel systems, **distributed systems, multi-GPU**.



From the last lectures (01): GPU Computing - Definition

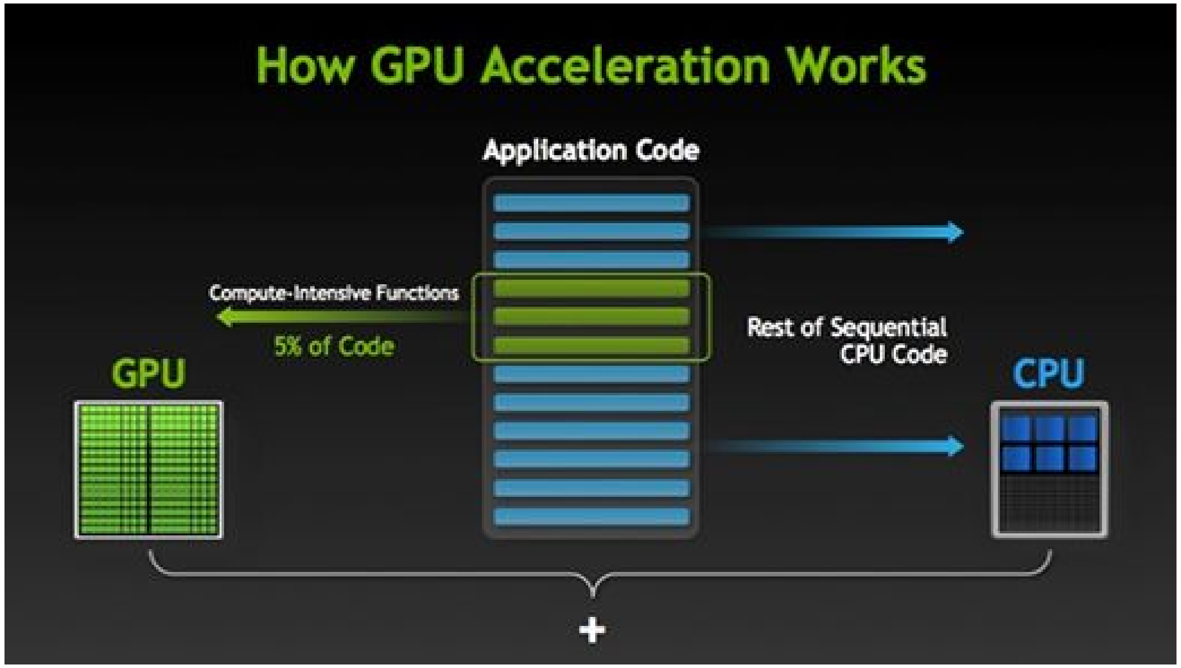
What is GPU Computing?

General-purpose computing on graphics processing units (**GPGPU** or **GPU**)

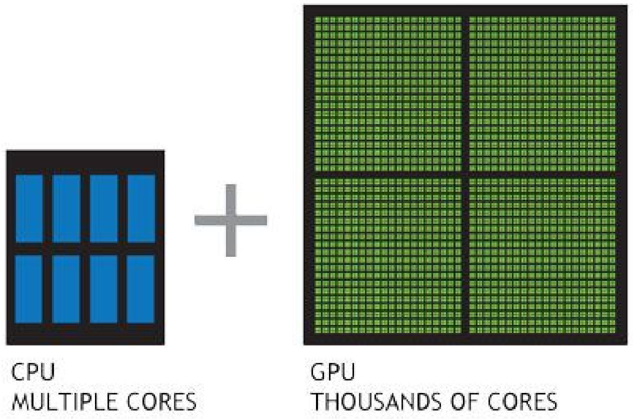
Work: vector instructions (**SIMD**), only effective for problems that can be solved using stream processing (data for similar computation) **SIMD - why it is distributed? – independent from CPU, several graphic cards can be integrated in PC, clusters, etc.**

Applications: calculations, gaming, multimedia.

From the last lectures (01):
GPU Computing - Scheme


How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
Rest of Sequential CPU Code
GPU
CPU
+

(C) NVIDIA



CPU versus GPU

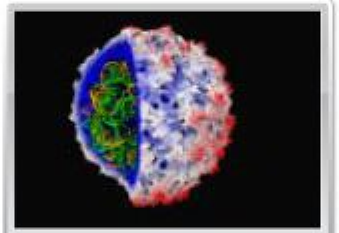
NVIDIA “Tesla K40” card: **2880** parallel processing cores. Compare: **1.3 TFLOPs <-> 2-8 GFLOPs in PC**!

From the last lectures (01):

GPU Computing - Examples

146X

Interactive visualization of volumetric white matter connectivity

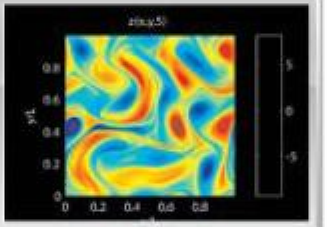


36X

Ionic placement for molecular dynamics simulation on GPU

19X

Transcoding HD video stream to H.264



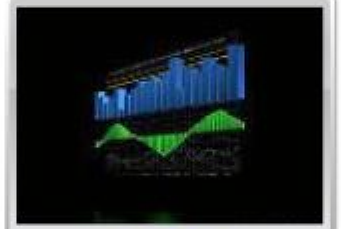
17X

Fluid mechanics in Matlab using .mex file CUDA function

100X

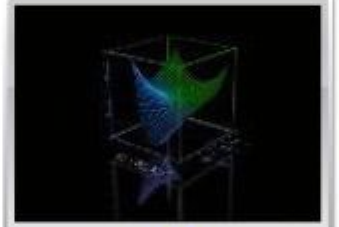
Astrophysics N-body simulation

(C) Srinivasan



149X

Financial simulation of LIBOR model with swaptions



47X

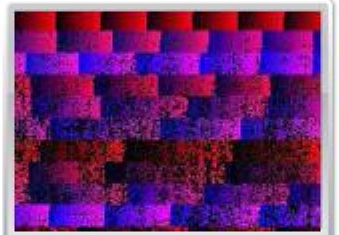
GLAME@lab: an M-script API for GPU linear algebra

20X

Ultrasound medical imaging for cancer diagnostics

24X

Highly optimized object oriented molecular dynamics



30X

Cmatch exact string matching to find similar proteins and gene sequences

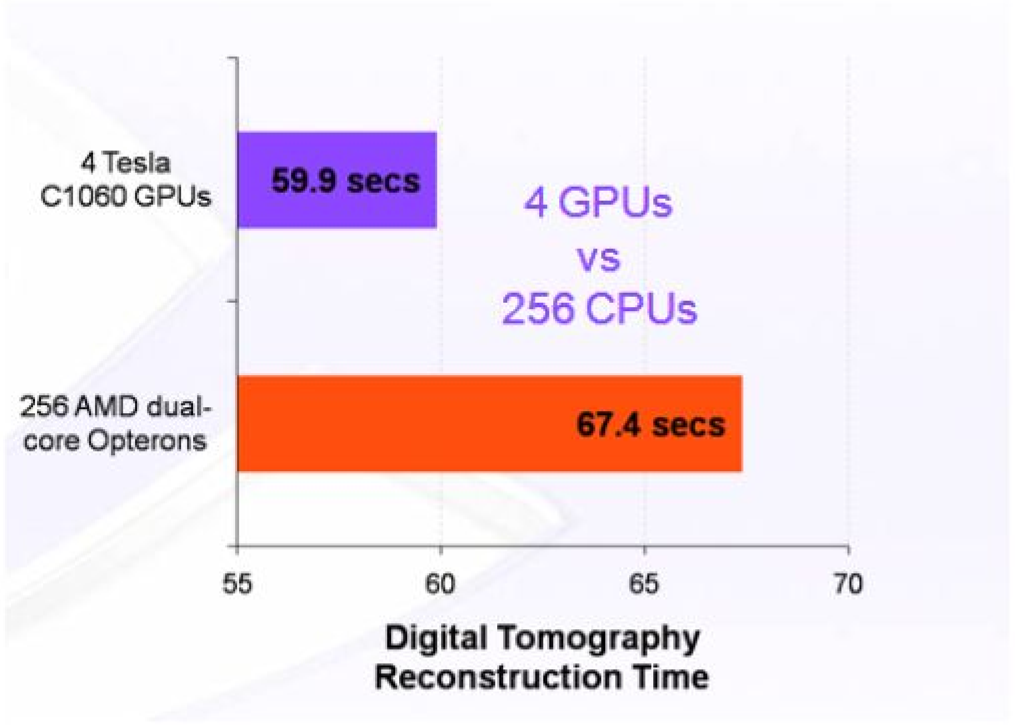
Science (above), gaming, multimedia

From the last lectures (01):
GPU Computing - Examples

Desktop beats Cluster

(C) Srinivasan

CalcUA
$5 Million



Tesla Personal Supercomputer
$10,000

Again: SIMD - why it is distributed? – independent from CPU, several graphic cards can be integrated in PC, clusters, etc.

From the last lectures (01): Distributed Computing - Illustration

Mythbusters:

- Adam
- Jamie

Vivid presentation on GPU-principle at NVIDIA conference (2008)

From the last lectures (01):

GPU

D[illegible]ted Computing - Illustration

Mythbusters:

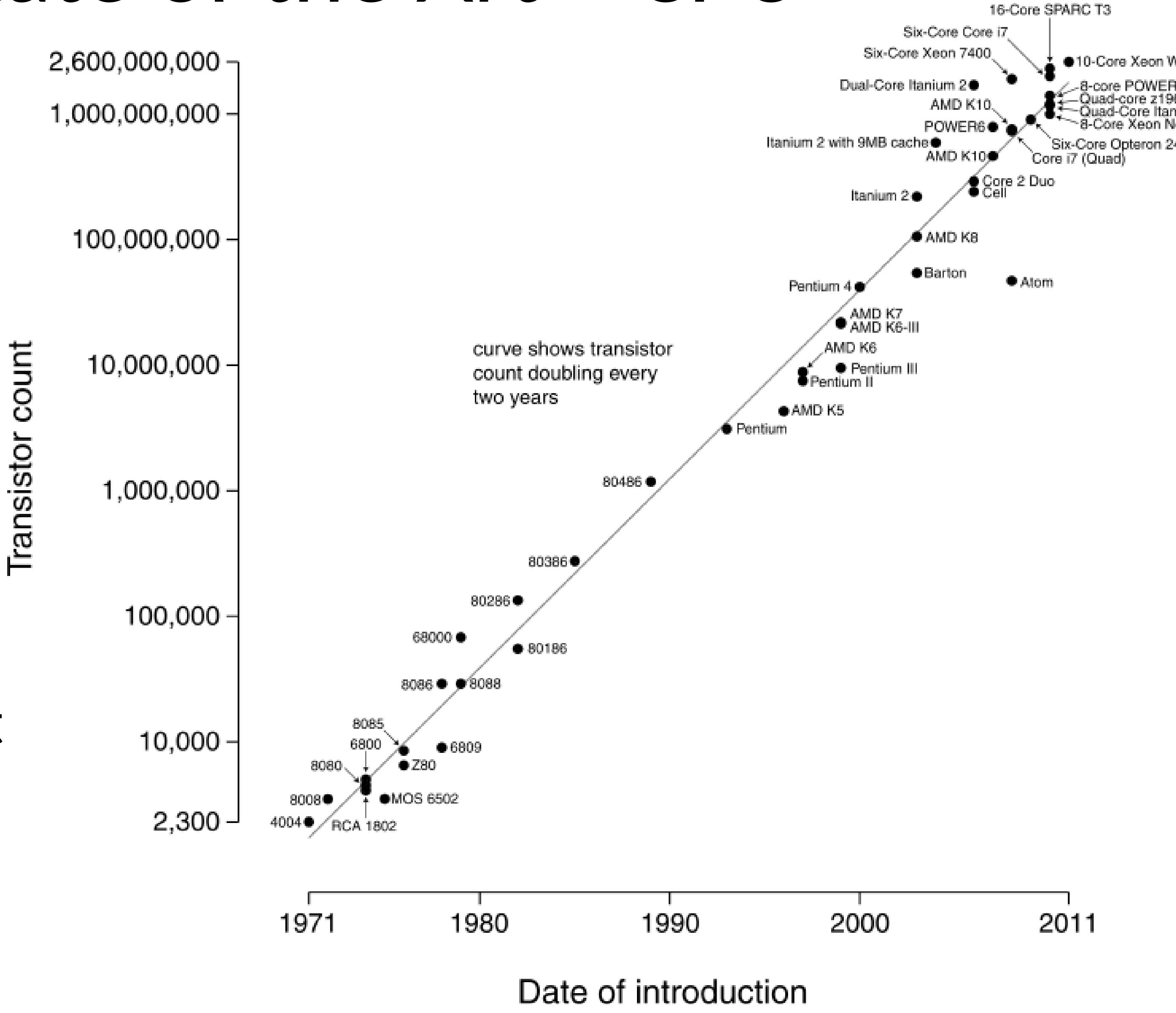
- Adam
- Jamie

Vivid presentation on GPU-principle at NVIDIA conference (2008)

GPU computing – why we need it?

From the last lectures (01):
State of the Art – CPU

Moore’ Law: CPU transistors versus dates of introduction. The line corresponds to exponential growth with transistor count doubling every two years.



Serial Performance Scaling is Over

- Cannot continue to scale processor frequencies
  - no 10 GHz chips
- Cannot continue to increase power consumption
  - can’t melt chip
- Can continue to increase transistor density
  - as per Moore’s Law

How to Use Transistors?

Instruction-level parallelism:

- out-of-order execution, speculation, …

Data-level parallelism:

- vector units, SIMD execution, …
- SSE, AVX (Advanced Vector Extensions), GPU

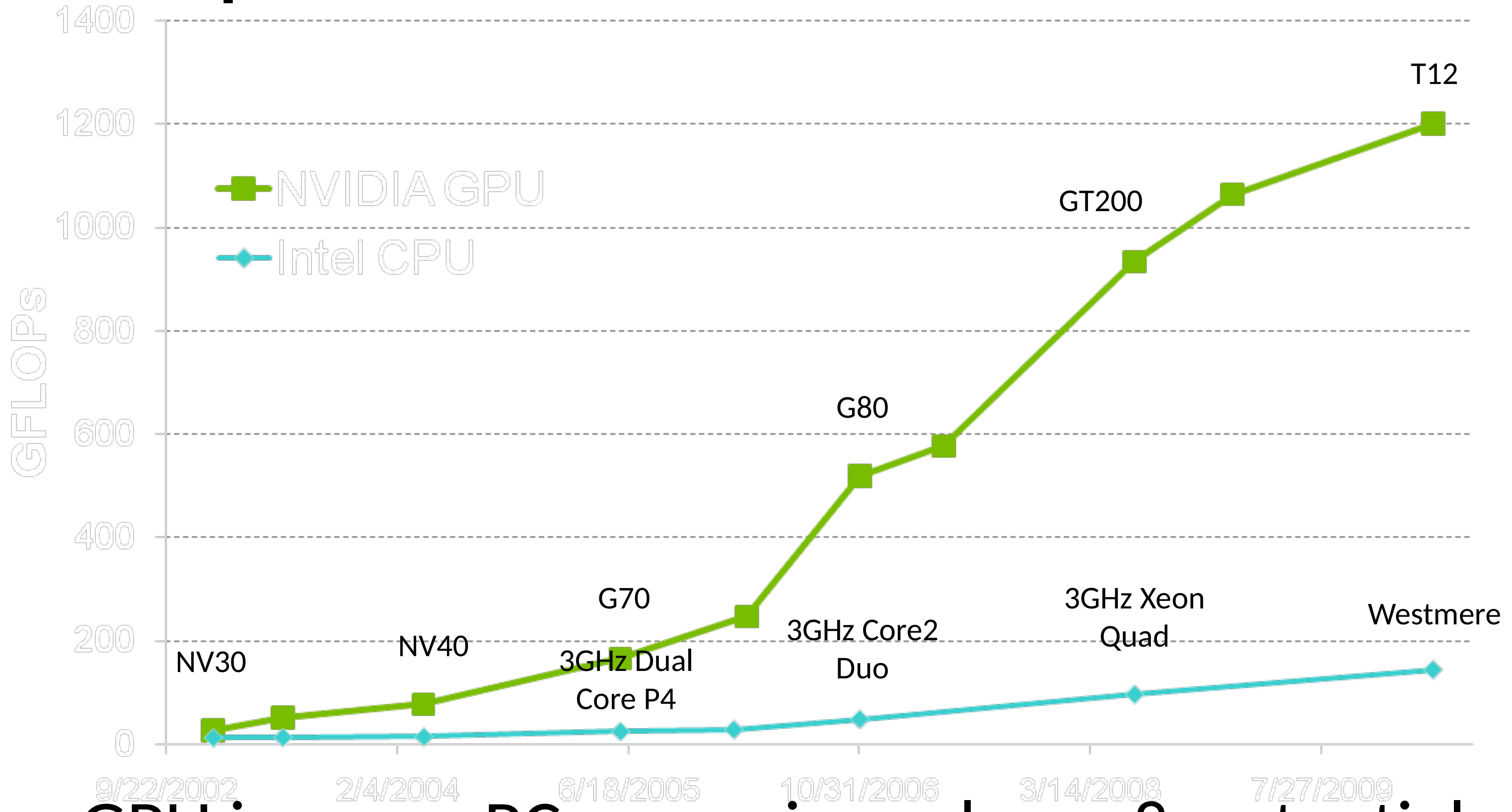
Thread-level parallelism:

- multithreading, multicore, manycore…
- Intel Core2, AMD Phenom, NVIDIA Fermi, …

Why Massively Parallel Processing?

A quiet revolution and potential build-up

Computation: TFLOPs vs. GFLOPs

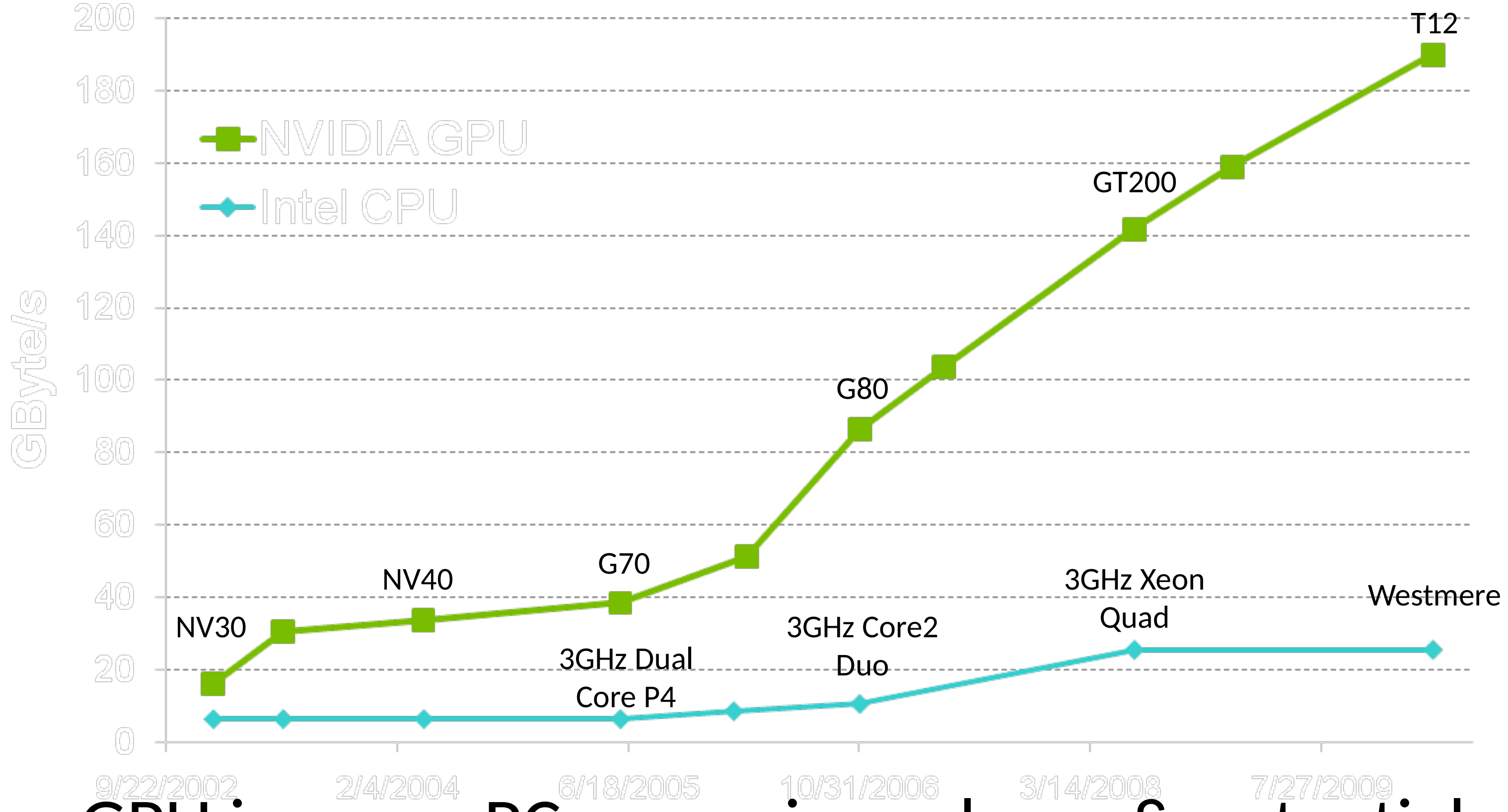


GPU in every PC – massive volume & potential impact

Why Massively Parallel Processing?

A quiet revolution and potential build-up

Bandwidth: ~10x



GPU in every PC – massive volume & potential impact

The “New” Moore’s Law?

Computers no longer get faster, just wider!

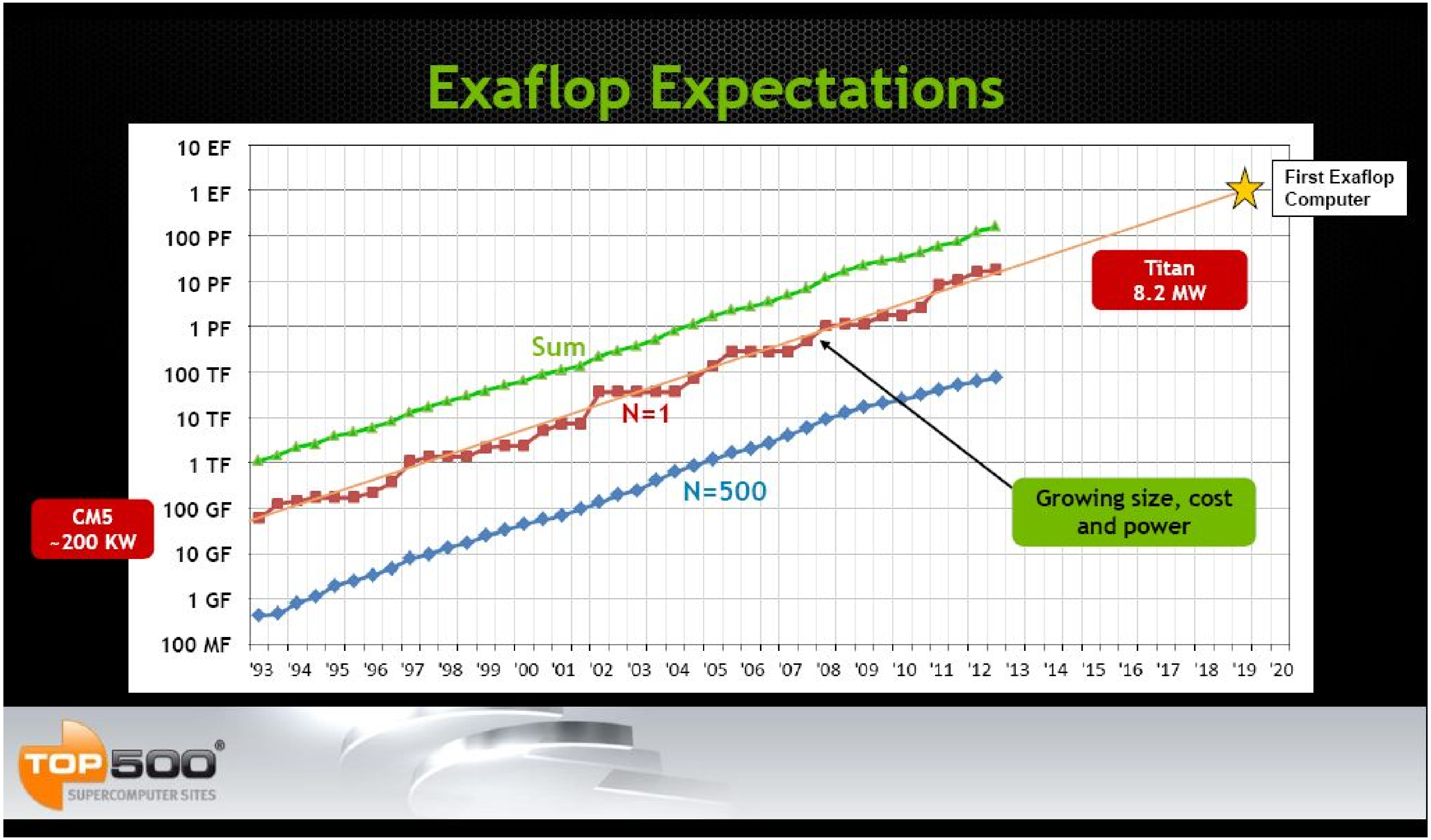
We *must* re-write algorithms to be parallel !

Data-parallel computing is most scalable solution

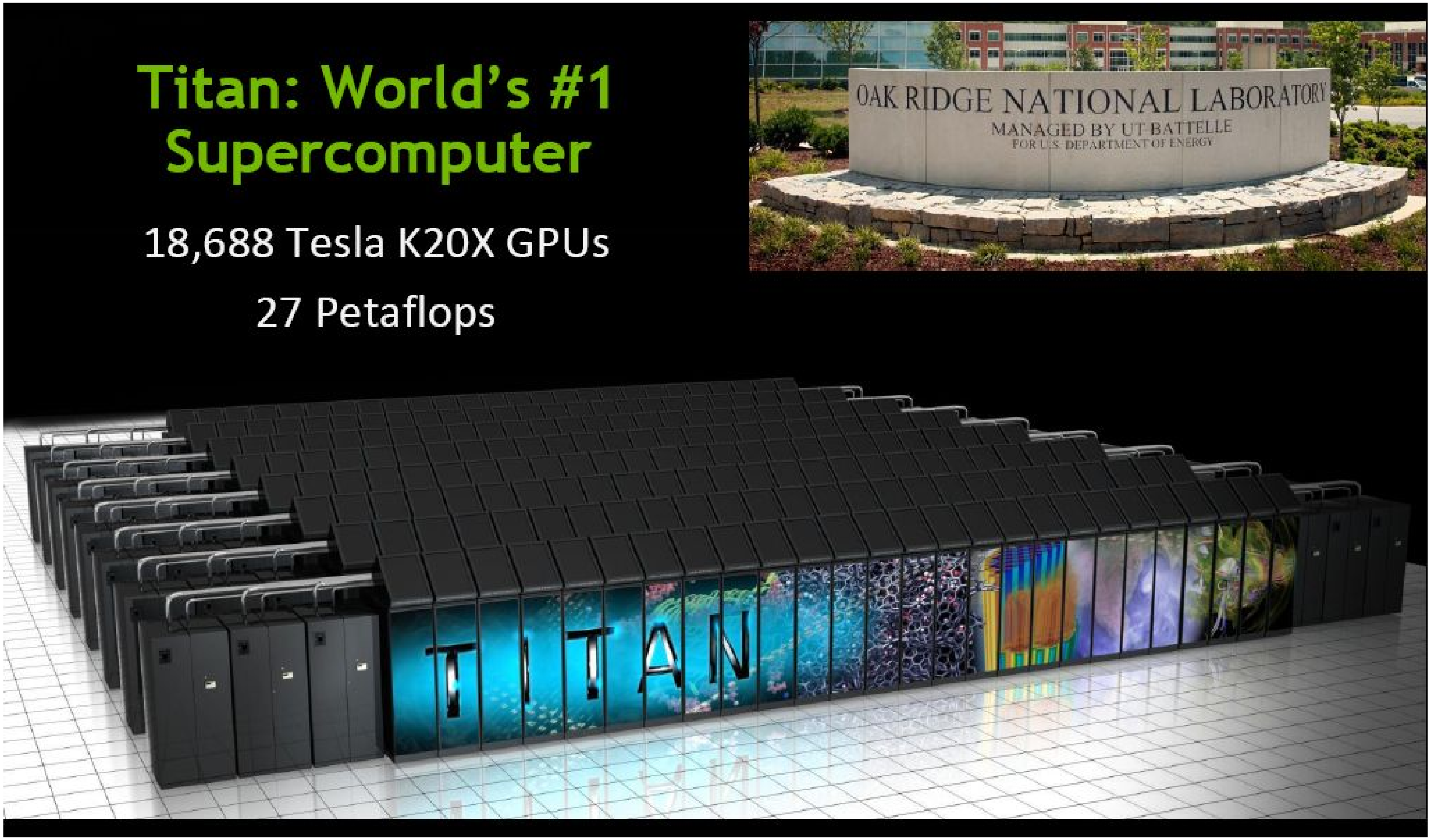
We always have more data than cores –

then let’s build the computation around the data

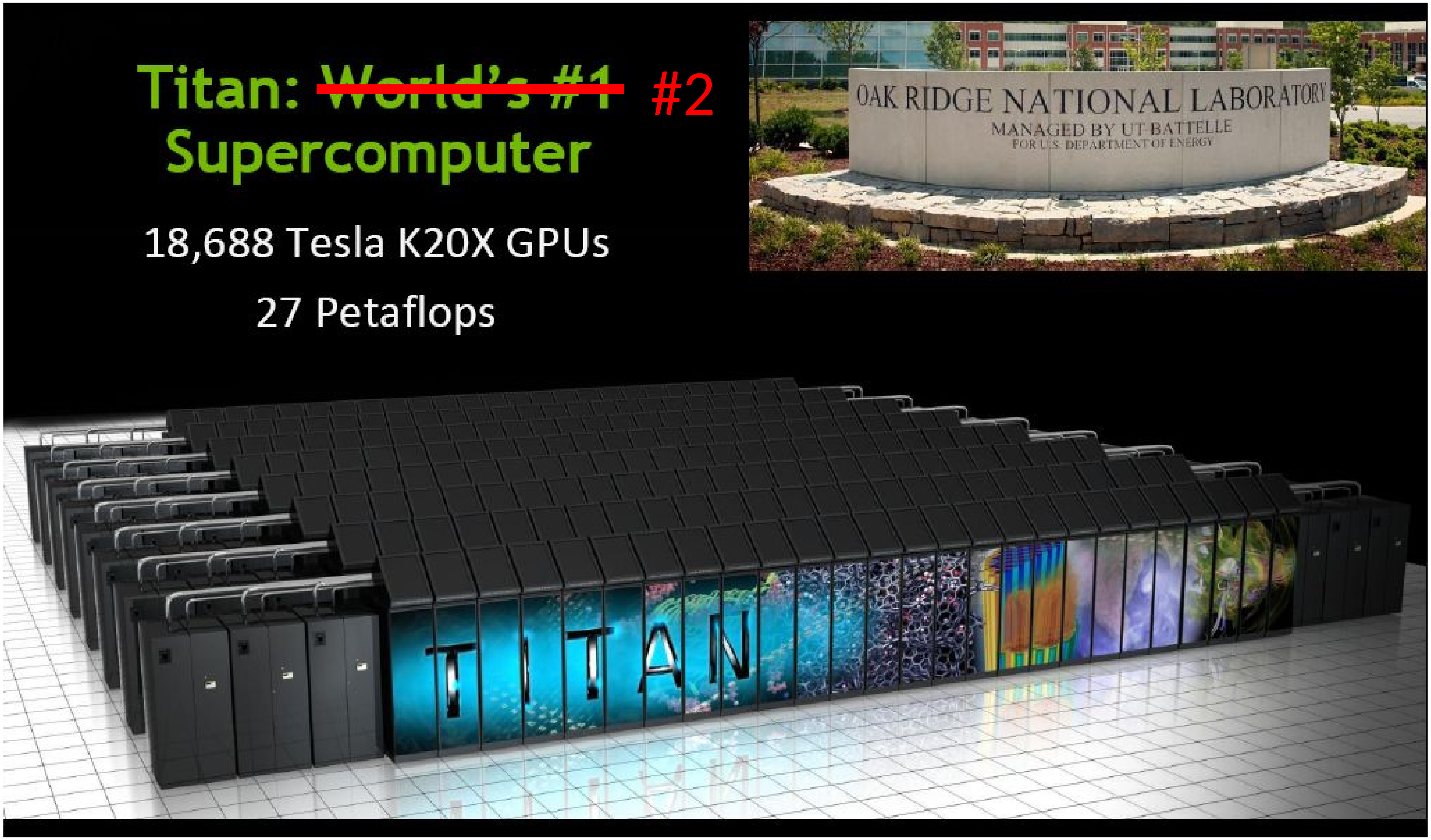
Progress for: 1st and top 500 (N=500)



The champion


Titan: World's #1 Supercomputer
18,688 Tesla K20X GPUs
27 Petaflops
OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE
FOR U.S. DEPARTMENT OF ENERGY
TITAN

The champion

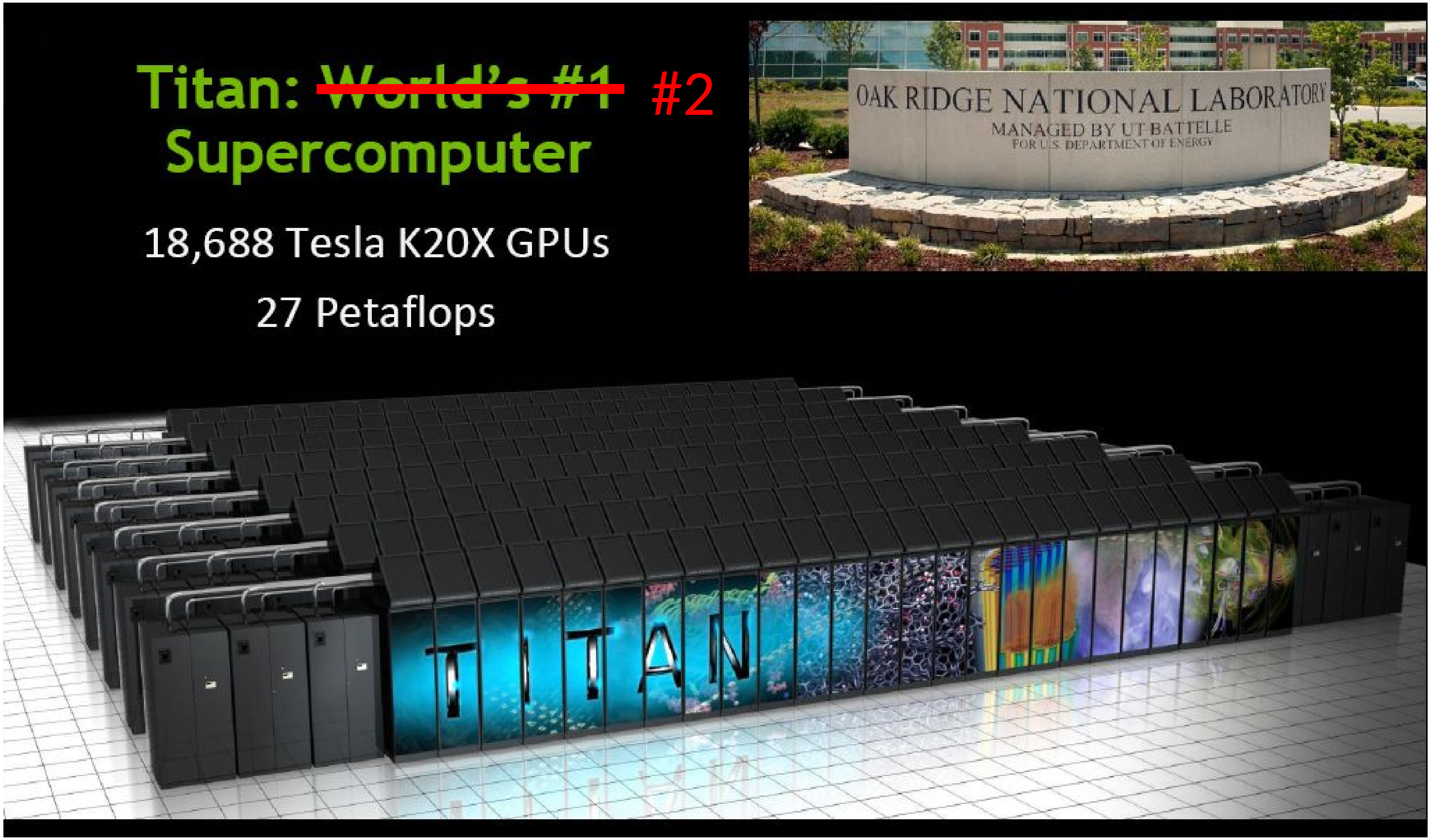


Titan: ~~World's #1~~ #2 Supercomputer

18,688 Tesla K20X GPUs

27 Petaflops

The champion: Tianhe-2 (China) 33 PFs

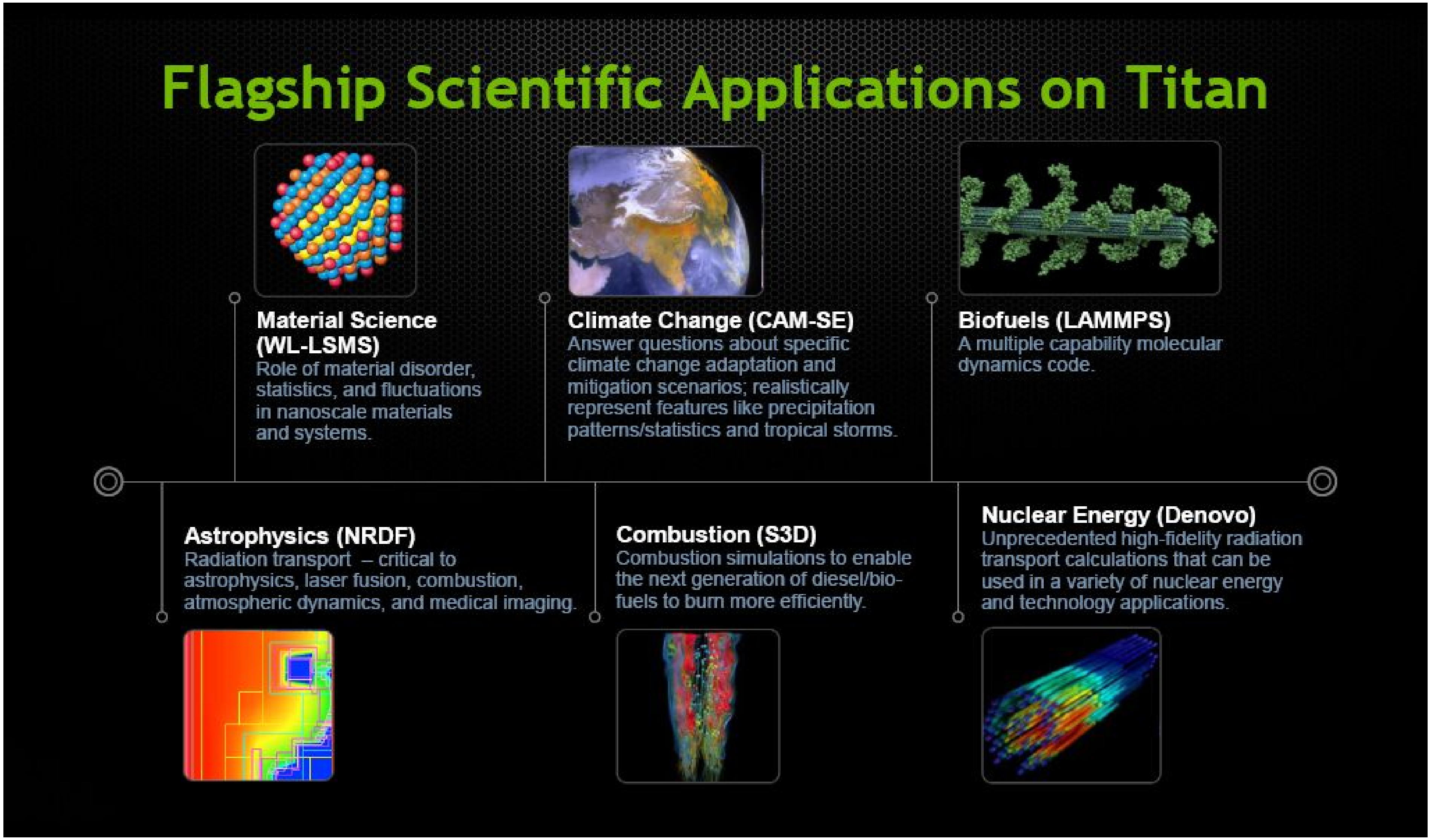


Titan: ~~World's #1~~ #2
Supercomputer

18,688 Tesla K20X GPUs

27 Petaflops

The most important applications



Flagship Scientific Applications on Titan

Material Science (WL-LSMS)
Role of material disorder, statistics, and fluctuations in nanoscale materials and systems.

Climate Change (CAM-SE)
Answer questions about specific climate change adaptation and mitigation scenarios; realistically represent features like precipitation patterns/statistics and tropical storms.

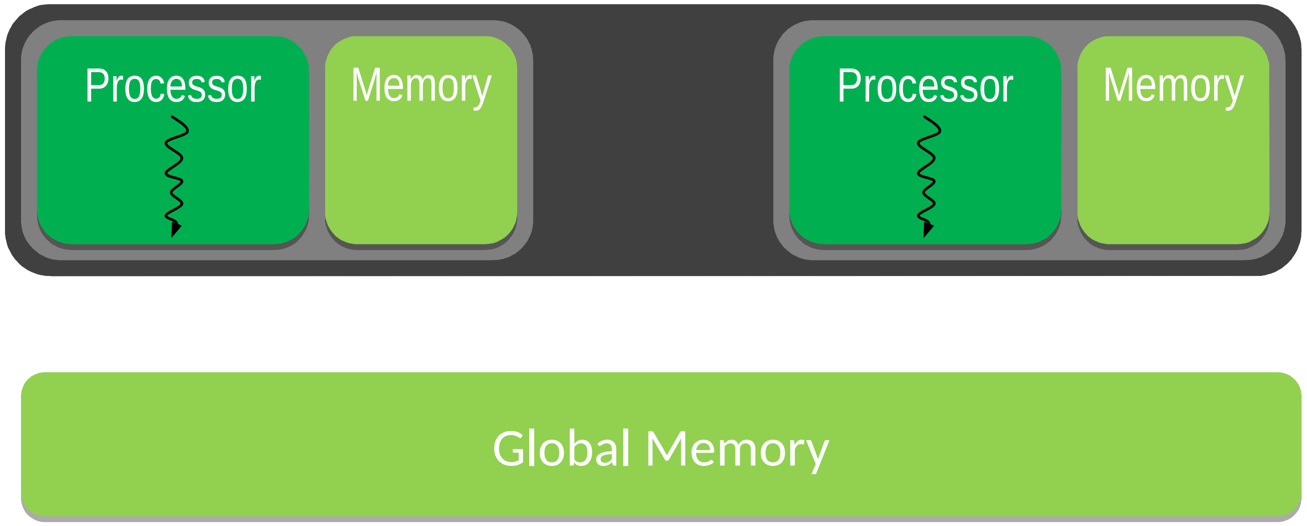
Biofuels (LAMMPS)
A multiple capability molecular dynamics code.

Astrophysics (NRDF)
Radiation transport – critical to astrophysics, laser fusion, combustion, atmospheric dynamics, and medical imaging.

Combustion (S3D)
Combustion simulations to enable the next generation of diesel/bio-fuels to burn more efficiently.

Nuclear Energy (Denovo)
Unprecedented high-fidelity radiation transport calculations that can be used in a variety of nuclear energy and technology applications.

Generic Multicore Chip

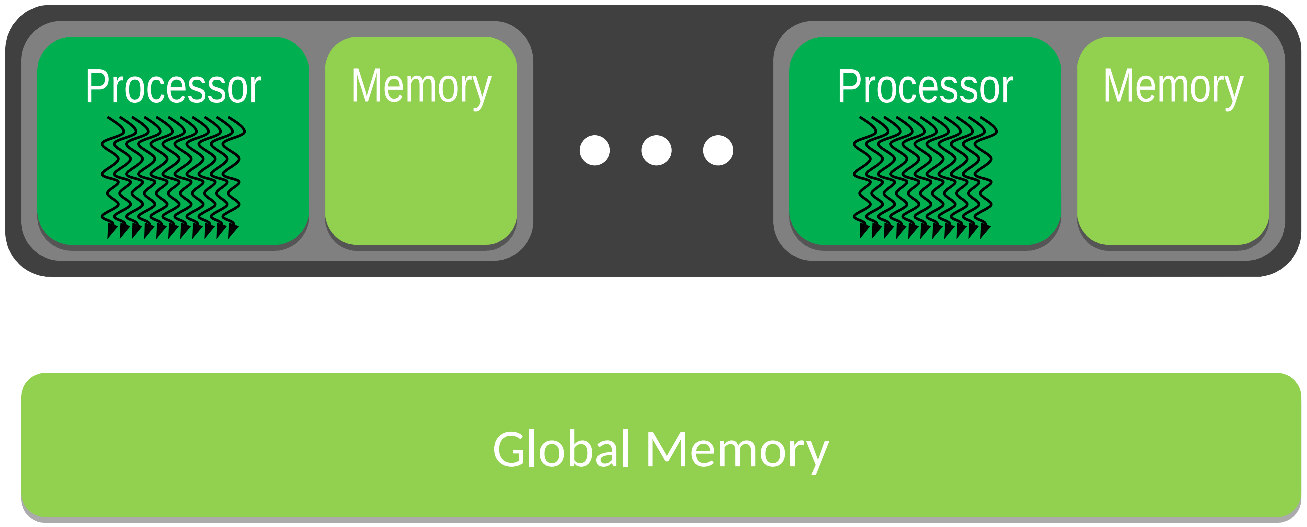


Handful of processors each supporting ~1 hardware thread

On-chip memory near processors (cache, RAM, or both)

Shared global memory space (external DRAM)

Generic Manycore Chip



Many processors each supporting many hardware threads

On-chip memory near processors (cache, RAM, or both)

Shared global memory space (external DRAM)

How many cores in GPU?

GeForce 8800 Ultra (2007) - 128

GeForce GTX 260 (2008) - 192

GeForce GTX 295 (2009) - 480*

GeForce GTX 480 (2010) - 480

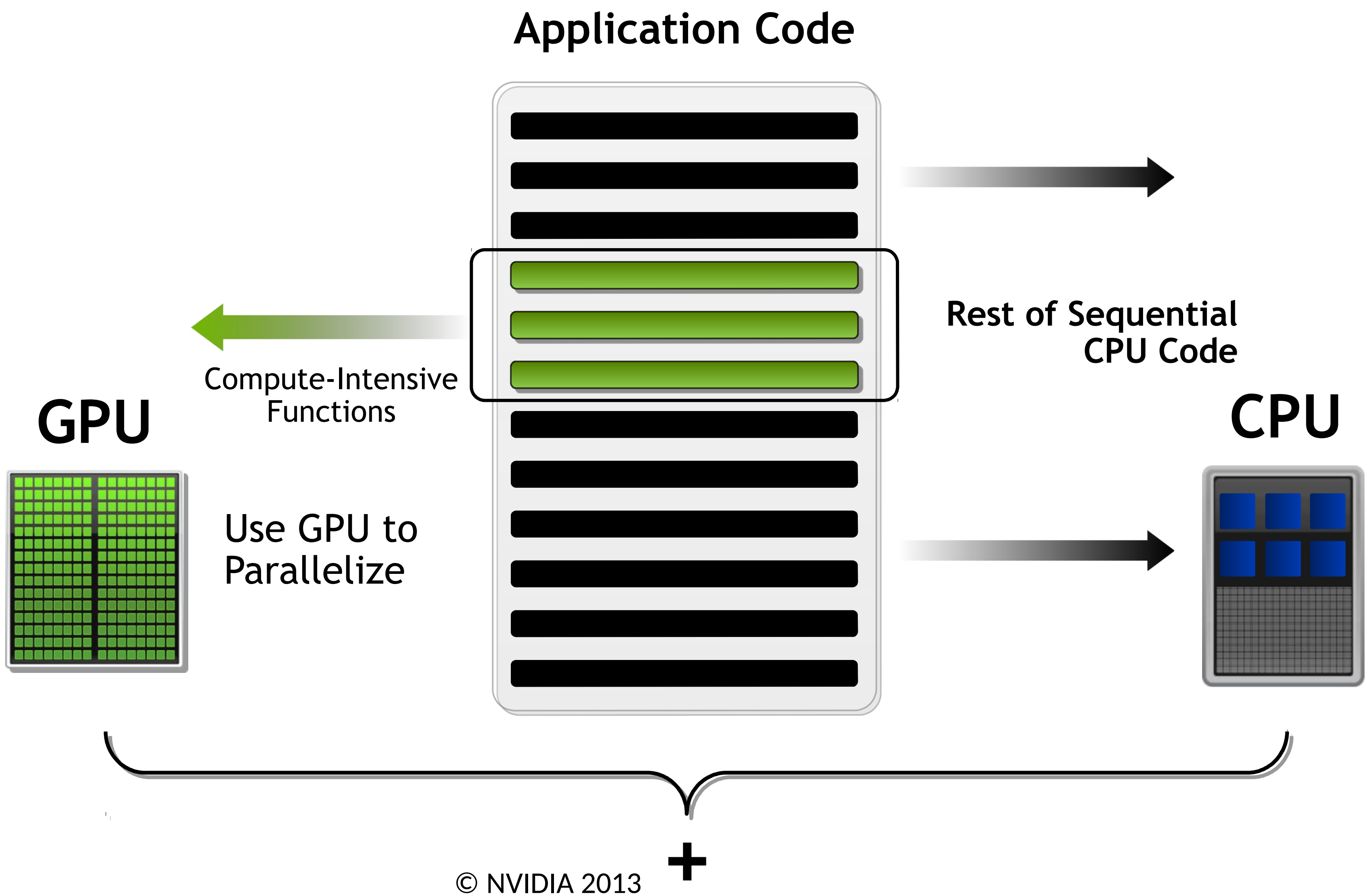
GeForce GTX 590 (2011) - 1024*

GeForce GTX 690 (2012) - 3072*

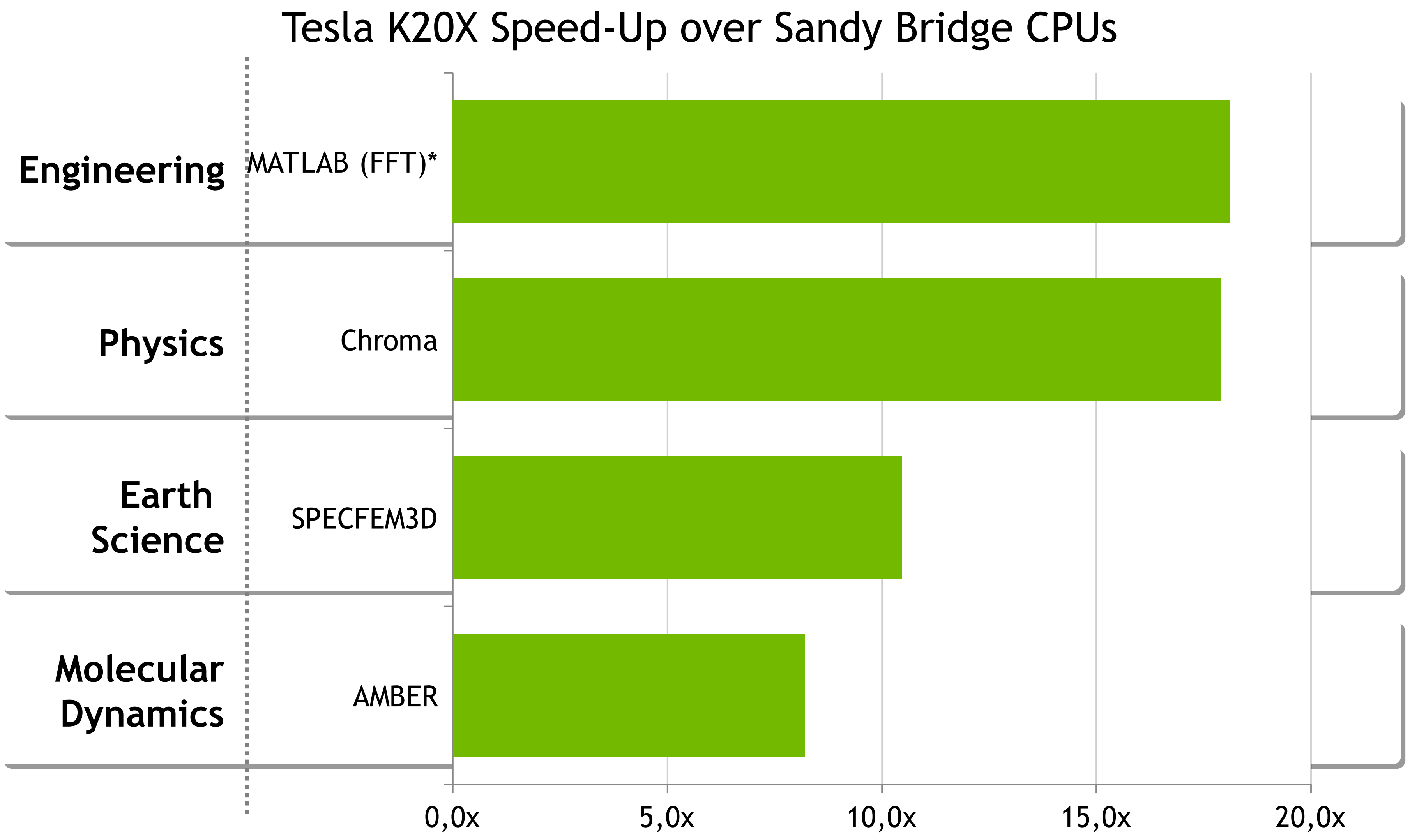
GeForce GTX Titan Z (2014) - 5760*

* indicates these are cards shipped with 2 GPUs in them, effectively doubling the cores

Small Changes, Big Speed-up



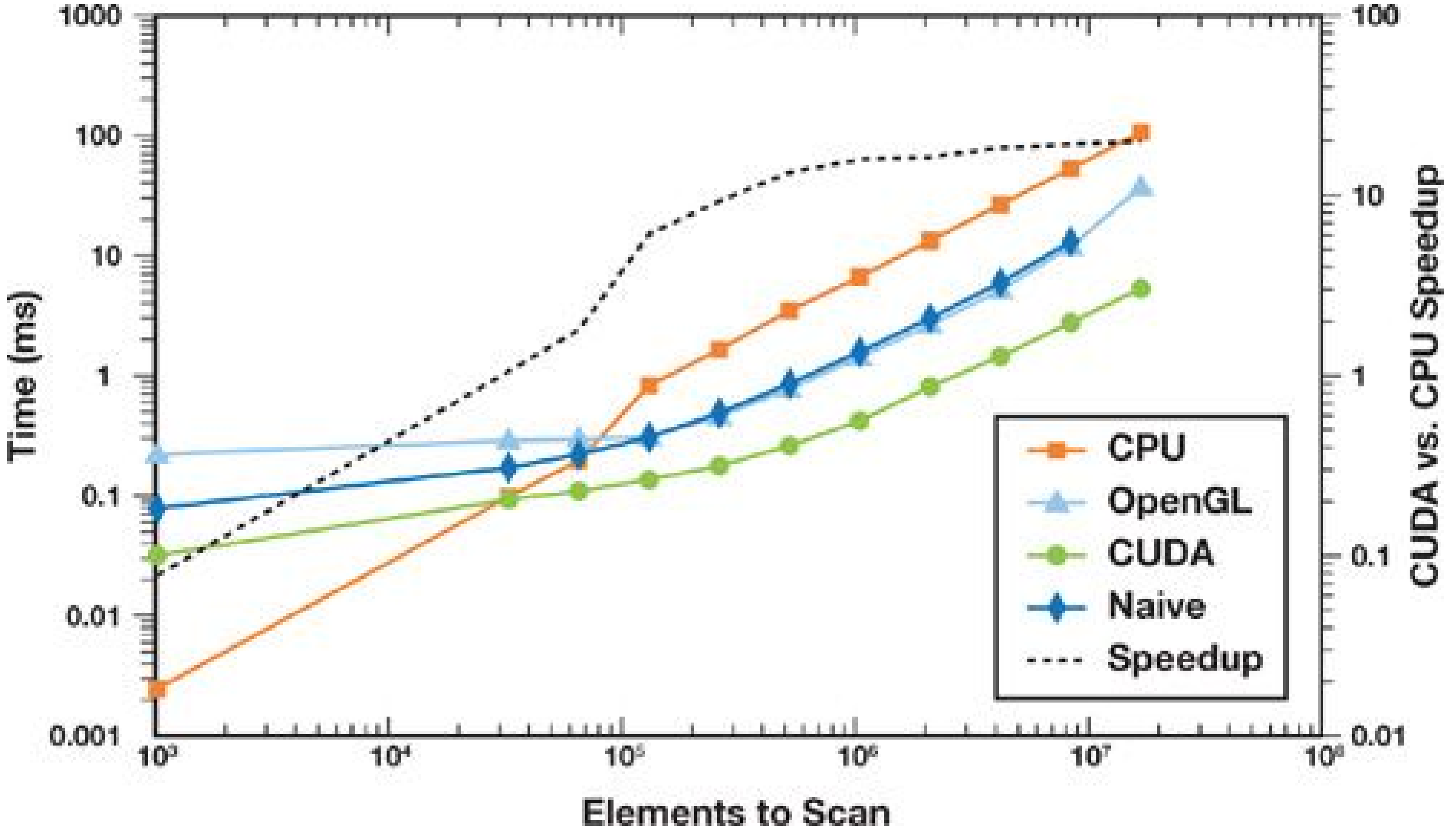
Fastest Performance on Scientific Applications



CPU results: Dual socket E5-2687w, 3.10 GHz, GPU results: Dual socket E5-2687w + 2 Tesla K20X GPUs
*MATLAB results comparing one i7-2600K CPU vs with Tesla K20 GPU
Disclaimer: Non-NVIDIA implementations may not have been fully optimized

GPU – Speedup?

What kinds of speedups do we get with GPU?

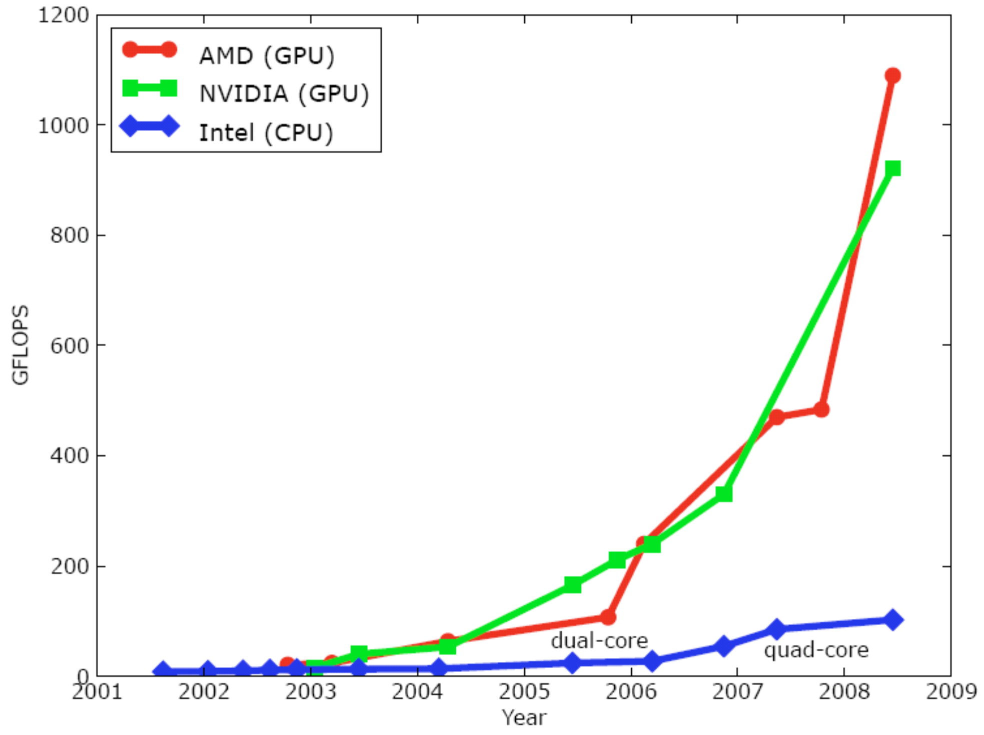


Performance Advantage of GPUs

An enlarging peak performance advantage:

Calculation: 1 TFLOPS vs. 100 GFLOPS

Memory Bandwidth: 100-150 GB/s vs. 32-64 GB/s



(C) John Owens

GPU in every PC and workstation – massive volume and potential impact

Harvesting Performance Benefit of Many-core GPU Requires

- Massive parallelism in application algorithms
  - Data parallelism

- Regular computation and data accesses
  - Similar work for parallel threads

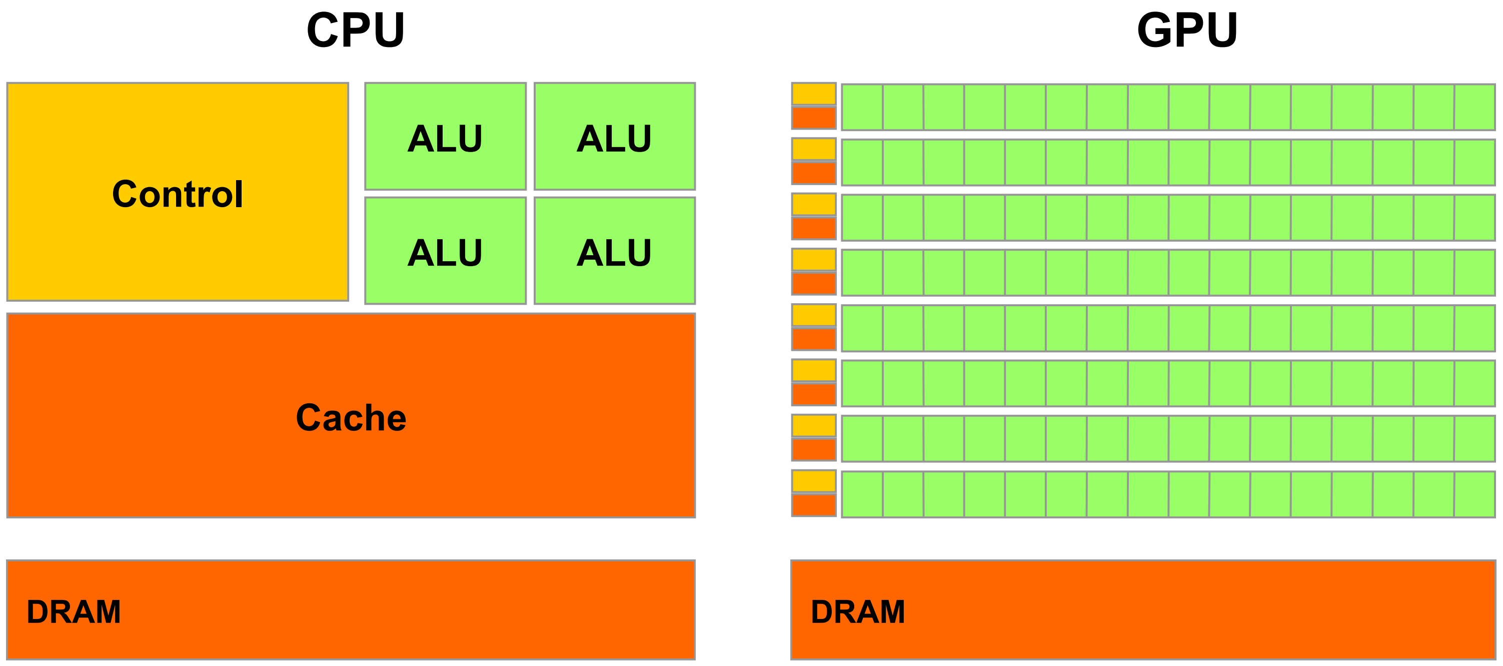
- Avoidance of conflicts in critical resources
  - Off-chip DRAM (Global Memory) bandwidth
  - Conflicting parallel updates to memory locations

CPU vs GPU – what is the difference?

Why is this different from a CPU?

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches share control logic across many threads

CPU vs. GPU: fundamentally different design



CPU
- **task** parallelism (diverse tasks)
- minimize latency
- multithreaded
- some SIMD

GPU
- excel at number crunching
- **data** parallelism (single task)
- maximize throughput
- super-threaded
- large-scale SIMD

From the last lectures (01):

GPU

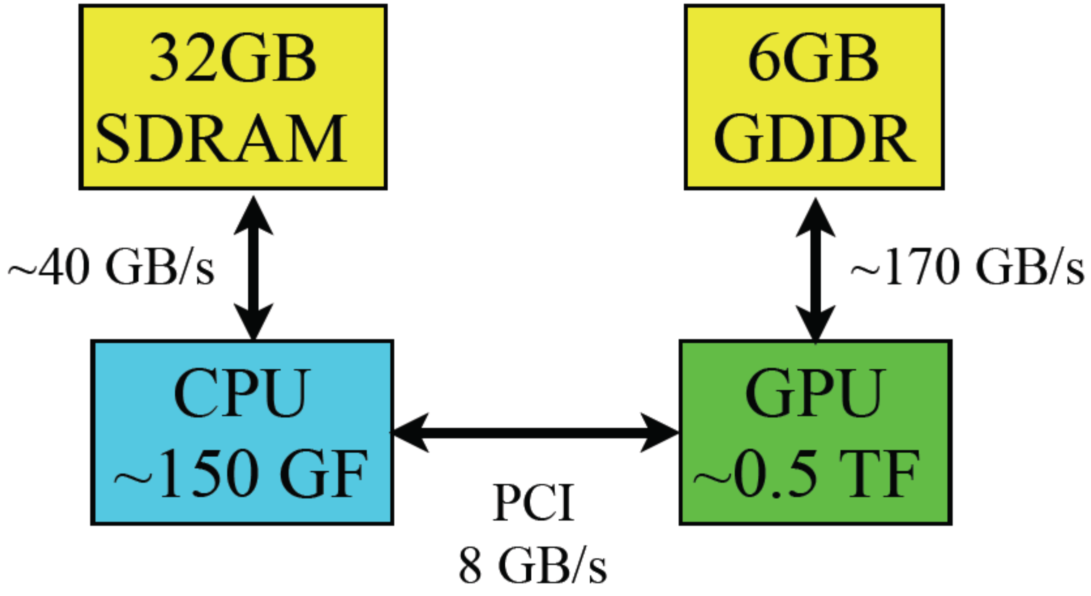
D ted Computing - Illustration

Mythbusters:

- Adam
- Jamie

Vivid presentation on GPU-principle at NVIDIA conference (2008)

Different Memory Bandwidths!



The connection between CPU and GPU has low bandwidth:

- need to minimize data transfers
- important to use asynchronous transfers, if it is possible (overlap computation and transfer)

Why "Performance/Watt" is important?

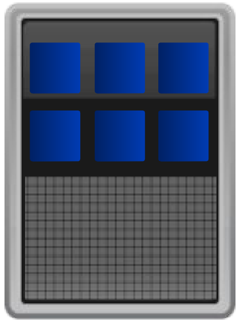
Cluster:
2.3 PFlops

7.0
Megawatts

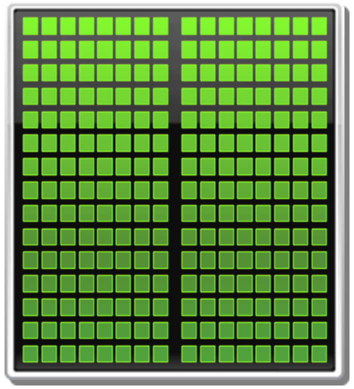
It is equal to:
7000 homes

7.0
Megawatts

CPU
Optimized for Serial Tasks



GPU Accelerator
Optimized for Many Parallel Tasks



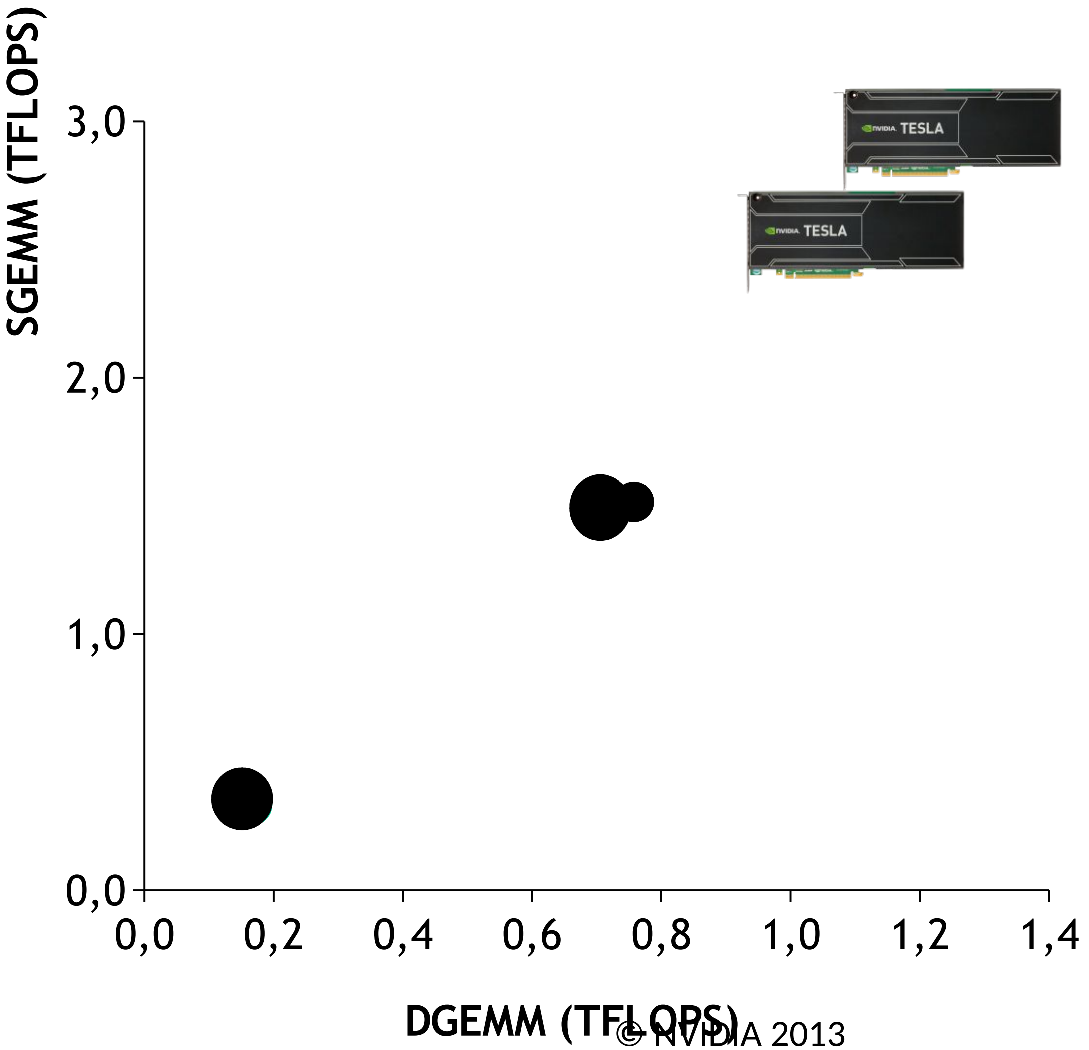
10x performance/socket

> 5x energy efficiency

Traditional CPUs are not economically feasible

Era of GPU-accelerated computing is here

World's Fastest, Most Energy Efficient Accelerator (GEMM - general matrix multiplication tests)



Tesla K20X vs Xeon CPU

8x Faster SGEMM

6x Faster DGEMM

Tesla K20X vs Xeon Phi

90% Faster SGEMM

60% Faster DGEMM

GPU – programming?

APIs for GPUs

- **CUDA** is supported by NVIDIA

- **OpenCL** is supported by AMD (and NVIDIA)
  - more recent and less developed alternative to CUDA
  - a **vendor-agnostic** computing platform
  - **supports vendor-specific extensions** like in OpenGL
  - goal is to support a **range of hardware architectures** including GPUs, CPUs, Cell processors, Larrabee and DSPs using a standard low-level API

  - **OpenACC compiler directive approach is emerging as an alternative** (works somewhat like OpenMP)

Motives for GPU programming

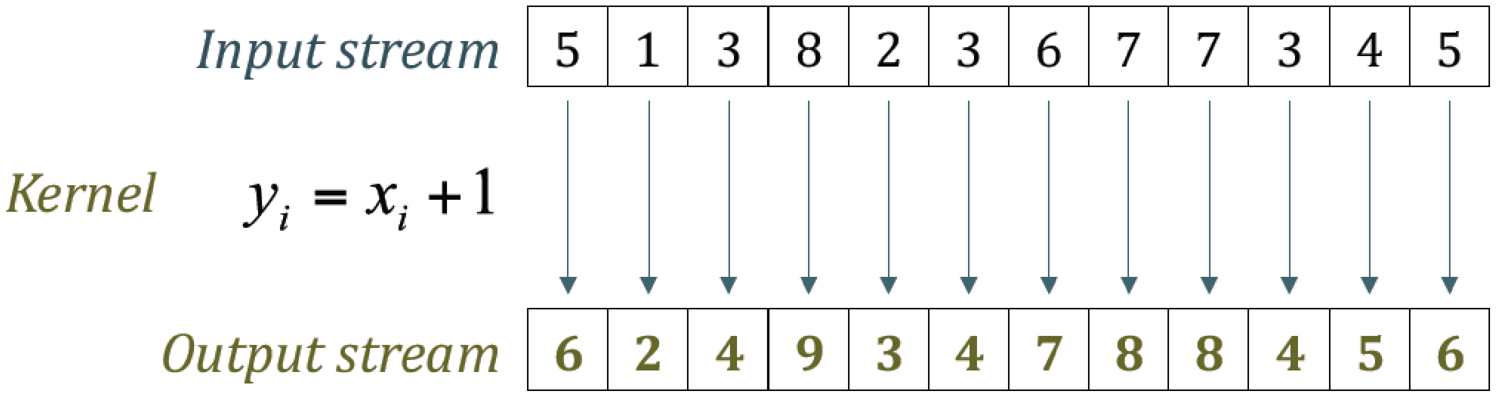
- “Supercomputing for the masses”
– significant computational horsepower at an attractive price point – readily accessible hardware

- Scalability
– programs can execute without modification on a run-of-the-mill PC with a $150 graphics card or a dedicated multi-card supercomputer worth thousands of dollars

- Bright future – the computational capability of GPUs doubles each year
– more thread processors, faster clocks, faster DRAM, …
– “GPUs are getting faster, faster”

Standard parallel programming

• Stream computing – a parallel processing model where a computational ***kernel*** is applied to a set of data (a ***stream)*** – the kernel is applied to stream elements in parallel
• GPUs excel at this thanks to a large number of processing units and a parallel architecture



GPUs offer functionality that goes beyond mere stream computing:
• Shared memory and thread synchronization primitives eliminate the need for data independence
• Gather and scatter operations allow kernels to read and write data at arbitrary locations

GPU programming – CUDA

What is CUDA?

"Compute Unified Device Architecture"

- A platform that exposes NVIDIA GPUs as general purpose *compute devices*
- Is CUDA considered GPGPU? – yes and no
- CUDA can execute on devices with no graphics output capabilities (the NVIDIA Tesla product line) – these are not "GPUs", per se
- however, if you are using CUDA to run some generic algorithms on your graphics card, you are indeed performing some ***General Purpose*** computation on your ***Graphics Processing Unit…***

What is CUDA?

CUDA Architecture

- Provide GPU parallelism for general-purpose computing
- Propose potentially high performance

CUDA C/C++

- Based on industry-standard C/C++
- Small set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

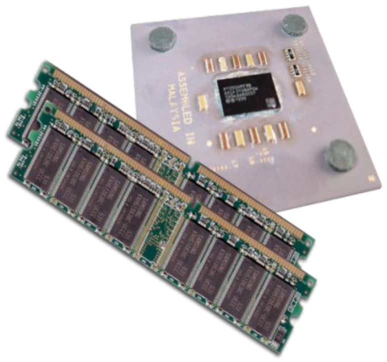
CUDA basics

Concepts

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Heterogeneous Computing

- Terminology:
  - **Host** The **C**PU and its memory (**host** memory)
  - **Device** The **G**PU and its memory (**device** memory)



Host

Device

Memory Management

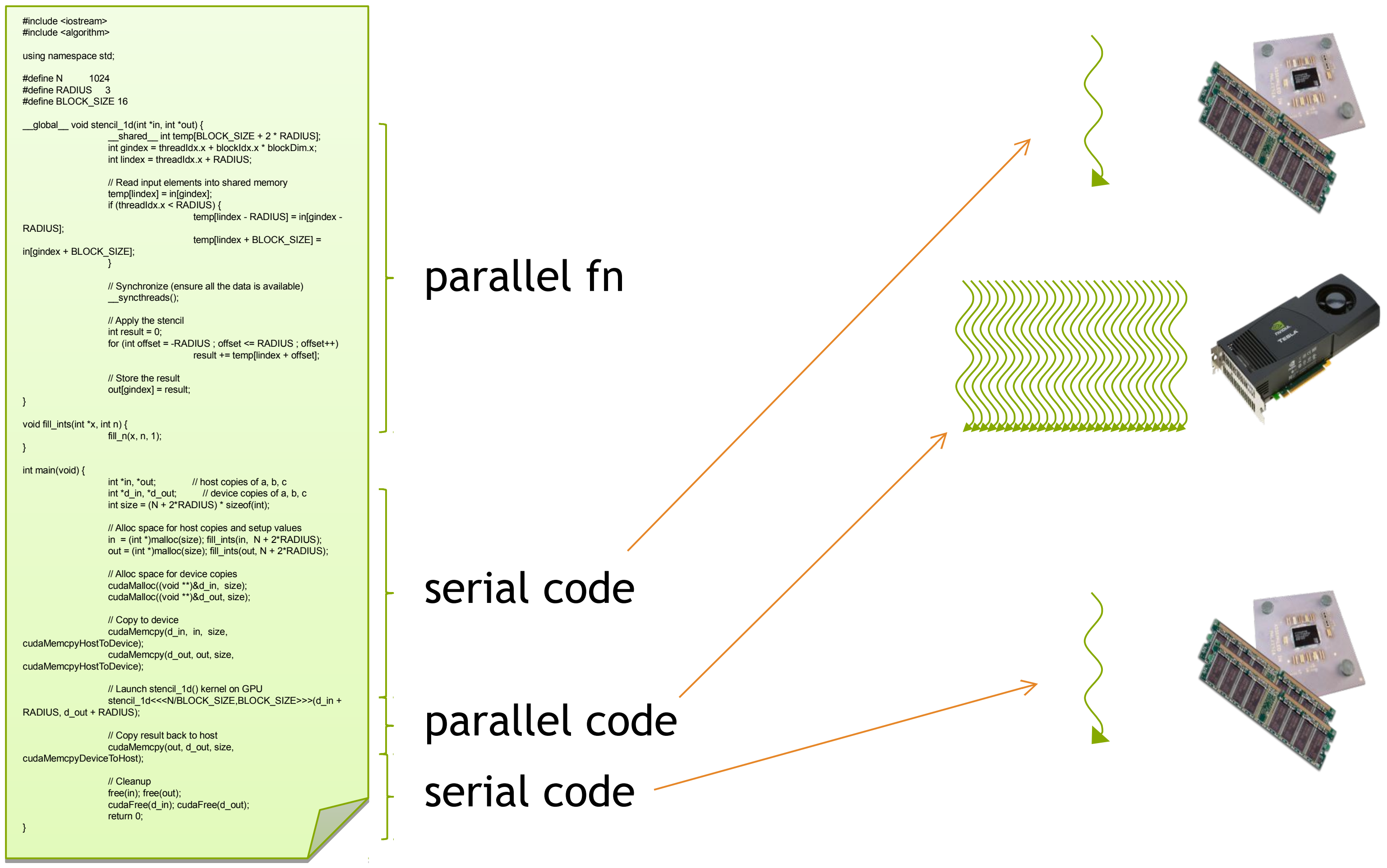
Host and device memory are separate entities

- **Device** pointers point to GPU memory
  - May be passed to/from host code
  - May not be dereferenced in host code
- **Host** pointers point to CPU memory
  - May be passed to/from device code
  - May not be dereferenced in device code

Simple CUDA API for handling device memory

- `cudaMalloc(), cudaFree(), cudaMemcpy()`
- Similar to the C equivalents: `malloc(), free(), memcpy()`

Heterogeneous Computing

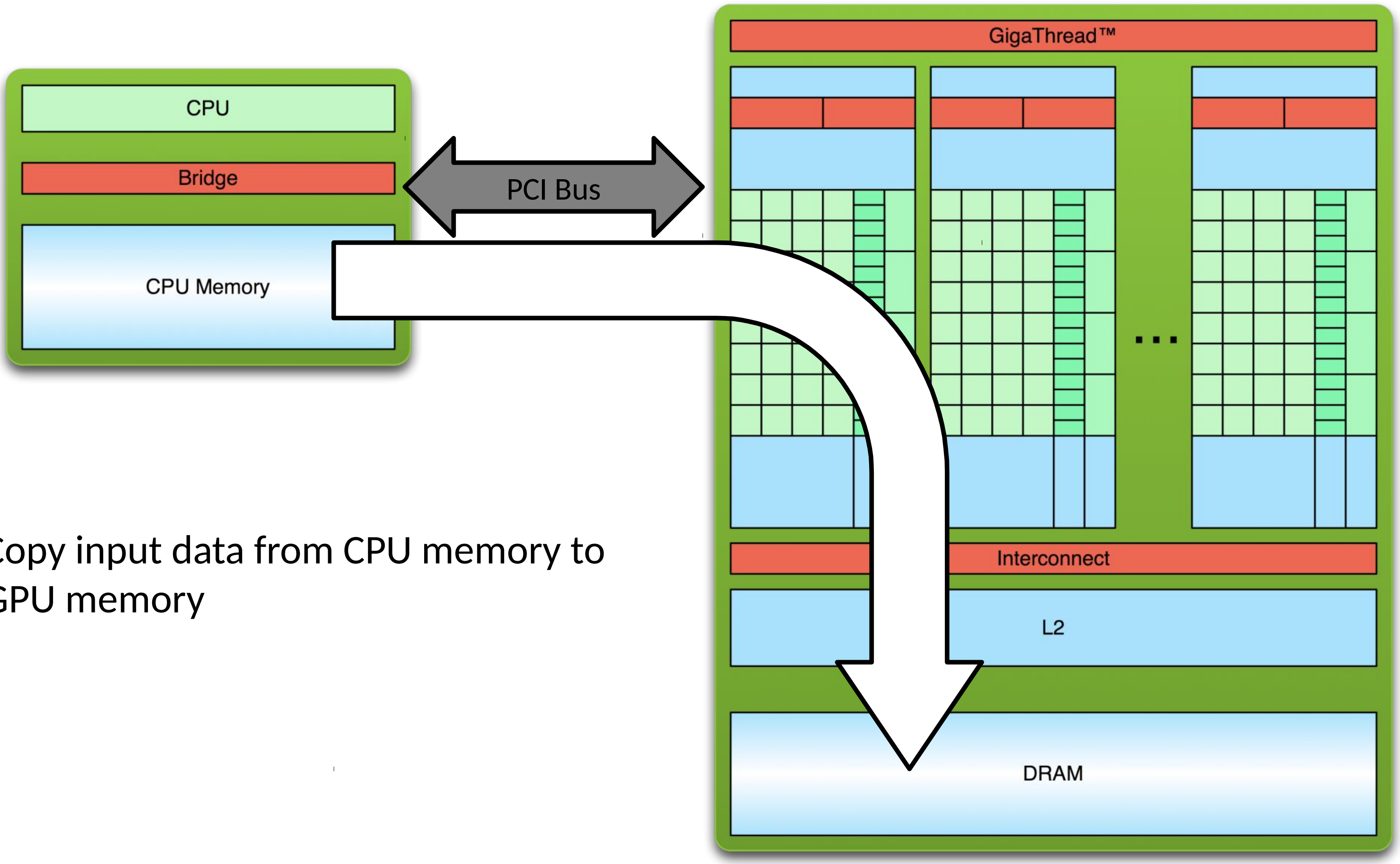


```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
                __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
                int gindex = threadIdx.x + blockIdx.x * blockDim.x;
                int lindex = threadIdx.x + RADIUS;

                // Read input elements into shared memory
                temp[lindex] = in[gindex];
                if (threadIdx.x < RADIUS) {
                                temp[lindex - RADIUS] = in[gindex -
RADIUS];
                                temp[lindex + BLOCK_SIZE] =
in[gindex + BLOCK_SIZE];
                }

                // Synchronize (ensure all the data is available)
                __syncthreads();

                // Apply the stencil
                int result = 0;
                for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                                result += temp[lindex + offset];

                // Store the result
                out[gindex] = result;
}

void fill_ints(int *x, int n) {
                fill_n(x, n, 1);
}

int main(void) {
                int *in, *out;          // host copies of a, b, c
                int *d_in, *d_out;      // device copies of a, b, c
                int size = (N + 2*RADIUS) * sizeof(int);

                // Alloc space for host copies and setup values
                in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
                out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

                // Alloc space for device copies
                cudaMalloc((void **)&d_in,  size);
                cudaMalloc((void **)&d_out, size);

                // Copy to device
                cudaMemcpy(d_in,  in,  size,
cudaMemcpyHostToDevice);
                cudaMemcpy(d_out, out, size,
cudaMemcpyHostToDevice);

                // Launch stencil_1d() kernel on GPU
                stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in +
RADIUS, d_out + RADIUS);

                // Copy result back to host
                cudaMemcpy(out, d_out, size,
cudaMemcpyDeviceToHost);

                // Cleanup
                free(in); free(out);
                cudaFree(d_in); cudaFree(d_out);
                return 0;
}
```

CUDA programming model

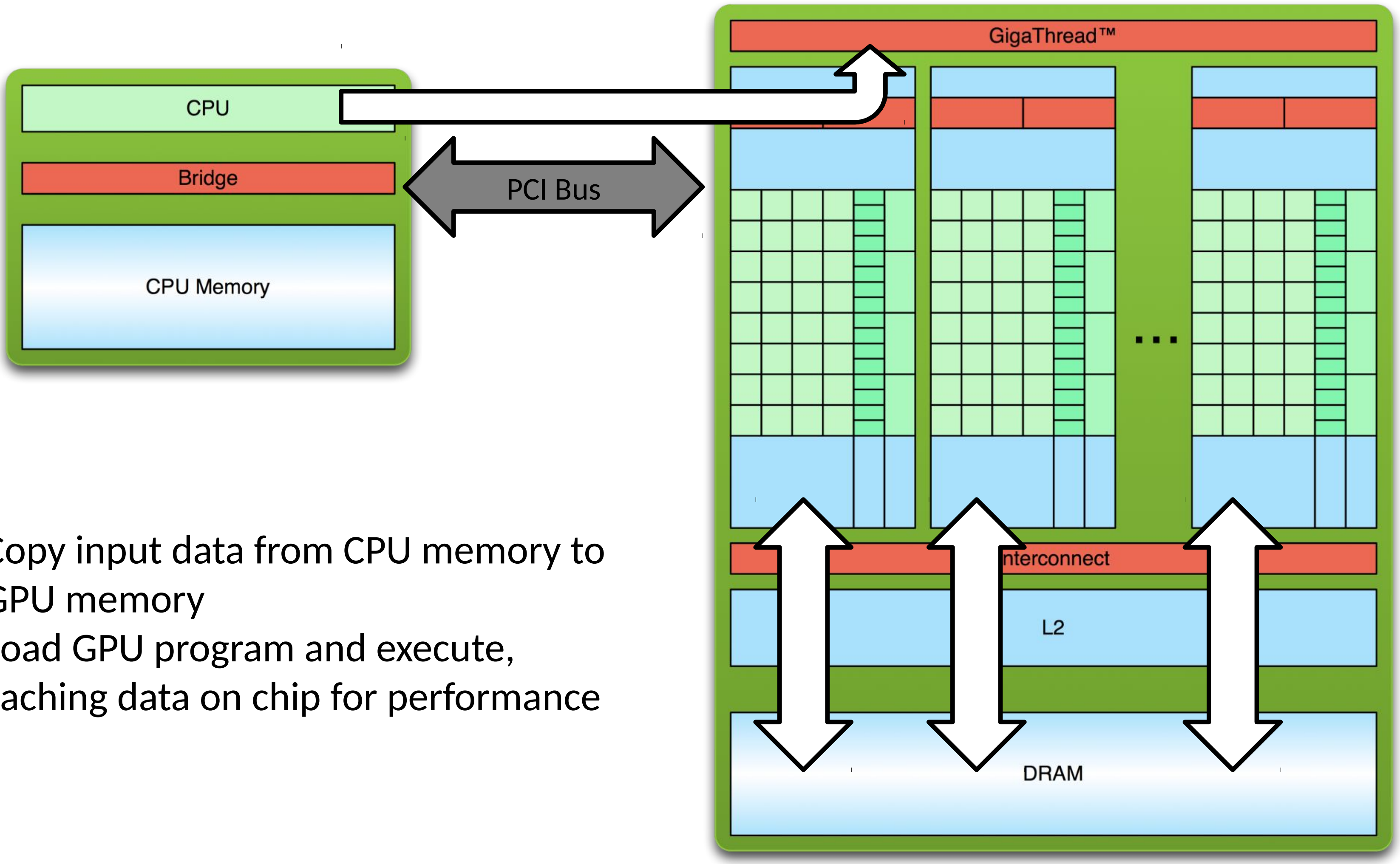
- The main CPU is referred to as the ***host***
- The compute device is viewed as a ***coprocessor capable of*** executing a large number of light threads in parallel
- Computation on the device is performed by ***kernels, functions*** executed in parallel on each data element
- Both the host and the device have their own ***memory:*** host and device cannot directly access each other’s memory, but data can be transferred
- The host manages all memory allocations on the device, data transfers, and the invocation of kernels on the device

Simple Processing Flow



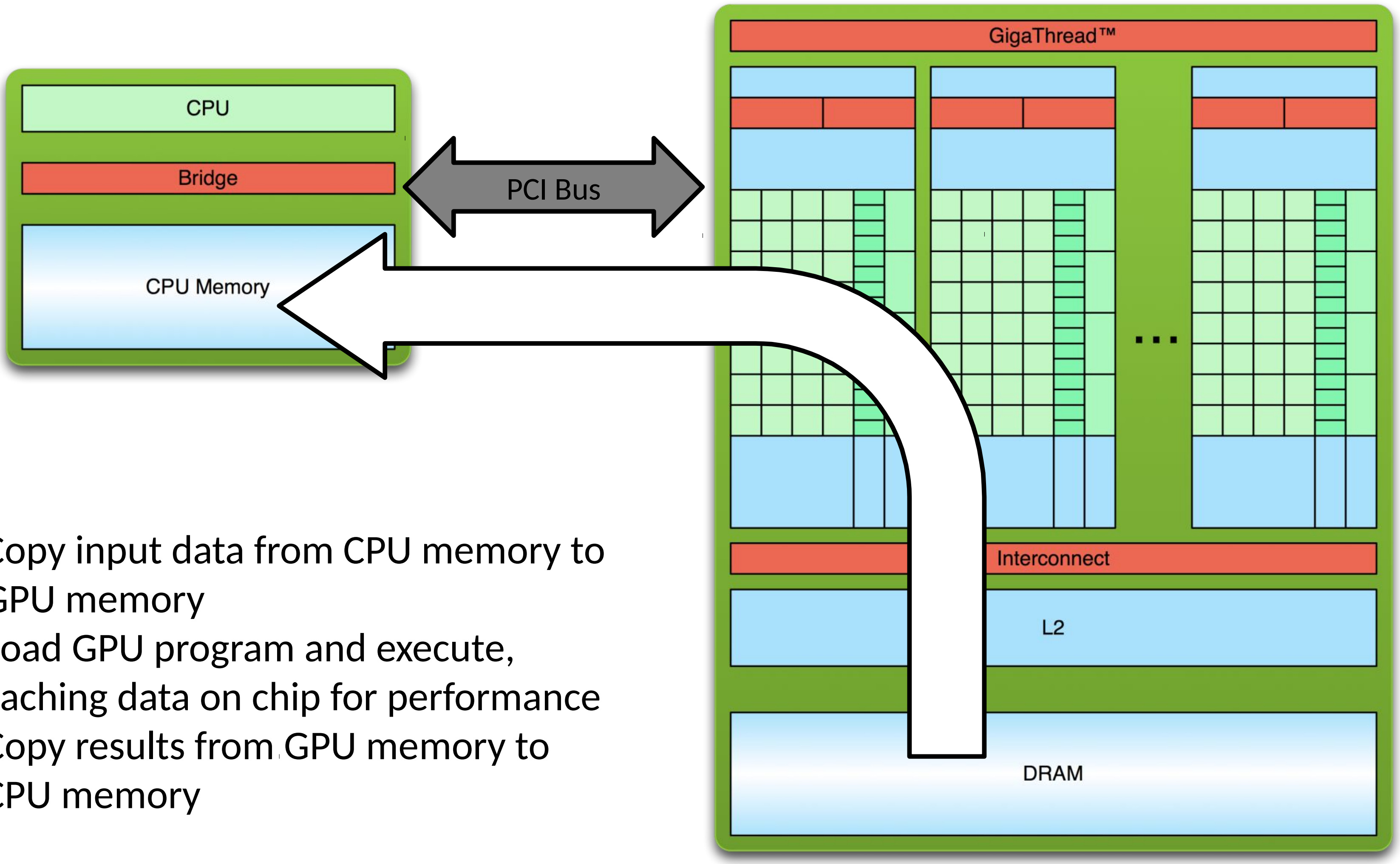
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Language and compiler

CUDA provides a set of extensions to the C programming language – new storage quantifiers, kernel invocation syntax, intrinsics, vector types, etc.

- CUDA source code saved in .cu files
  - host and device code and coexist in the same file
  - storage qualifiers determine type of code

- Compiled to object files using nvcc compiler
  - object files contain executable host and device code

- Can be linked with object files generated by other C/C++ compilers

GPU programming – CUDA – trivial example (“Hello World”)

Hello World!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc
hello_world.
cu
$ a.out
Hello World!
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- Two new syntactic elements…

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

CUDA C/C++ keyword `__global__` indicates a function that:

- runs on the device
- is called from host code

`nvcc` separates source code into host and device components

- device functions (e.g. `mykernel()`) processed by NVIDIA compiler
- host functions (e.g. `main()`) processed by standard host compiler
  - `gcc`, `cl.exe`

Hello World! with Device COde

```
mykernel<<<1,1>>>();
```

Triple angle brackets mark a call from *host* code to *device* code

- also called a “kernel launch”
- we’ll return to the parameters (1,1) in a moment

That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

```
__global__ void mykernel(void){
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

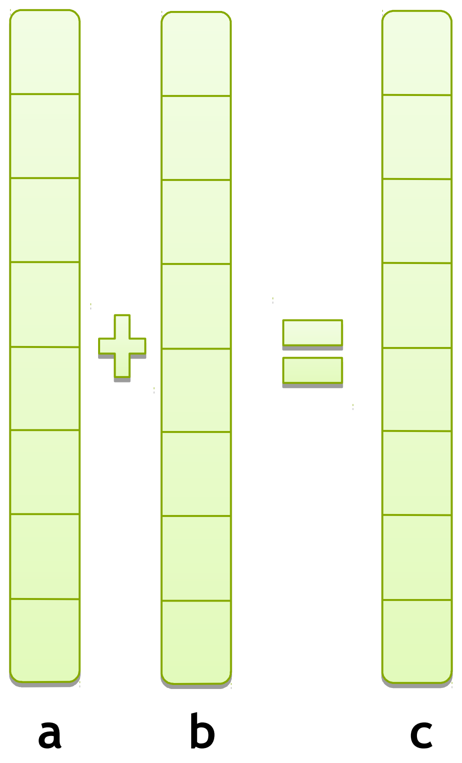
Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

- **`mykernel()`** does nothing, somewhat disappointing!

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We'll start by adding two integers and build up to vector addition



GPU programming – CUDA – simple example (addition)

Addition on the Device

A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
   *c = *a + *b;
}
```

As before `__global__` is a CUDA C/C++ keyword meaning

- `add()` will execute on the device
- `add()` will be called from the host

Addition on the Device

Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

add() runs on the device, so **a**, **b** and **c** must point to device memory

We need to allocate memory on the GPU

Addition on the Device: `add()`

Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

Let’s take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
    int a, b, c;                    // host copies of a, b, c
    int *d_a, *d_b, *d_c;               // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

CUDA basics

Blocks

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
                ↓
add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

With `add()` running in parallel we can do vector addition

Terminology: each parallel invocation of `add()` is referred to as a block

- The set of blocks is referred to as a grid
- Each invocation can refer to its block index using `blockIdx.x`

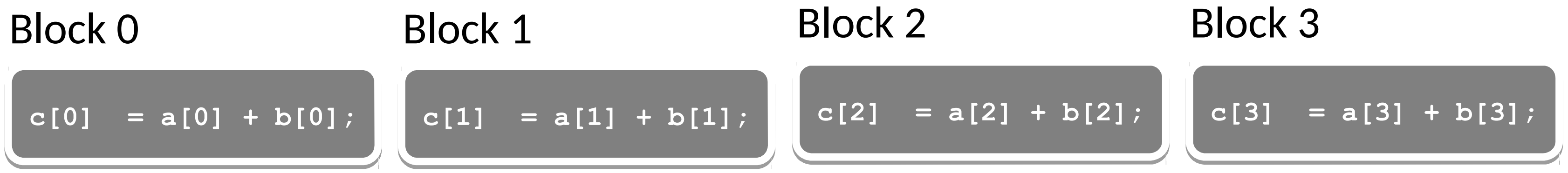
```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

On the device, each block can execute in parallel:



```
c[0]   = a[0] + b[0];
c[1]   = a[1] + b[1];
c[2]   = a[2] + b[2];
c[3]   = a[3] + b[3];
```

Vector Addition on the Device: `add()`

Returning to our parallelized **`add()`** kernel

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Let’s take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
int *a  *b  *c          // host copies of a, b, c
int *d_a, *d_b, *d_c; // device copies of a, b, c
int size = N * sizeof(int);

// Alloc space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

// Alloc space for host copies of a, b, c and setup input values
a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Threads

CUDA basics

CUDA Threads

Terminology: a block can be split into parallel threads

Let’s change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

We use `threadIdx.x` instead of `blockIdx.x`

Need to make one change in `main()`…

Vector Addition Using Threads: `main()`

```
#define N 512
int main(void) {
    int *a, *b, *c;            // host copies of a, b, c
    int *d_a, *d_b, *d_c;      // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: `main()`

```
     // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Indexing

CUDA basics

Combining Blocks and Threads

We’ve seen parallel vector addition using:

- Many blocks with one thread each
- One block with many threads

Let’s adapt vector addition to use both blocks and threads

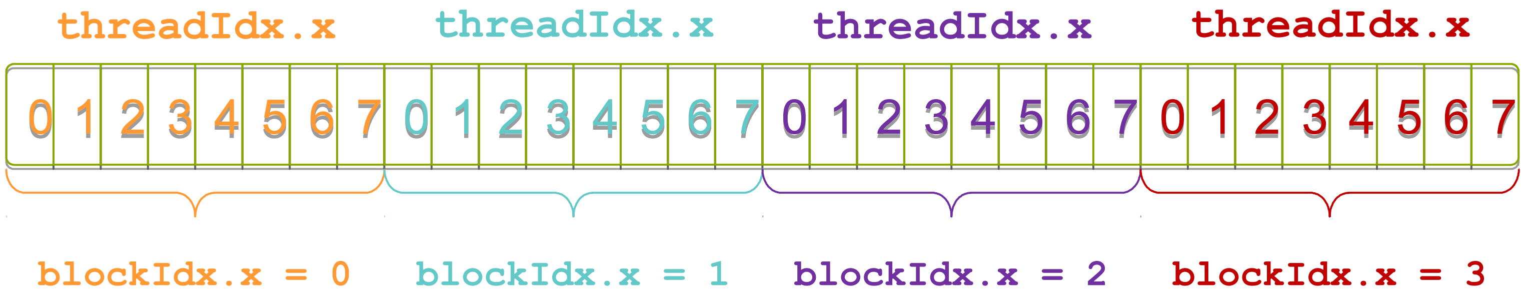
Why? We’ll come to that…

First let’s discuss data indexing…

Indexing Arrays with Blocks and Threads

No longer as simple as using `blockIdx.x` and `threadIdx.x`

- Consider indexing an array with one element per thread (8 threads/block)

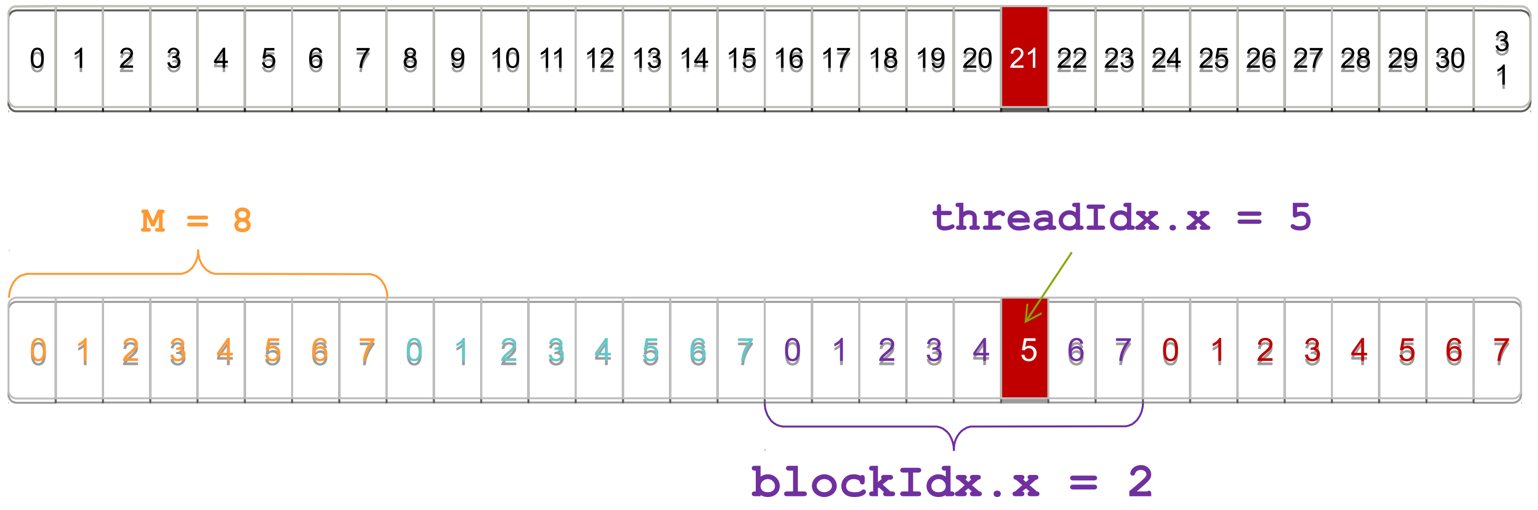


With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +     2       * 8;
          = 21;
```

Vector Addition with Blocks and Threads

Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

What changes need to be made in `main()`?

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;              // host copies of a, b, c
    int *d_a, *d_b, *d_c;        // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

Typical problems are not friendly multiples of `blockDim.x`

Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

CUDA basics

Memory

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

__syncthreads()

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent some hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Review (1 of 2)

Launching parallel threads

- Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
- Use `blockIdx.x` to access block index within grid
- Use `threadIdx.x` to access thread index within block

Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

CUDA basics

Management

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

Handling Errors

All CUDA API calls return an error code (`cudaError_t`)

- Error in the API call itself

  OR

- Error in an earlier asynchronous operation (e.g. kernel)

Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Managing Devices

Application can query and select GPUs

- `cudaGetDeviceCount(int *count)`
- `cudaSetDevice(int device)`
- `cudaGetDevice(int *device)`
- `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`

Multiple threads can share a device

A single thread can manage multiple devices

- `cudaSetDevice(i)` to select current device
- `cudaMemcpy(…)` for peer-to-peer copies†

Resume

What have we learned?

- Write and launch CUDA C/C++ kernels
  - `__global__, blockIdx.x, threadIdx.x, <<<>>>`
- Manage GPU memory
  - `cudaMalloc(), cudaMemcpy(), cudaFree()`
- Manage communication and synchronization
  - `__shared__, __syncthreads()`
  - `cudaMemcpy()` VS `cudaMemcpyAsync(), cudaDeviceSynchronize()`

Compute Capability

The **compute capability** of a device describes its architecture, e.g.

Number of registers

Sizes of memories

Features & capabilities

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	GPU models
1.0	Fundamental CUDA support	Tesla C870
1.3	Double precision, improved memory accesses, atomics	Tesla 10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	Tesla 20-series
2.1	**-**	**GTX 560**
3.5	**Warp shuffle functions, funnel shift, dynamic parallelism**	**Tesla K40**

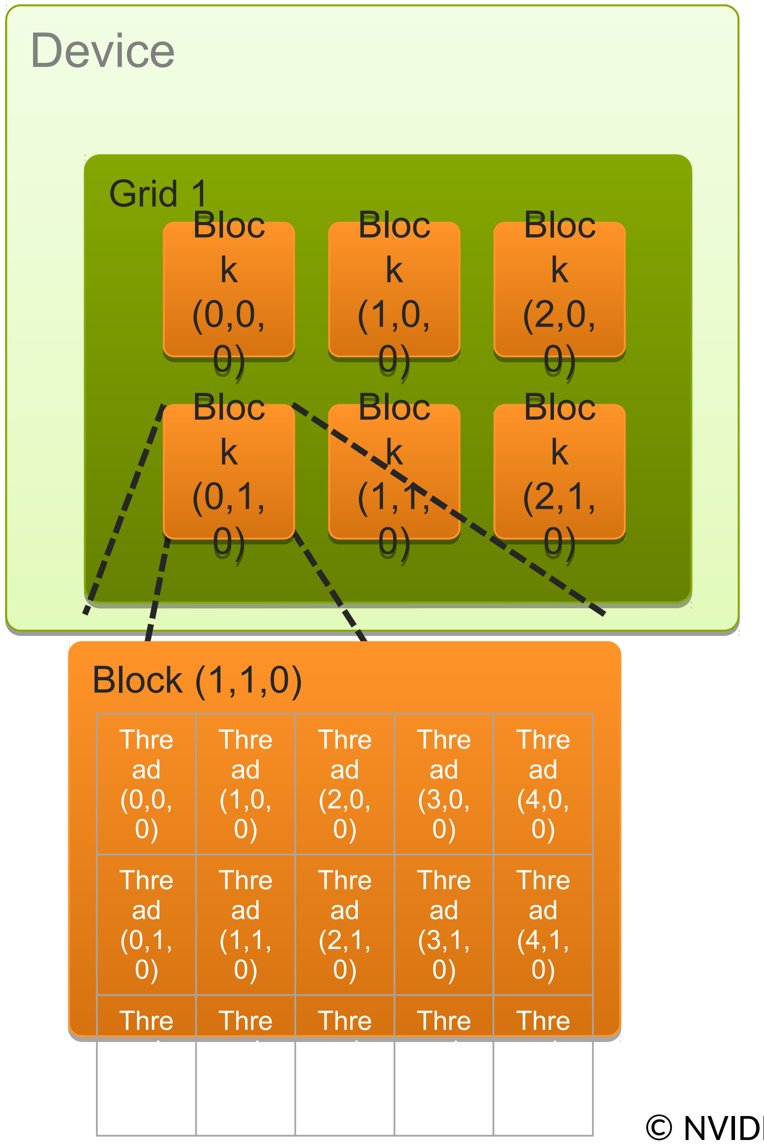
IDs and Dimensions

A kernel is launched as a grid of blocks of threads

- `blockIdx` and `threadIdx` are 3D
- We showed only one dimension (`x`)

Built-in variables:

```
threadIdx
blockIdx
blockDim
gridDim
```



Textures

- Read-only object

  Dedicated cache

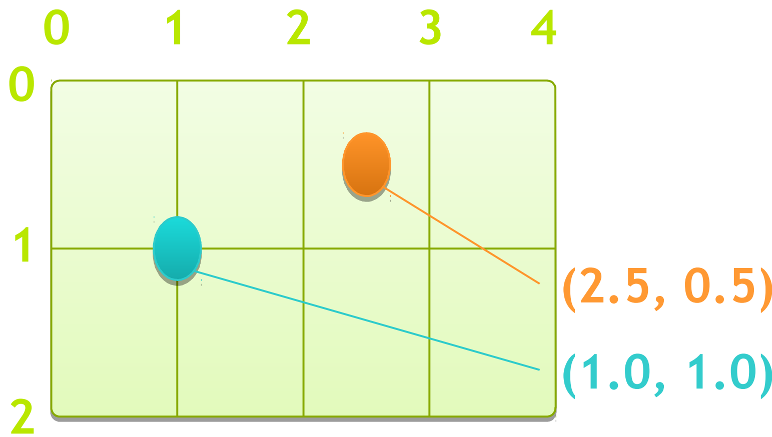
- Dedicated filtering hardware

  (Linear, bilinear, trilinear)

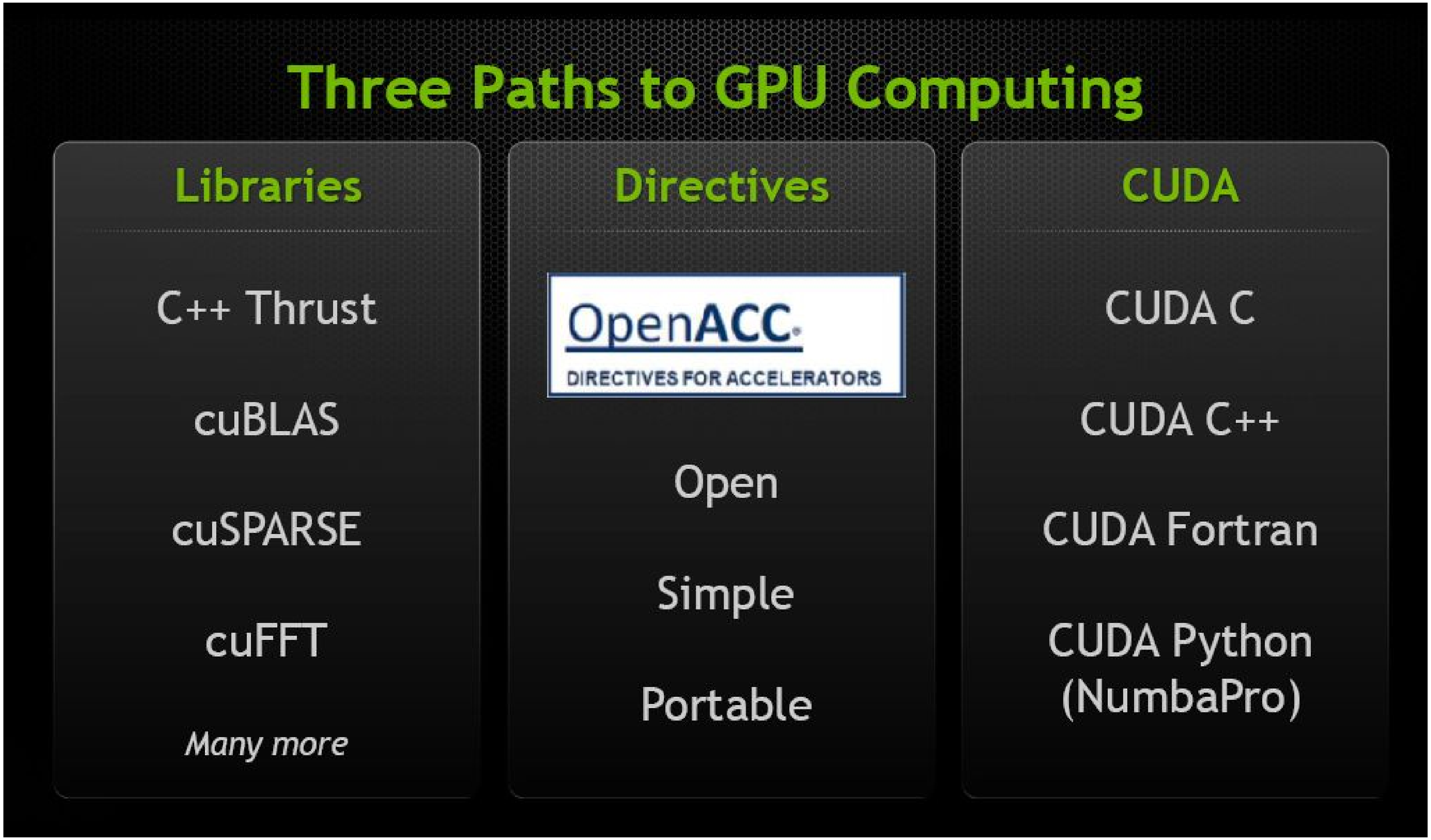
- Addressable as 1D, 2D or 3D

- Out-of-bounds address handling

  (Wrap, clamp)



Current trends in GPU programming



Three Paths to GPU Computing

Libraries	Directives	CUDA
C++ Thrust	OpenACC DIRECTIVES FOR ACCELERATORS	CUDA C
cuBLAS	Open	CUDA C++
cuSPARSE	Simple	CUDA Fortran
cuFFT	Portable	CUDA Python (NumbaPro)
Many more		

Contacts

Any course-related information
(notifications, reports) from you:

send your message to my e-mail
yuri.gordienko@gmail.com
with the word **GPU2021** in the “Subject” field
(if not, your message will be filtered out to Spam).