Really Fast Introduction to CUDA and CUDA C

Jul 2013

Dale Southard, NVIDIA

dsouthard@nvidia.com

About the Speaker

[Dale] is a senior solution architect with NVIDIA (I fix things). I primarily cover HPC in Gov/Edu/Research and cloud computing. In the past I was a HW architect in the LLNL systems group designing the vis/post-processing solutions.

GeForce

Quadro

Tegra

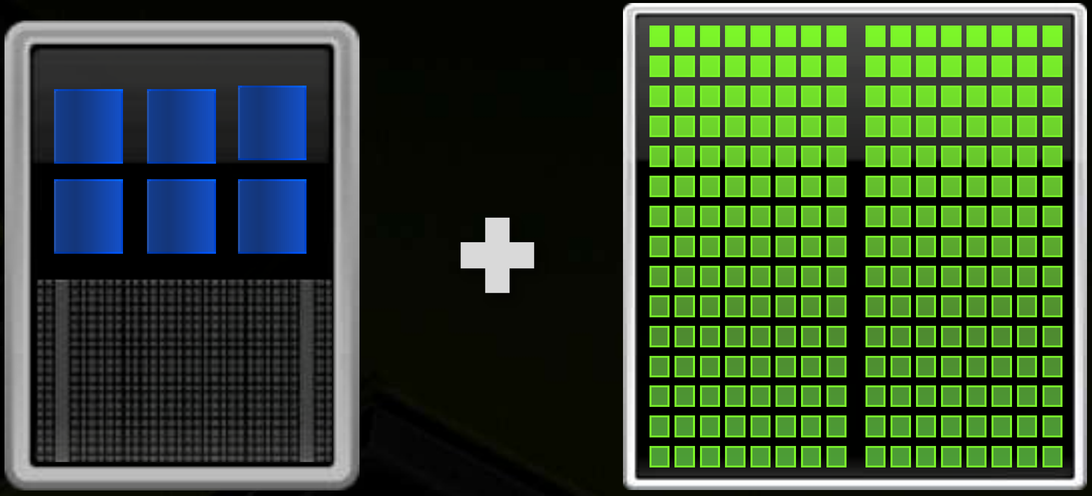
Tesla

The World Leader in Parallel Processing

Why CUDA

CUDA Accelerates Computing

Choose the right processor for the right task



CPU

Several sequential cores

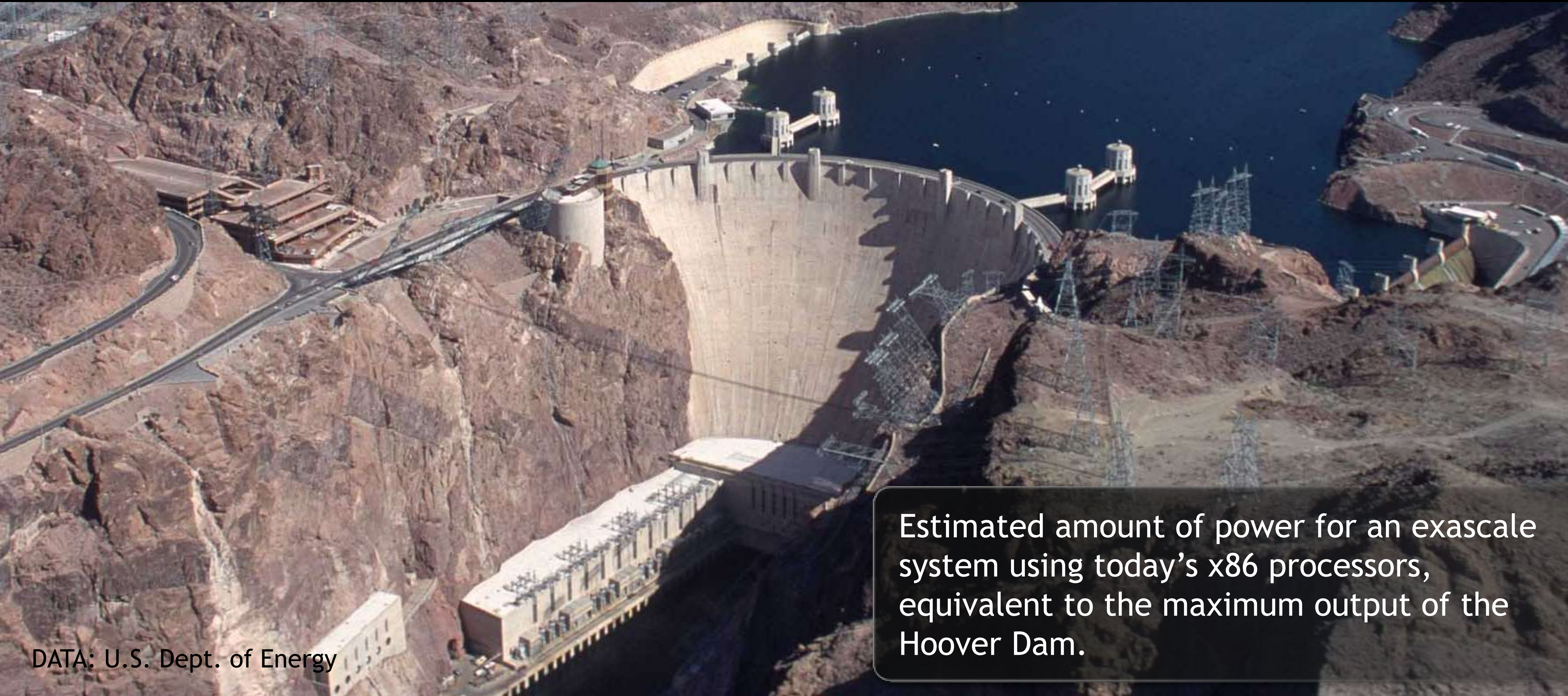
CUDA GPU

Thousands of parallel cores

Power is THE problem

*Underlying changes in silicon technology scaling mean the "free ride" of the past several decades is over. Power will be the **dominant** constraint in computing.*

2 GIGAWATTS



Estimated amount of power for an exascale system using today’s x86 processors, equivalent to the maximum output of the Hoover Dam.

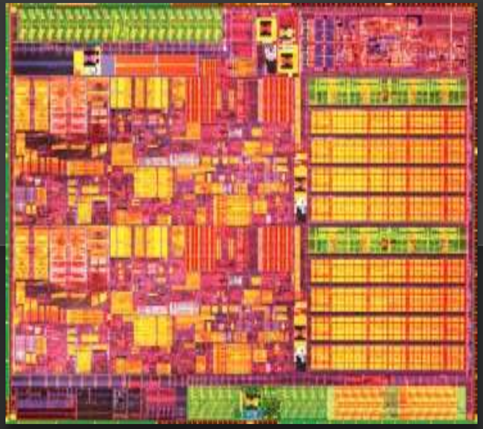
DATA: U.S. Dept. of Energy

CPU

>1000 pJ/Instruction

Optimized for Latency

Caches

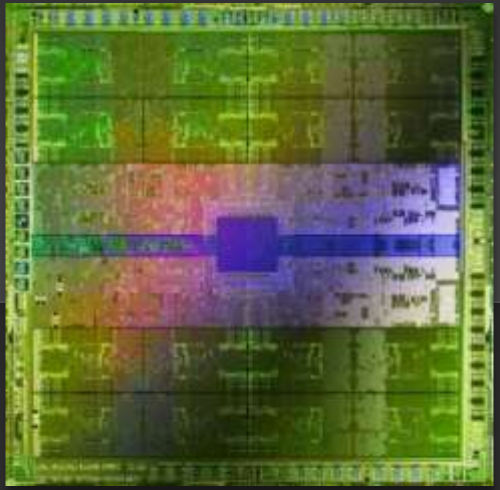


GPU

<200 pJ/Instruction

Optimized for Throughput

Explicit Management
of On-chip Memory



The World's Most Energy Efficient Supercomputer

3208 MFLOPS/Watt

128 Tesla K20 Accelerators

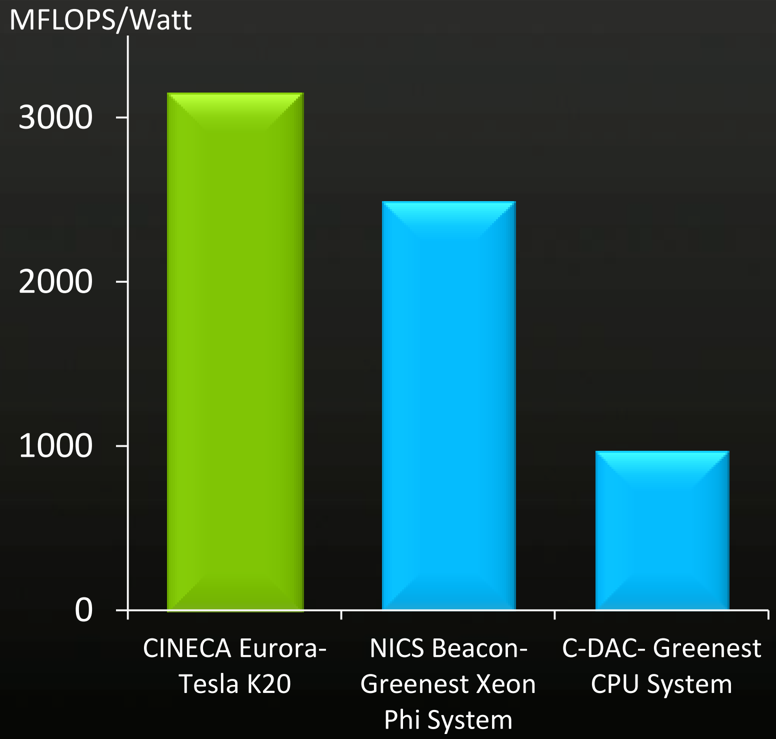
$100k Energy Savings / Yr

300 Tons of CO_2 Saved / Yr

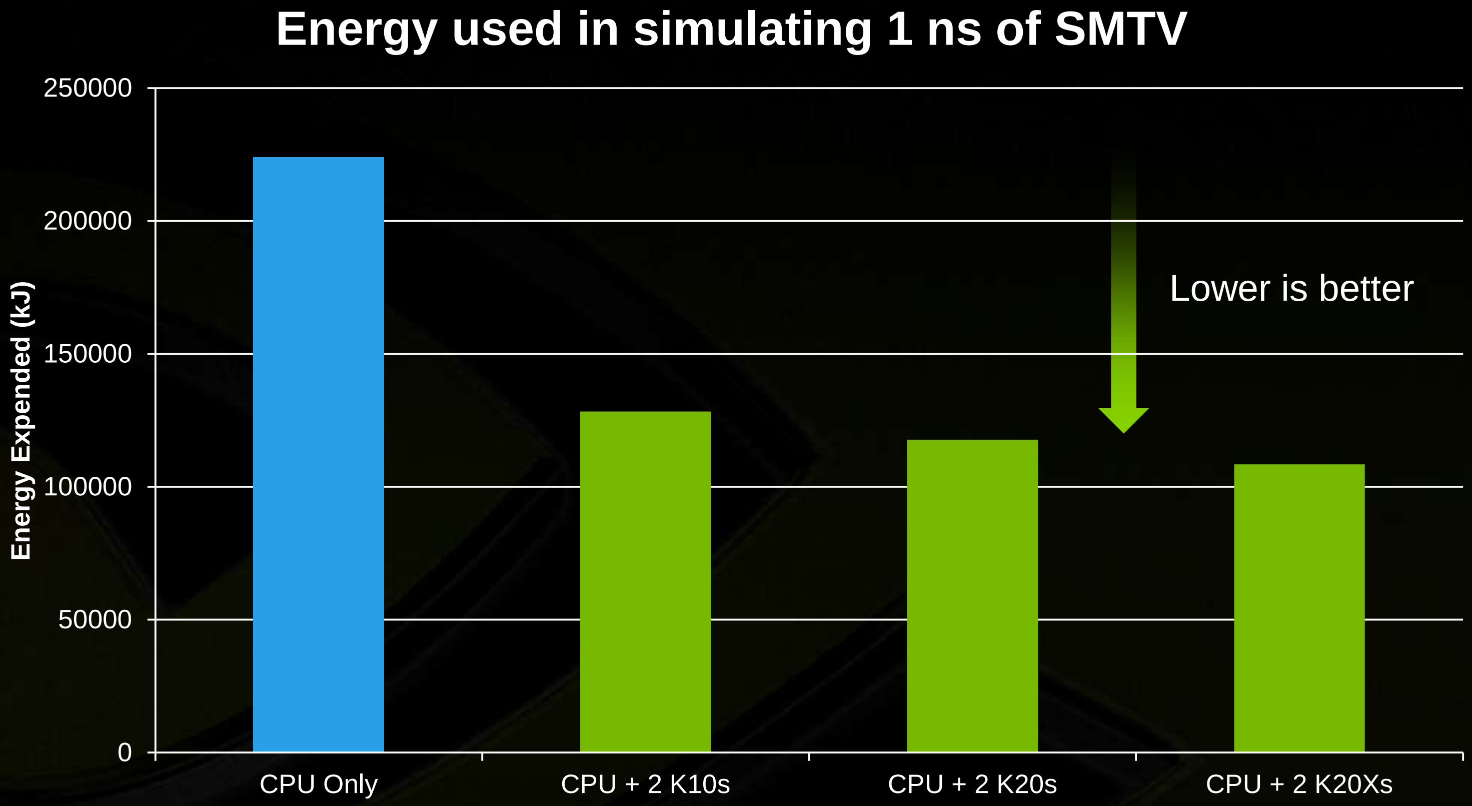
CINECA Eurora

"Liquid-Cooled" Eurotech Aurora Tigon

Greener than Xeon Phi, Xeon CPU



Kepler - Greener: Twice The Science/Joule



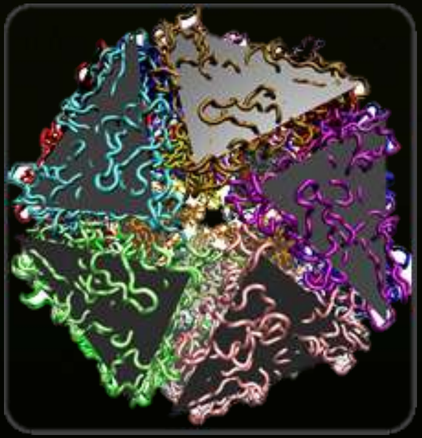
Cut down energy usage by ½ with GPUs

Running NAMD version 2.9

The blue node contains Dual E5-2687W CPUs (150W each, 8 Cores per CPU).

The green nodes contain Dual E5-2687W CPUs (8 Cores per CPU) and 2x NVIDIA K10, K20, or K20X GPUs (235W each).

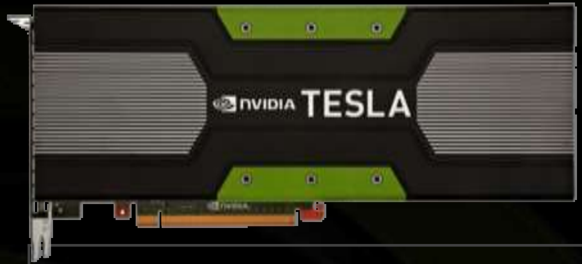
Energy Expended
= Power x Time



Satellite Tobacco Mosaic Virus

Tesla Kepler Family

World's Fastest and Most Efficient HPC Accelerators

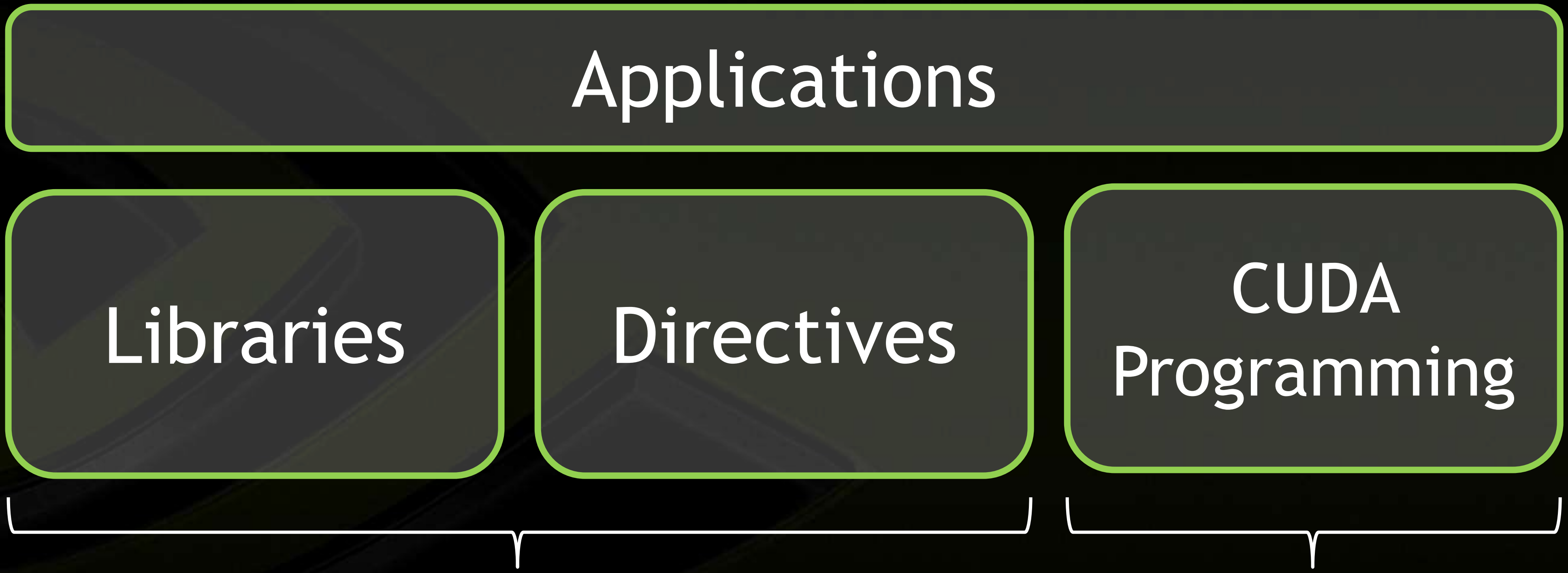
	GPUs	Single Precision Peak *(SGEMM)*	Double Precision Peak *(DGEMM)*	Memory Size	Memory Bandwidth (ECC off)	System Solution
Weather & Climate, Physics, BioChemistry, CAE, Material Science	**K20X**	3.95 TF *(2.90 TF)*	1.32 TF *(1.22 TF)*	6 GB	250 GB/s	Server only
	K20	3.52 TF *(2.61 TF)*	1.17 TF *(1.10 TF)*	5 GB	208 GB/s	Server + Workstation
Image, Signal, Video, Seismic	**K10**	4.58 TF	0.19 TF	8 GB	320 GB/s	Server only

GTX Titan: For High Performance Gaming Enthusiasts

CUDA Cores	2688
Single Precision	~4.5 Tflops
Double Precision	~1.27 Tflops
Memory Size	6GB
Memory B/W	288GB/s

What is CUDA
(six ways to saxpy)

Programming GPUs



Easiest Approach for 2x to 10x Acceleration

Maximum Performance

Single precision Alpha X Plus Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) Library

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

GPU SAXPY in multiple languages and libraries

A menagerie* of possibilities, not a tutorial

*technically, a *program chrestomathy*: http://en.wikipedia.org/wiki/Chrestomathy

1

OpenACC Compiler Directives

Parallel C Code

```
void saxpy(int n,
           float a,
           float *x,
           float *y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

Parallel Fortran Code

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$acc kernels
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$acc end kernels
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

http://developer.nvidia.com/openacc or http://openacc.org

CUBLAS Library

Serial BLAS Code

```
int N = 1<<20;


...

// Use your choice of blas library

// Perform SAXPY on 1M elements
blas_saxpy(N, 2.0, x, 1, y, 1);
```

Parallel cuBLAS Code

```
int N = 1<<20;


cublasInit();
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
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

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

cublasShutdown();
```

You can also call cuBLAS from Fortran, C++, Python, and other languages

http://developer.nvidia.com/cublas

3

CUDA C

Standard C

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

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

Parallel C

```
__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];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

http://developer.nvidia.com/cuda-toolkit

4

Thrust C++ Template Library

Serial C++ Code
with STL and Boost

```
int N = 1<<20;
std::vector<float> x(N), y(N);

...




// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
               2.0f * _1 + _2);
```

www.boost.org/libs/lambda

Parallel C++ Code

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);

...

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
                  d_y.begin(), d_y.begin(),
                  2.0f * _1 + _2);
```

http://thrust.github.com

5

CUDA Fortran

Standard Fortran

```
module mymodule contains
  subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  end subroutine saxpy
end module mymodule

program main
  use mymodule
  real :: x(2**20), y(2**20)
  x = 1.0, y = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy(2**20, 2.0, x, y)

end program main
```

Parallel Fortran

```
module mymodule contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)

end program main
```

http://developer.nvidia.com/cuda-fortran

Python

Standard Python

```
import numpy as np


def saxpy(a, x, y):
  return [a * xi + yi
          for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)





cpu_result = saxpy(2.0, x, y)
```

http://numpy.scipy.org

Copperhead: Parallel Python

```
from copperhead import *
import numpy as np

@cu
def saxpy(a, x, y):
  return [a * xi + yi
          for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

with places.gpu0:
  gpu_result = saxpy(2.0, x, y)

with places.openmp:
  cpu_result = saxpy(2.0, x, y)
```

http://copperhead.github.com

Enabling Endless Ways to SAXPY

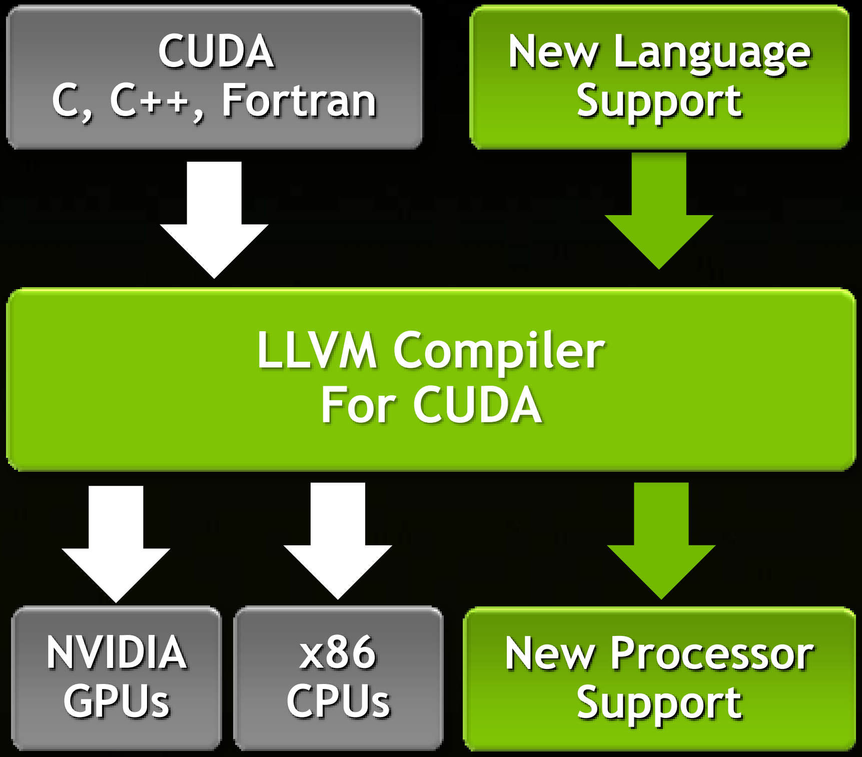
Developers want to build front-ends for

Java, Python, R, DSLs

Target other processors like

ARM, FPGA, GPUs, x86

CUDA Compiler Contributed to Open Source LLVM



CUDA Basics

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)

Host

Device

Heterogeneous Computing

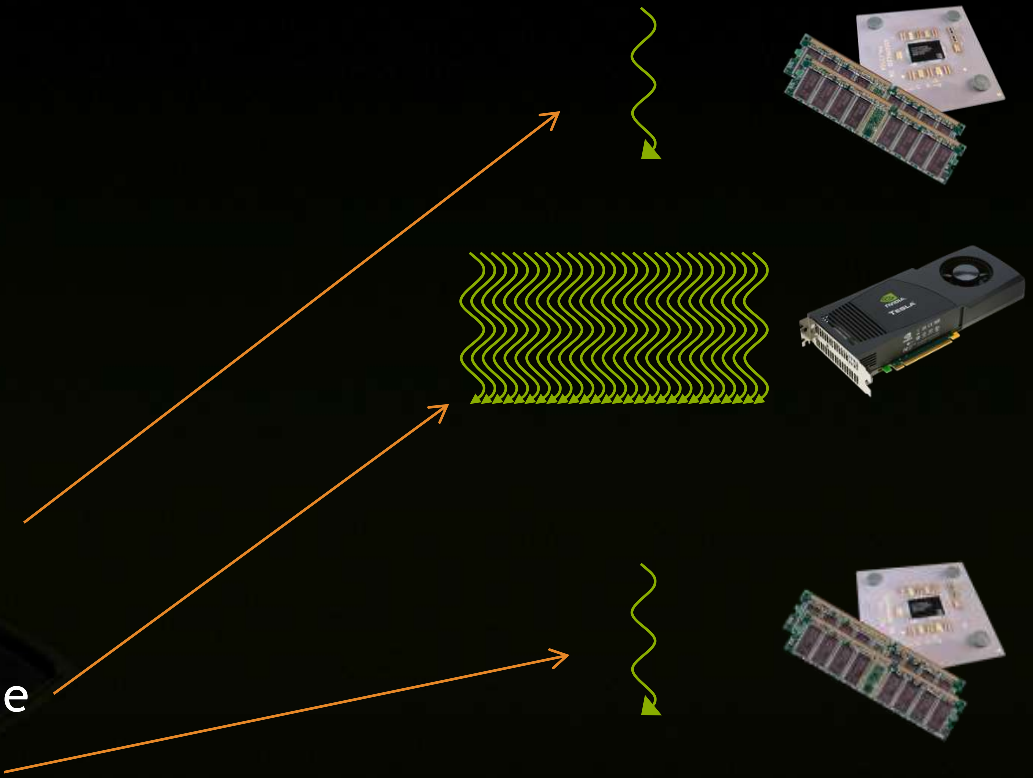
```
#include <iostream>
#include <algorithm>

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;
}
```

parallel fn

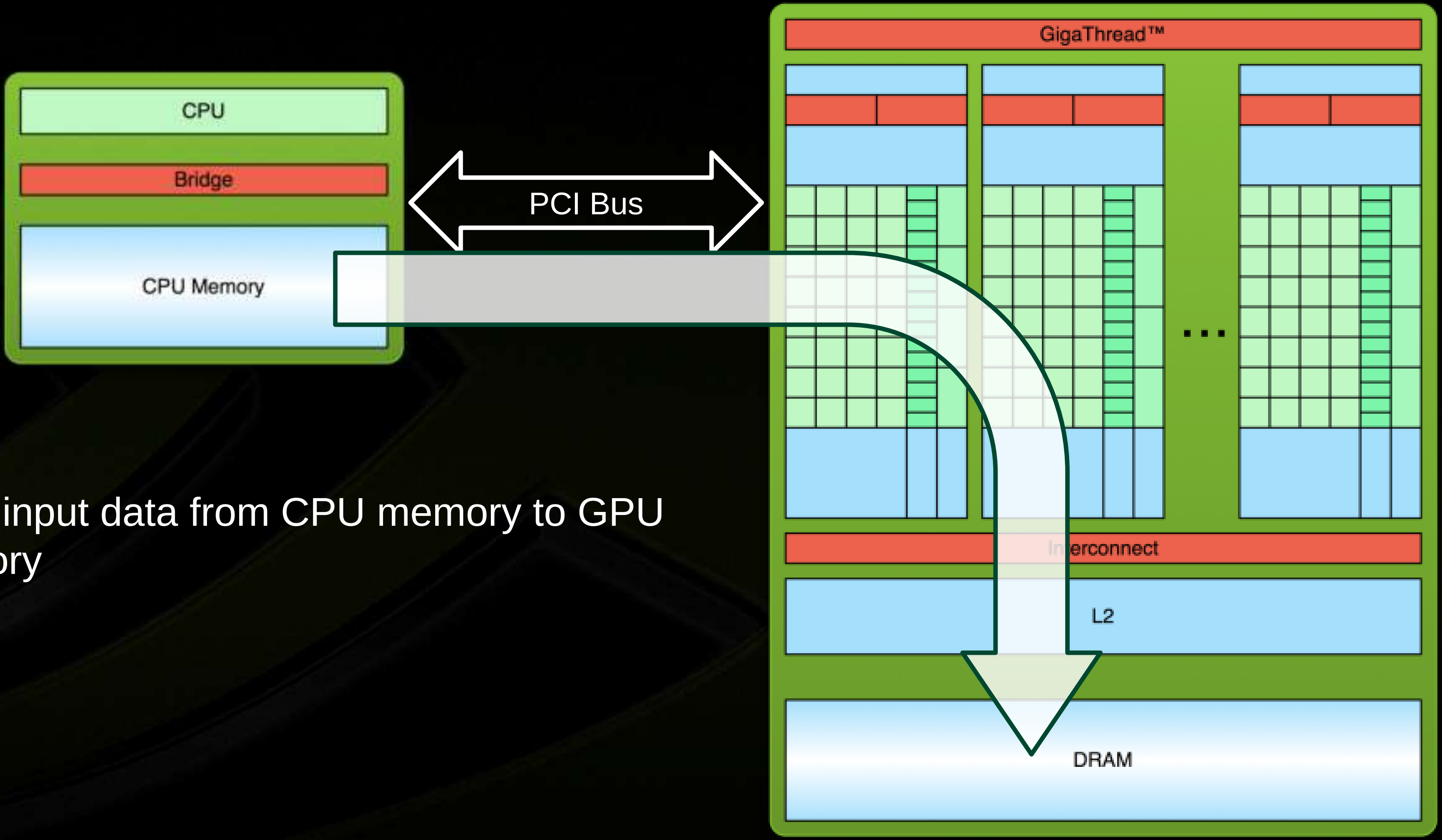
serial code

parallel code

serial code

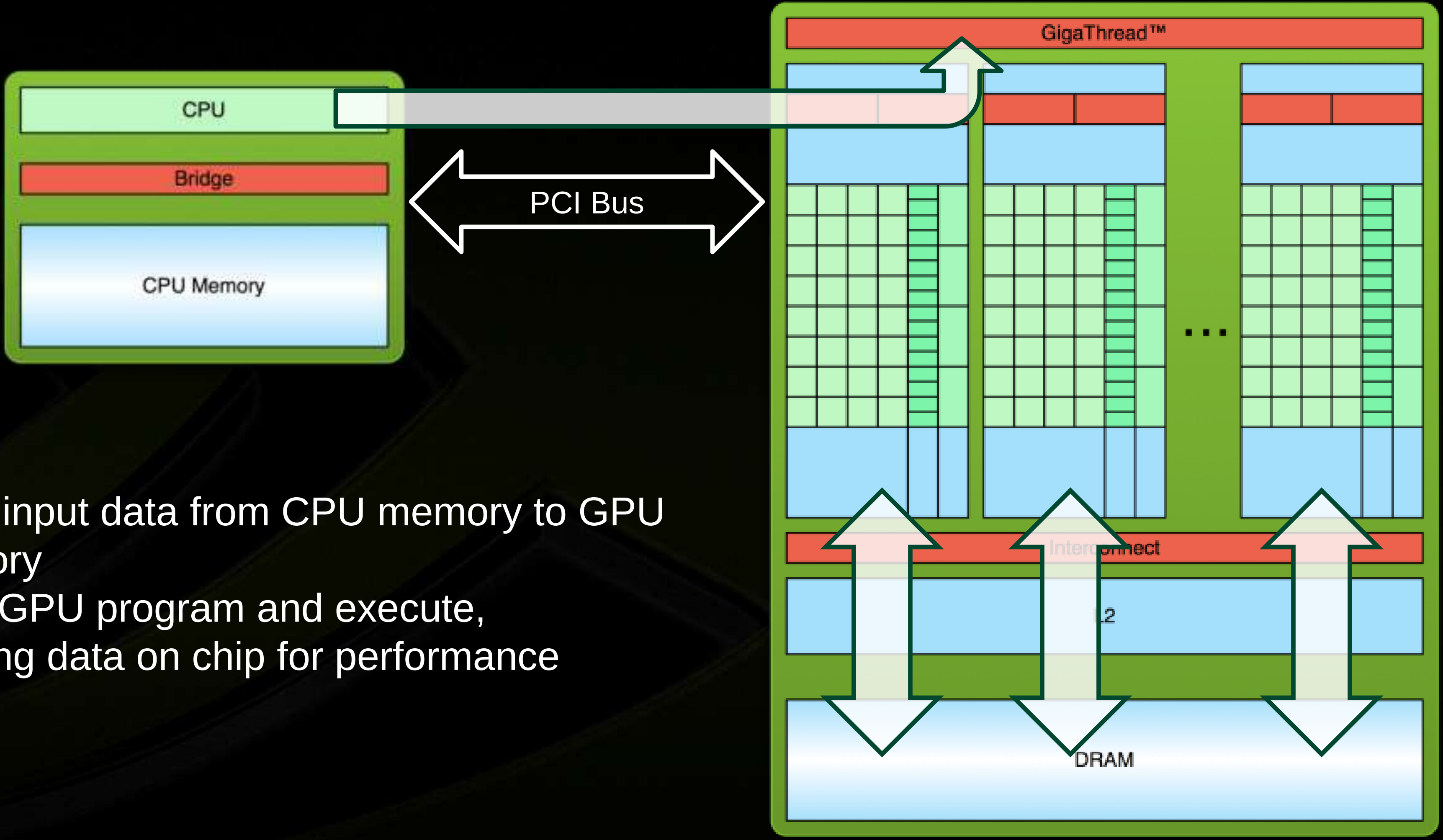


Simple Processing Flow



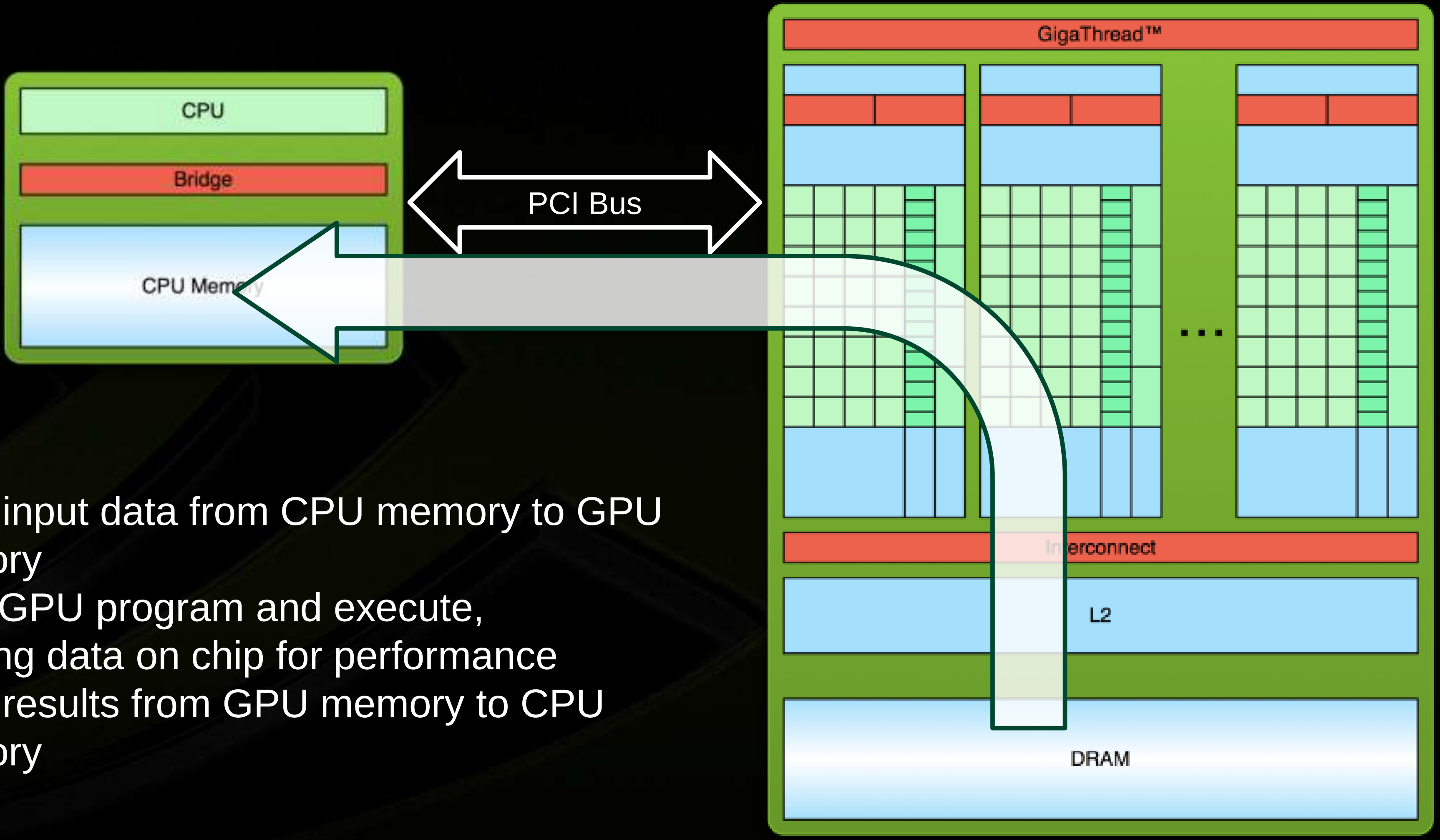
1. Copy input data from CPU memory to GPU memory

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

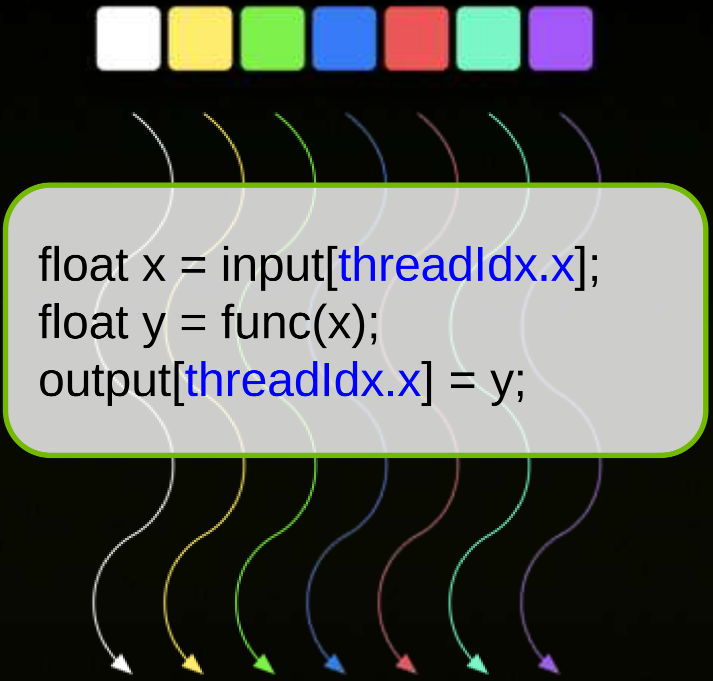
Simple Processing Flow



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

CUDA Kernels: Parallel Threads

- A **kernel** is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions

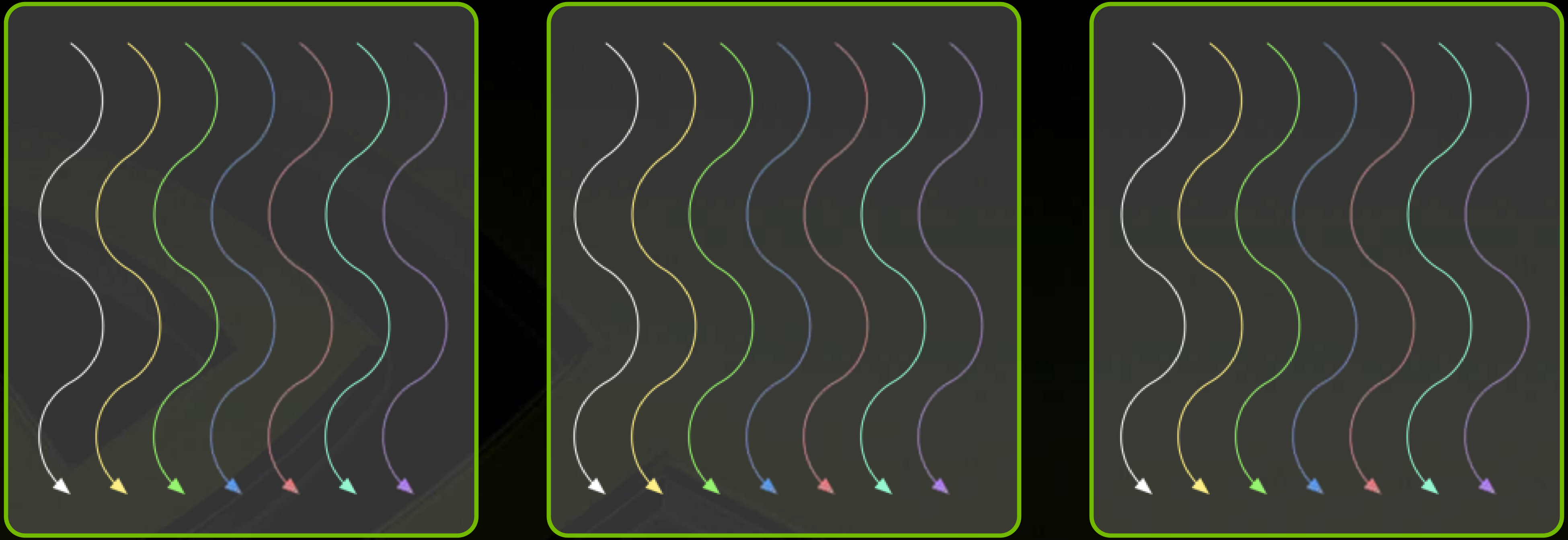


```
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```

CUDA Kernels: Subdivide into Blocks

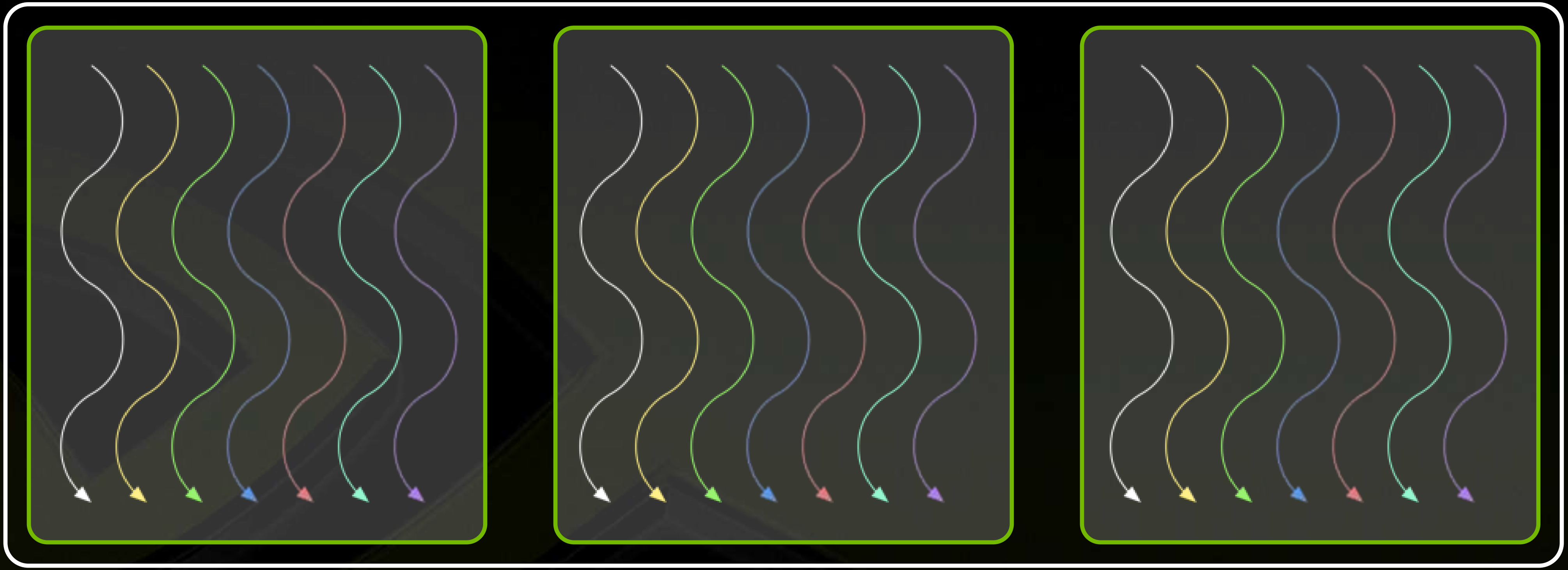
- Threads

CUDA Kernels: Subdivide into Blocks



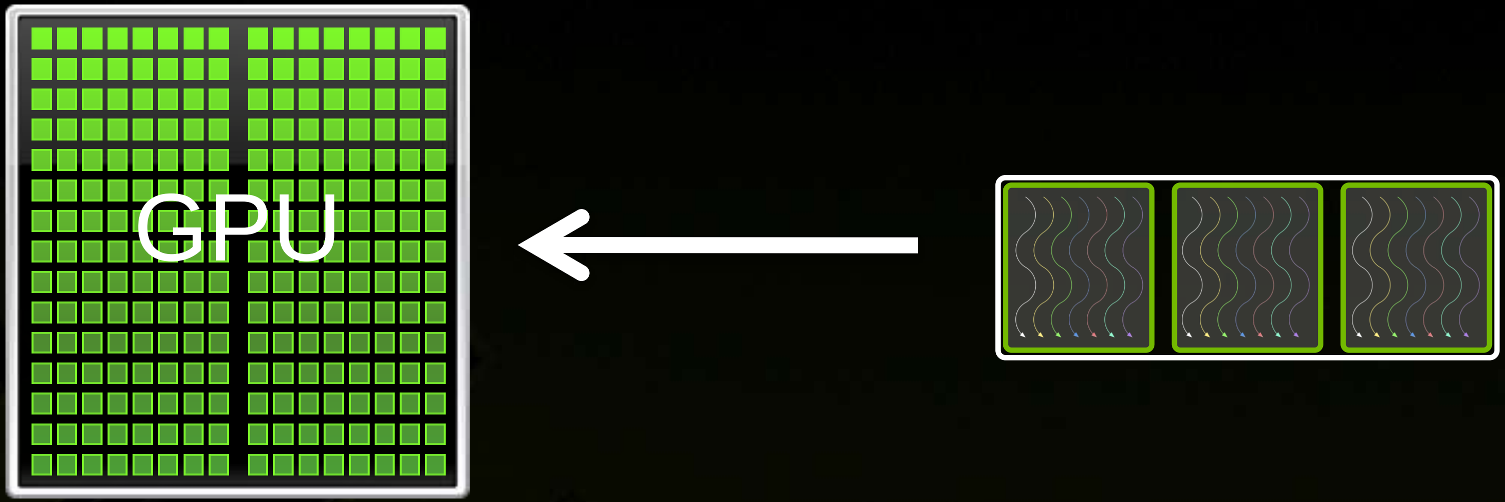
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



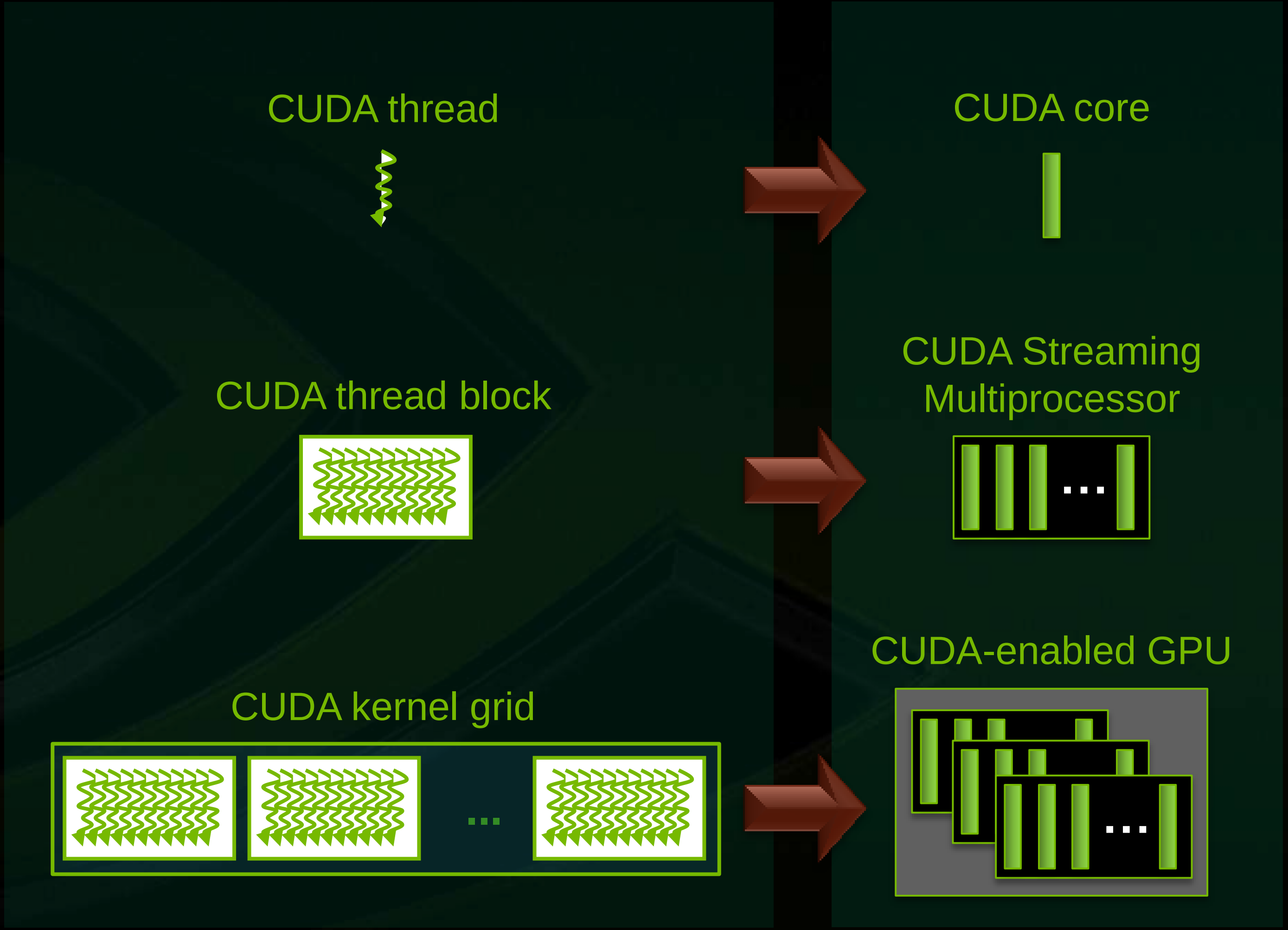
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

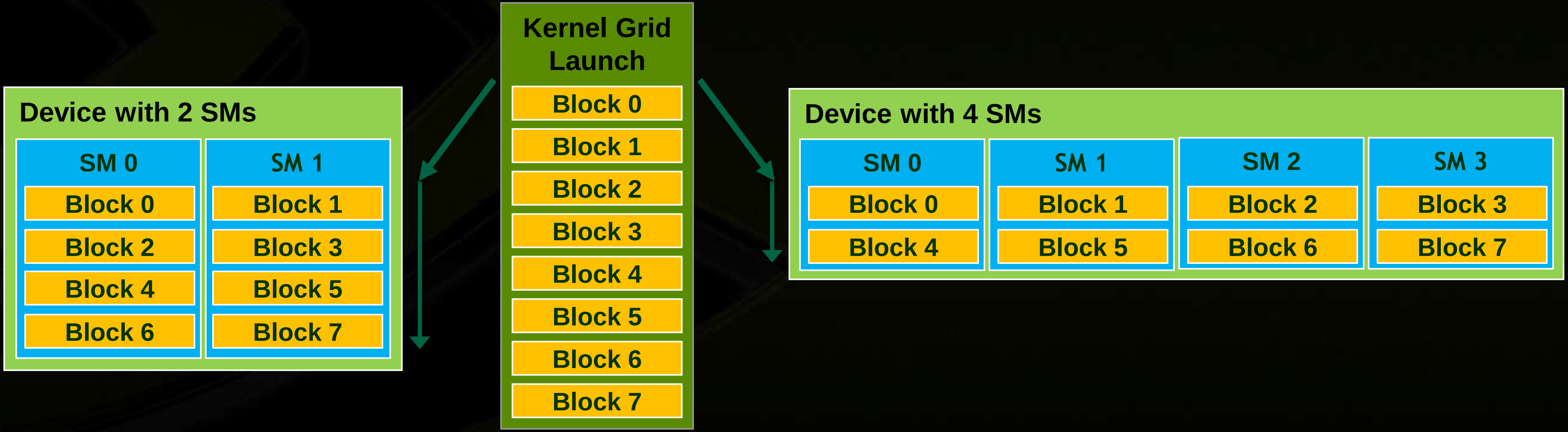
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store memory that they all use
  - Share results with each other
  - Cooperate to produce a single result
  - Synchronize with each other

Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs

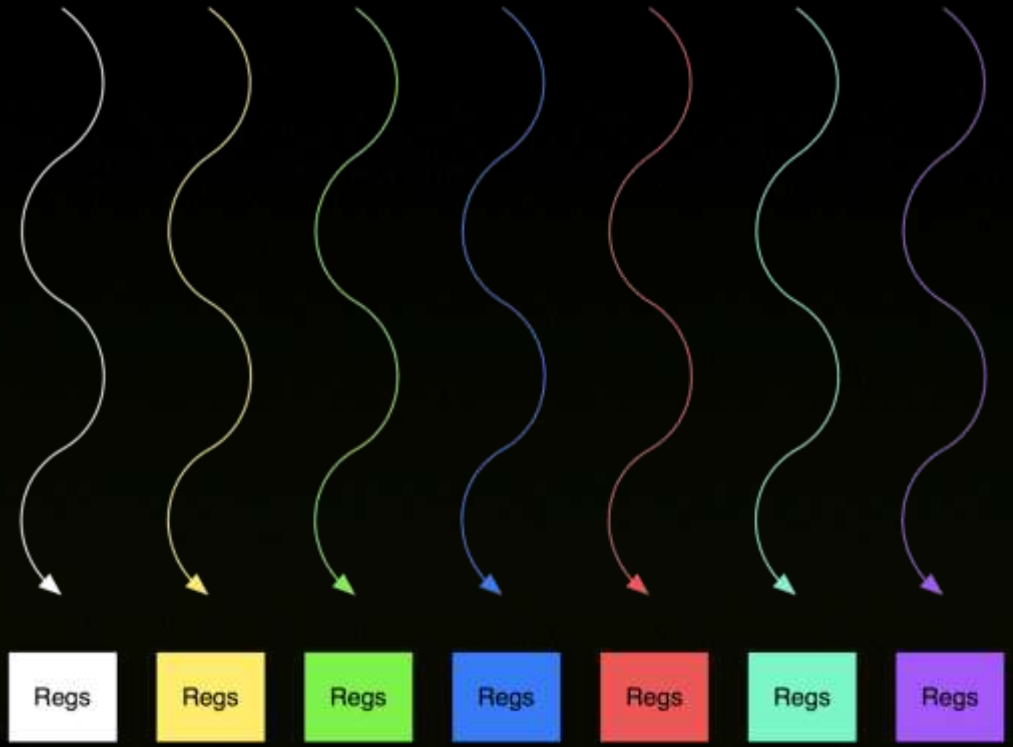


Warps (extra credit)

- Blocks are divided into 32 thread wide units called warps
  - Size of warps is implementation specific and can change in the future
- The SM creates, manages, schedules and executes threads at warp granularity
  - Each warp consists of 32 threads of contiguous threadIds
- All threads in a warp execute the same instruction
  - If threads of a warp diverge the warp serially executes each branch path taken
- When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible

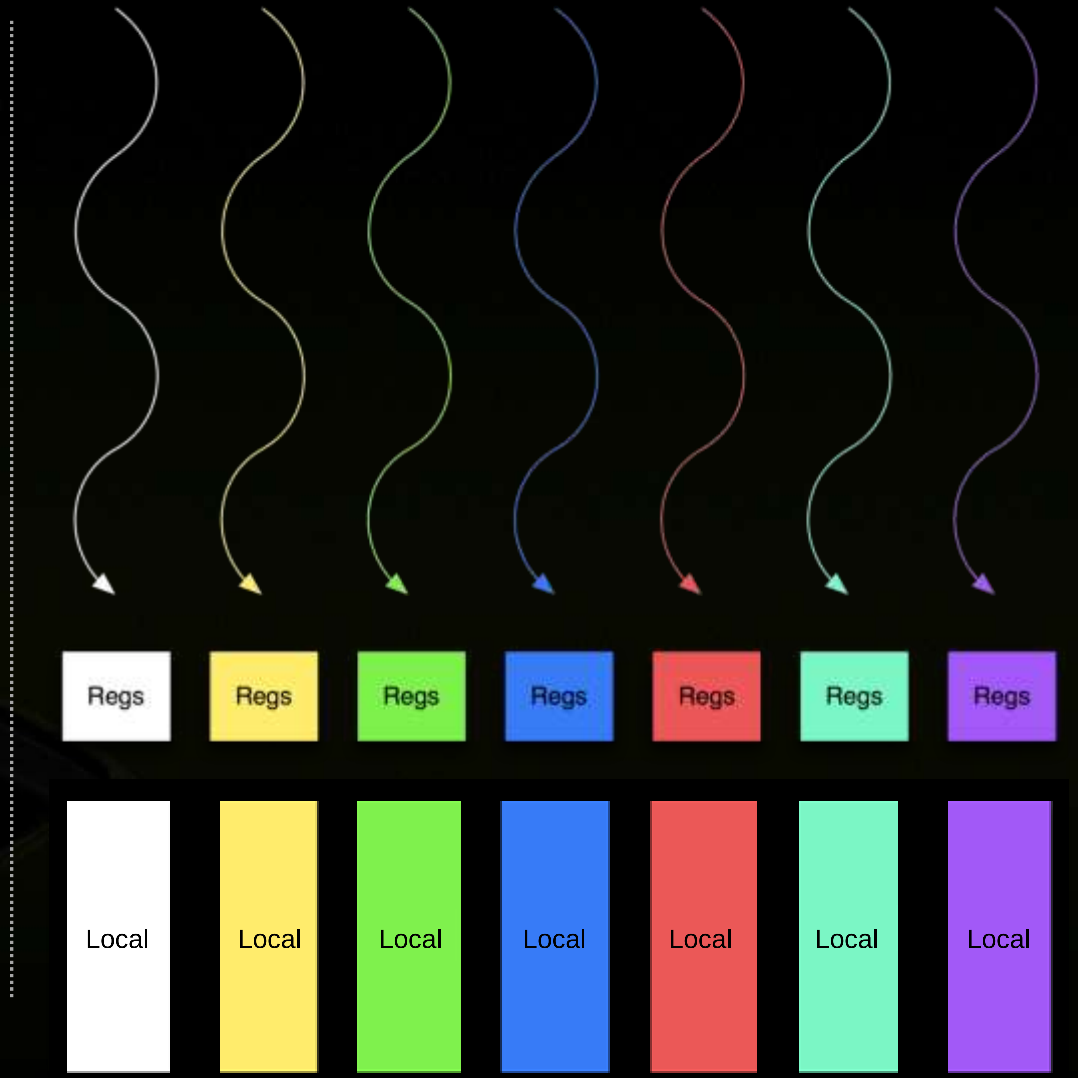
Memory hierarchy

- **Thread:**
  - Registers



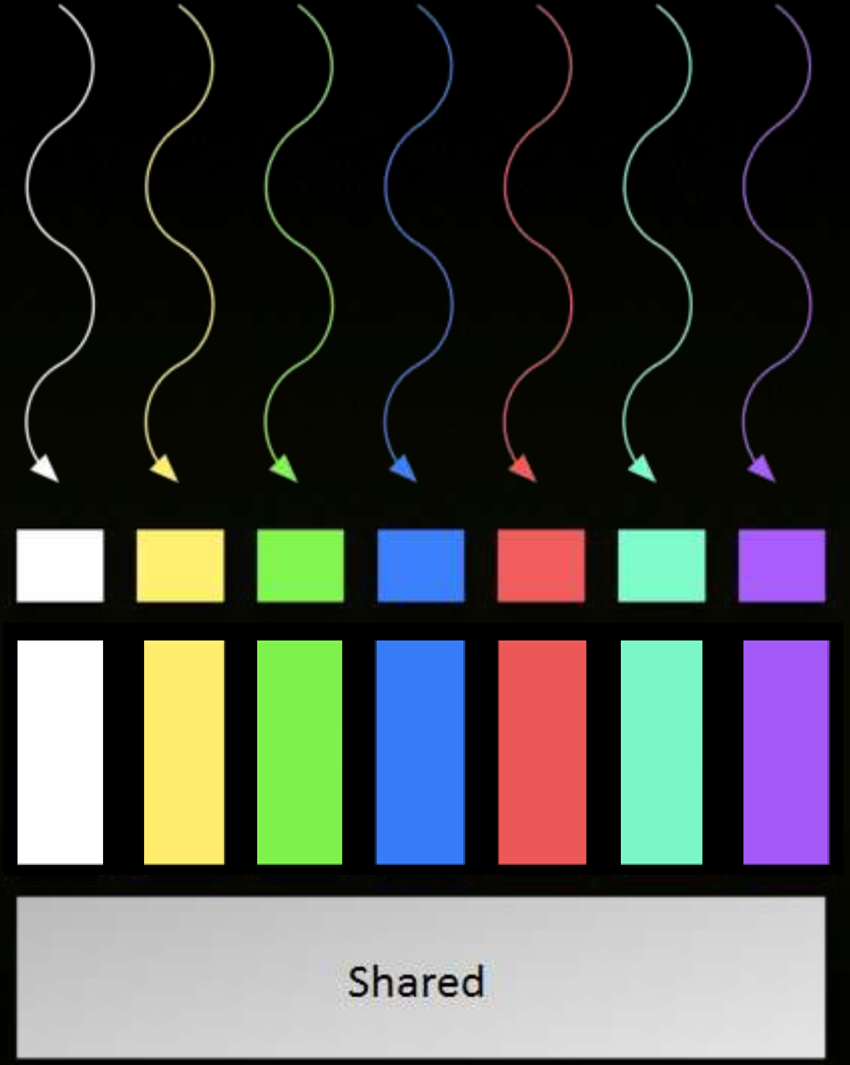
Memory hierarchy

- Thread:
  - Registers
  - Local memory



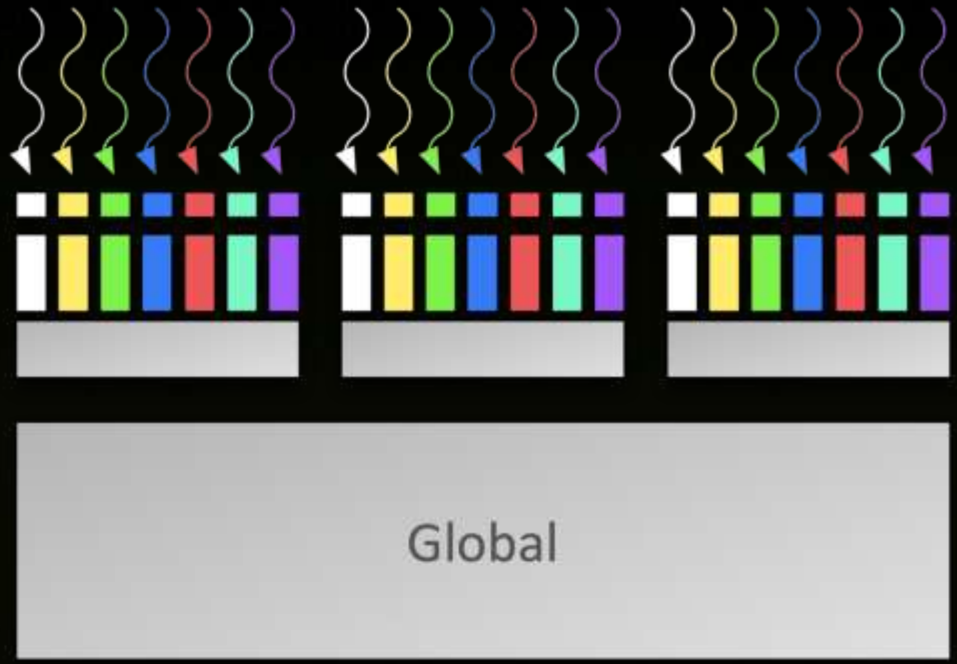
Memory hierarchy

- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory



Memory hierarchy

- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory



CUDA C Basics

(finally)

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”
  - First parameter is the number of blocks
  - Second parameter is the number of threads in each block
  - Parameters can be scalars (int) or multidimensional (dim3)
- 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;
}
```

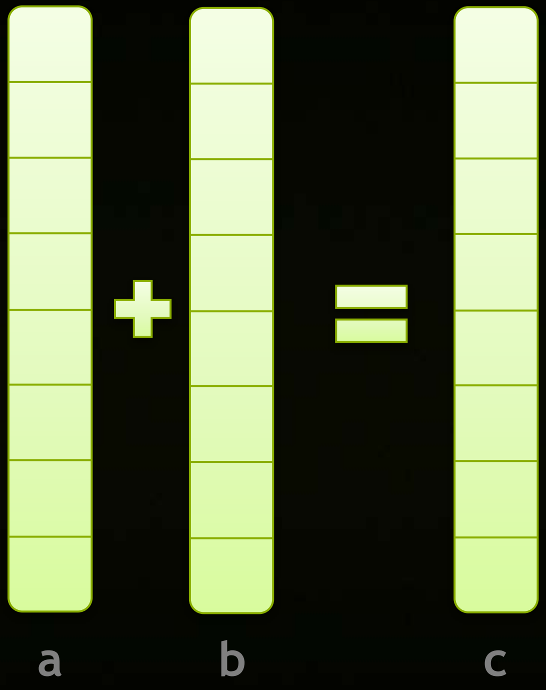
Output:

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

- `mykernel()` does nothing, somewhat anticlimactic!

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



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

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()`

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;
    }
```

Moving to Parallel

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

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

$$\downarrow$$

```
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`

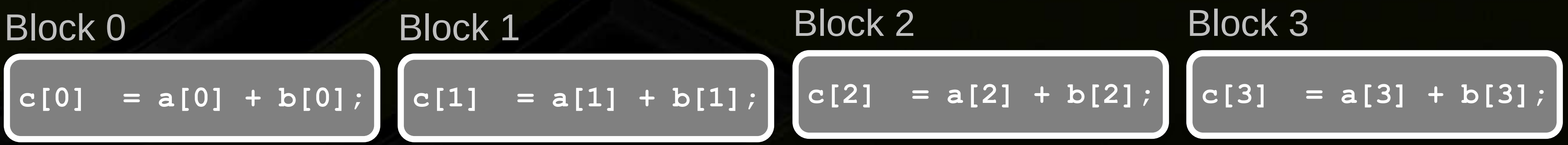
```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- 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:**



Block 0

```
c[0]  = a[0] + b[0];
```

Block 1

```
c[1]  = a[1] + b[1];
```

Block 2

```
c[2]  = a[2] + b[2];
```

Block 3

```
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;
}
```

Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

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()`

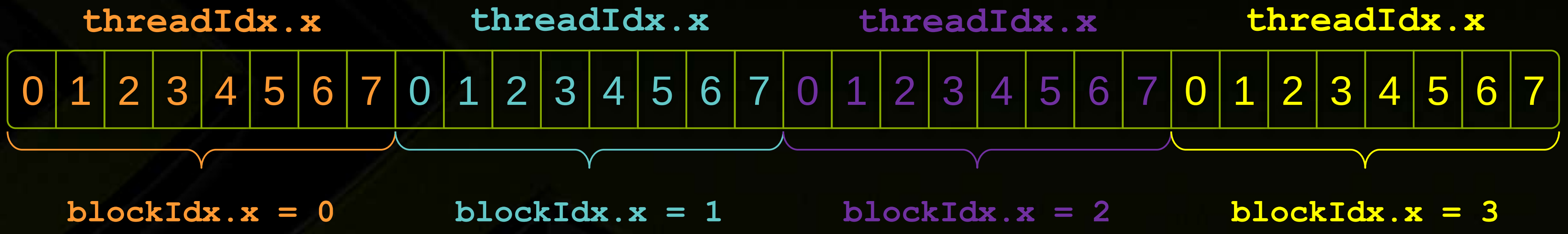
```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // 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;
}
```

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)

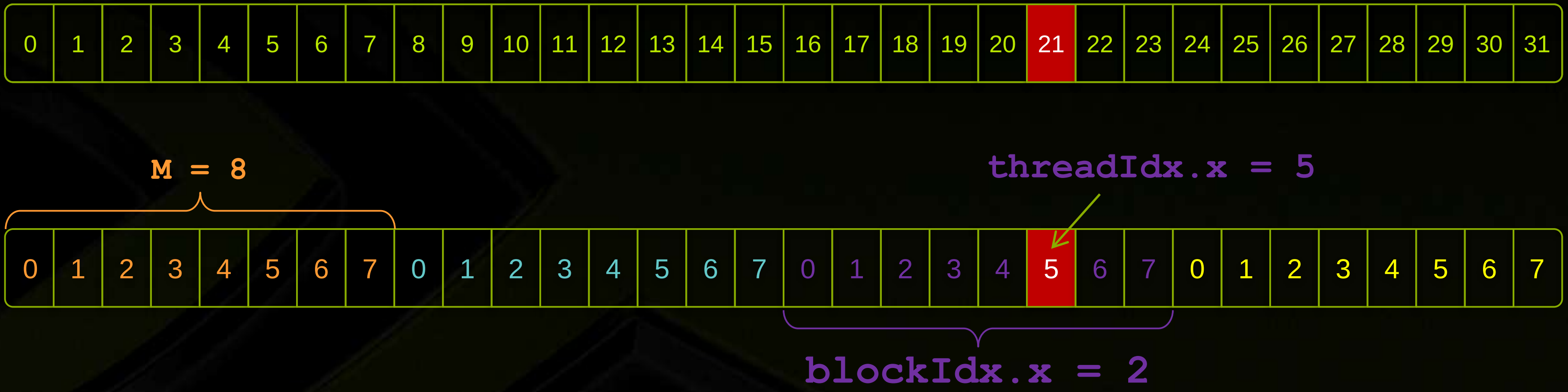


- With M threads/block a unique index for each thread is given by:

```
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);
```

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example…

Need More?

- Get CUDA: www.nvidia.com/getcuda
- Nsight: www.nvidia.com/nsight
- Programming Guide/Best Practices…
  - docs.nvidia.com
- Questions:
  - NVIDIA Developer forums devtalk.nvidia.com
  - Search or ask on www.stackoverflow.com/tags/cuda
- General: www.nvidia.com/cudazone

BACKUP

Quantum Chemistry Applications

Application	Features Supported	GPU Perf	Release Status	Notes
Abinit	Local Hamiltonian, non-local Hamiltonian, LOBPCG algorithm, diagonalization / orthogonalization	1.3-2.7X	Released; Version 7.0.5 Multi-GPU support	www.abinit.org
ACES III	Integrating scheduling GPU into SIAL programming language and SIP runtime environment	10X on kernels	Under development Multi-GPU support	http://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/deumens_ESaccel_2012.pdf
ADF	Fock Matrix, Hessians	TBD	Pilot project completed, Under development Multi-GPU support	www.scm.com
BigDFT	DFT; Daubechies wavelets, part of Abinit	5-25X (1 CPU core to GPU kernel)	Released June 2009, current release 1.6.0 Multi-GPU support	http://inac.cea.fr/L_Sim/BigDFT/news.html, http://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/BigDFT-Formalism.pdf and http://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/BigDFT-HPC-tues.pdf
Casino	TBD	TBD	Under development, Spring 2013 release Multi-GPU support	http://www.tcm.phy.cam.ac.uk/~mdt26/casino.html
CP2K	DBCSR (spare matrix multiply library)	2-7X	Under development Multi-GPU support	http://www.olcf.ornl.gov/wp-content/training/ascc_2012/friday/ACSS_2012_VandeVondele_s.pdf
GAMESS-US	Libqc with Rys Quadrature Algorithm, Hartree-Fock, MP2 and CCSD in Q4 2012		Released Multi-GPU support	Next release Q4 2012. http://www.msg.ameslab.gov/gamess/index.html

GPU Perf compared against Multi-core x86 CPU socket.
GPU Perf benchmarked on GPU supported features and may be a kernel to kernel perf comparison

Quantum Chemistry Applications

Application	Features Supported	GPU Perf	Release Status	Notes
GAMESS-UK	(ss|ss) type integrals within calculations using Hartree Fock *ab initio* methods and density functional theory. Supports organics & inorganics.	8x	Release in 2012 Multi-GPU support	http://www.ncbi.nlm.nih.gov/pubmed/21541963
Gaussian	Joint PGI, NVIDIA & Gaussian Collaboration	TBD	Under development Multi-GPU support	Announced Aug. 29, 2011 http://www.gaussian.com/g_press/nvidia_press.htm
GPAW	Electrostatic poisson equation, orthonormalizing of vectors, residual minimization method (rmm-diis)	8x	Released Multi-GPU support	https://wiki.fysik.dtu.dk/gpaw/devel/projects/gpu.html, Samuli Hakala (CSC Finland) & Chris O'Grady (SLAC)
Jaguar	Investigating GPU acceleration	TBD	Under development Multi-GPU support	Schrodinger, Inc. http://www.schrodinger.com/kb/278
MOLCAS	CU_BLAS support	1.1x	Released, Version 7.8 Single GPU. Additional GPU support coming in Version 8	www.molcas.org
MOLPRO	Density-fitted MP2 (DF-MP2), density fitted local correlation methods (DF-RHF, DF-KS), DFT		Under development Multiple GPU	www.molpro.net Hans-Joachim Werner

GPU Perf compared against Multi-core x86 CPU socket.
GPU Perf benchmarked on GPU supported features
and may be a kernel to kernel perf comparison

Quantum Chemistry Applications

Application	Features Supported	GPU Perf	Release Status	Notes
MOPAC2009	pseudodiagonalization, full diagonalization, and density matrix assembling	3.8-14X	Under Development Single GPU	Academic port. http://openmopac.net
NWChem	Triples part of Reg-CCSD(T), CCSD & EOMCCSD task schedulers	3-10X projected	Release targeting March 2013 Multiple GPUs	Development GPGPU benchmarks: www.nwchem-sw.org And http://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/Krishnamoorthy-ESCMA12.pdf
Octopus	DFT and TDDFT	TBD	Released	http://www.tddft.org/programs/octopus/
PEtot	Density functional theory (DFT) plane wave pseudopotential calculations	6-10X	Released Multi-GPU	First principles materials code that computes the behavior of the electron structures of materials
Q-CHEM	RI-MP2	8x-14x	Released, Version 4.0	http://www.q-chem.com/doc_for_web/qchem_manual_4.0.pdf

GPU Perf compared against Multi-core x86 CPU socket.
GPU Perf benchmarked on GPU supported features
and may be a kernel to kernel perf comparison

Quantum Chemistry Applications

Application	Features Supported	GPU Perf	Release Status	Notes
QMCPACK	Main features	3-4x	Released Multiple GPUs	NCSA University of Illinois at Urbana-Champaign http://cms.mcc.uiuc.edu/qmcpack/index.php/GPU_version_of_QMCPACK
Quantum Espresso/PWscf	PWscf package: linear algebra (matrix multiply), explicit computational kernels, 3D FFTs	2.5-3.5x	Released Version 5.0 Multiple GPUs	Created by Irish Centre for High-End Computing http://www.quantum-espresso.org/index.php and http://www.quantum-espresso.org/
TeraChem	"Full GPU-based solution"	44-650X vs. GAMESS CPU version	Released Version 1.5 Multi-GPU/single node	Completely redesigned to exploit GPU parallelism. YouTube: http://youtu.be/EJODzk6RFxE?hd=1 and http://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/Luehr-ESCMA.pdf
VASP	Hybrid Hartree-Fock DFT functionals including exact exchange	2x 2 GPUs comparable to 128 CPU cores	Available on request Multiple GPUs	By Carnegie Mellon University http://arxiv.org/pdf/1111.0716.pdf
WL-LSMS	Generalized Wang-Landau method	3x with 32 GPUs vs. 32 (16-core) CPUs	Under development Multi-GPU support	NICS Electronic Structure Determination Workshop 2012: http://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/Eisenbach_OakRidge_February.pdf

GPU Perf compared against Multi-core x86 CPU socket.
GPU Perf benchmarked on GPU supported features
and may be a kernel to kernel perf comparison

Viz, “Docking” and Related Applications Growing

Related Applications	Features Supported	GPU Perf	Release Status	Notes
Amira 5®	3D visualization of volumetric data and surfaces	70x	Released, Version 5.3.3 Single GPU	Visualization from Visage Imaging. Next release, 5.4, will use GPU for general purpose processing in some functions http://www.visageimaging.com/overview.html
BINDSURF	Allows fast processing of large ligand databases	100X	Available upon request to authors; single GPU	High-Throughput parallel blind Virtual Screening, http://www.biomedcentral.com/1471-2105/13/S14/S13
BUDE	Empirical Free Energy Forcefield	6.5-13.4X	Released Single GPU	University of Bristol http://www.bris.ac.uk/biochemistry/cpfg/bude/bude.htm
Core Hopping	GPU accelerated application	3.75-5000X	Released, Suite 2011 Single and multi-GPUs.	Schrodinger, Inc. http://www.schrodinger.com/products/14/32/
FastROCS	Real-time shape similarity searching/comparison	800-3000X	Released Single and multi-GPUs.	Open Eyes Scientific Software http://www.eyesopen.com/fastrocs
PyMol	Lines: 460% increase Cartoons: 1246% increase Surface: 1746% increase Spheres: 753% increase Ribbon: 426% increase	1700x	Released, Version 1.5 Single GPUs	http://pymol.org/
VMD	High quality rendering, large structures (100 million atoms), analysis and visualization tasks, multiple GPU support for display of molecular orbitals	100-125X or greater on kernels	Released, Version 1.9	Visualization from University of Illinois at Urbana-Champaign http://www.ks.uiuc.edu/Research/vmd/

GPU Perf compared against Multi-core x86 CPU socket.
GPU Perf benchmarked on GPU supported features
and may be a kernel to kernel perf comparison

Bioinformatics Applications

Application	Features Supported	GPU Speedup	Release Status	Website
BarraCUDA	Alignment of short sequencing reads	6-10x	Version 0.6.2 – 3/2012 Multi-GPU, multi-node	http://seqbarracuda.sourceforge.net/
CUDASW++	Parallel search of Smith-Waterman database	10-50x	Version 2.0.8 – Q1/2012 Multi-GPU, multi-node	http://sourceforge.net/projects/cudasw/
CUSHAW	Parallel, accurate long read aligner for large genomes	10x	Version 1.0.40 – 6/2012 Multiple-GPU	http://cushaw.sourceforge.net/
GPU-BLAST	Protein alignment according to BLASTP	3-4x	Version 2.2.26 – 3/2012 Single GPU	http://eudoxus.cheme.cmu.edu/gpublast/gpublast.html
GPU-HMMER	Parallel local and global search of Hidden Markov Models	60-100x	Version 2.3.2 – Q1/2012 Multi-GPU, multi-node	http://www.mpihmmer.org/installguideGPUHMMER.htm
mCUDA-MEME	Scalable motif discovery algorithm based on MEME	4-10x	Version 3.0.12 Multi-GPU, multi-node	https://sites.google.com/site/yongchaosoftware/mcuda-meme
SeqNFind	Hardware and software for reference assembly, blast, SW, HMM, de novo assembly	400x	Released. Multi-GPU, multi-node	http://www.seqnfind.com/
UGENE	Fast short read alignment	6-8x	Version 1.11 – 5/2012 Multi-GPU, multi-node	http://ugene.unipro.ru/
WideLM	Parallel linear regression on multiple similarly-shaped models	150x	Version 0.1-1 – 3/2012 Multi-GPU, multi-node	http://insilicos.com/products/widelm

GPU Perf compared against same or similar code running on single CPU machine
Performance measured internally or independently

Overview

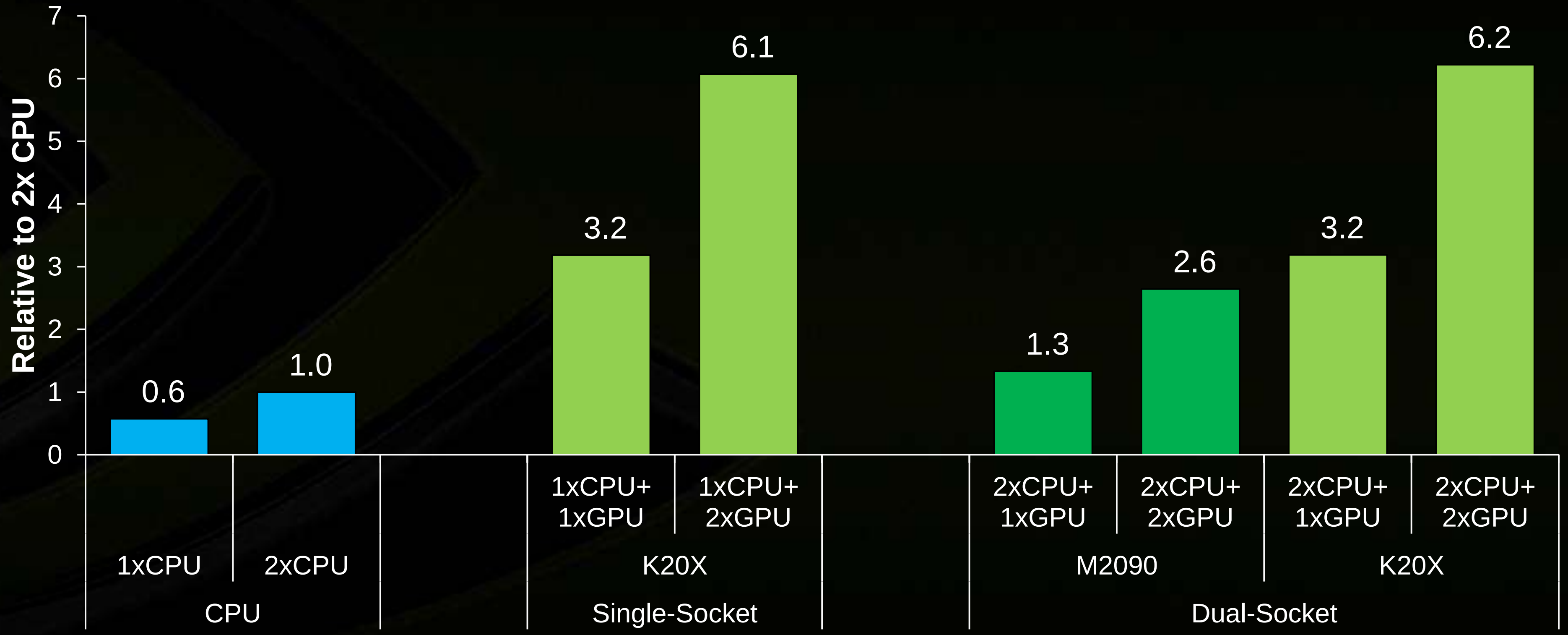
- GPU performance and scalability across multiple scientific domains & multiple algorithmic motifs
- Different approaches taken (libraries, OpenACC, programming languages)
- Lessons learned

LSMS – Materials Science

LSMS

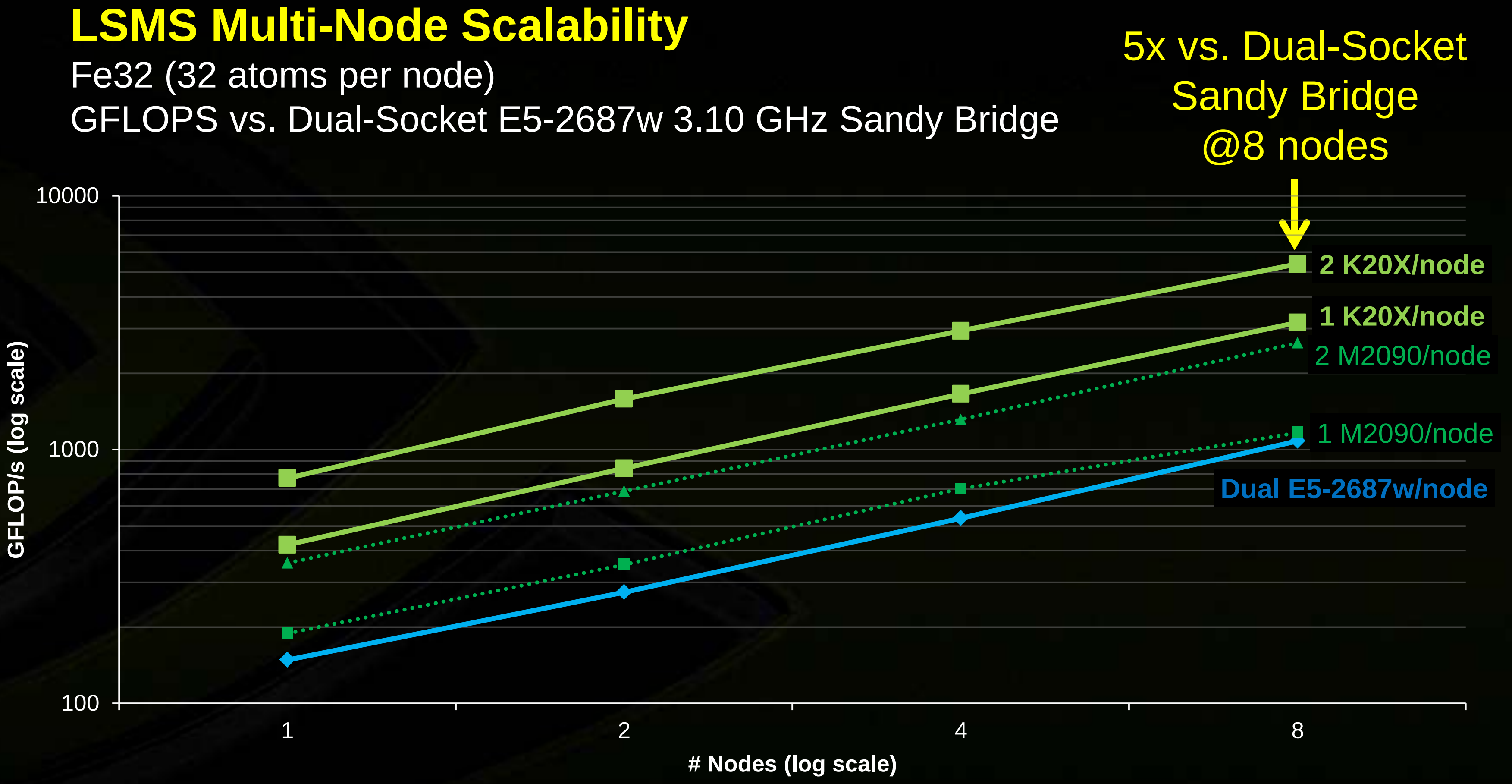
Fe32 (32 atoms per node)

Relative Performance (FLOPS) vs. Dual-Socket E5-2687w 3.10 GHz Sandy Bridge



Preliminary, NVIDIA Confidential – not for distribution

LSMS – Materials Science

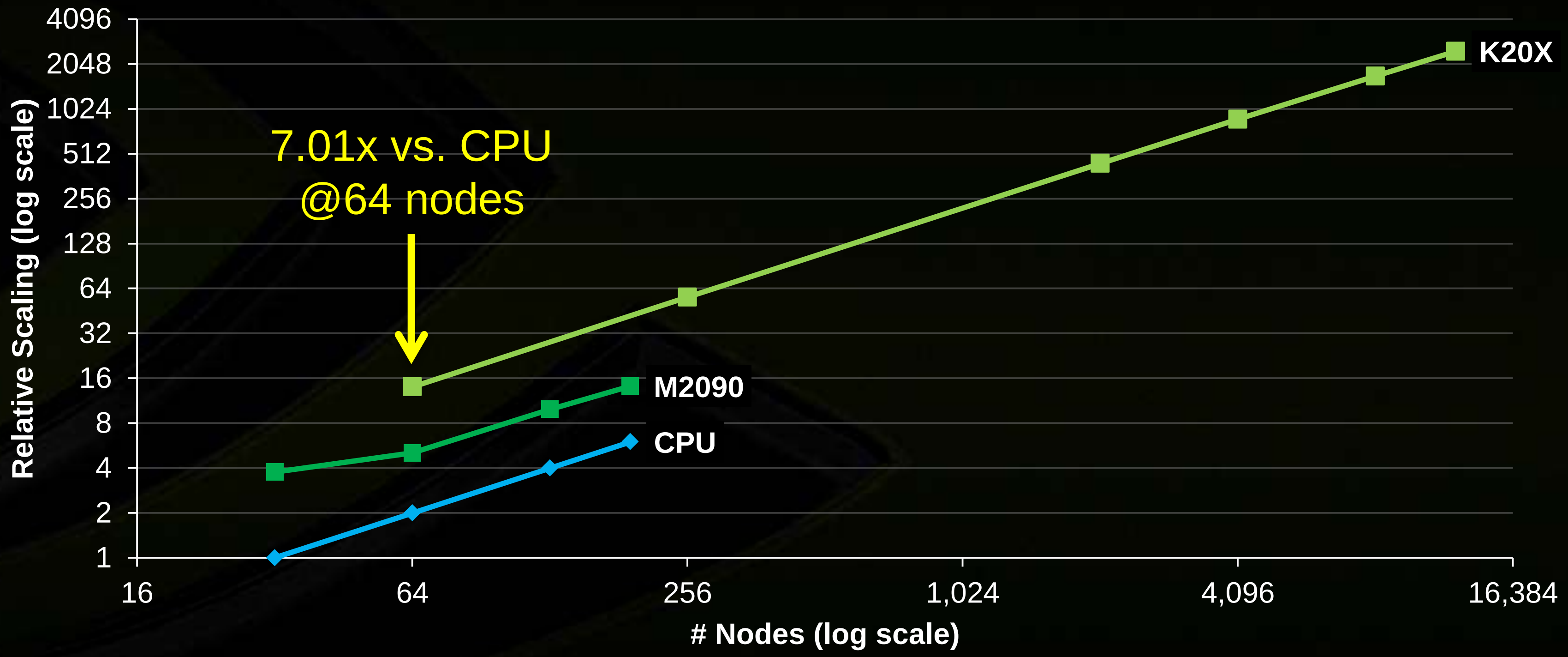


Preliminary, NVIDIA Confidential – not for distribution

gWL-LSMS – Materials Science

gWL-LSMS
Fe1024 (1024 atoms per 32 nodes)
Relative Scaling (FLOPS) on Cray XK7

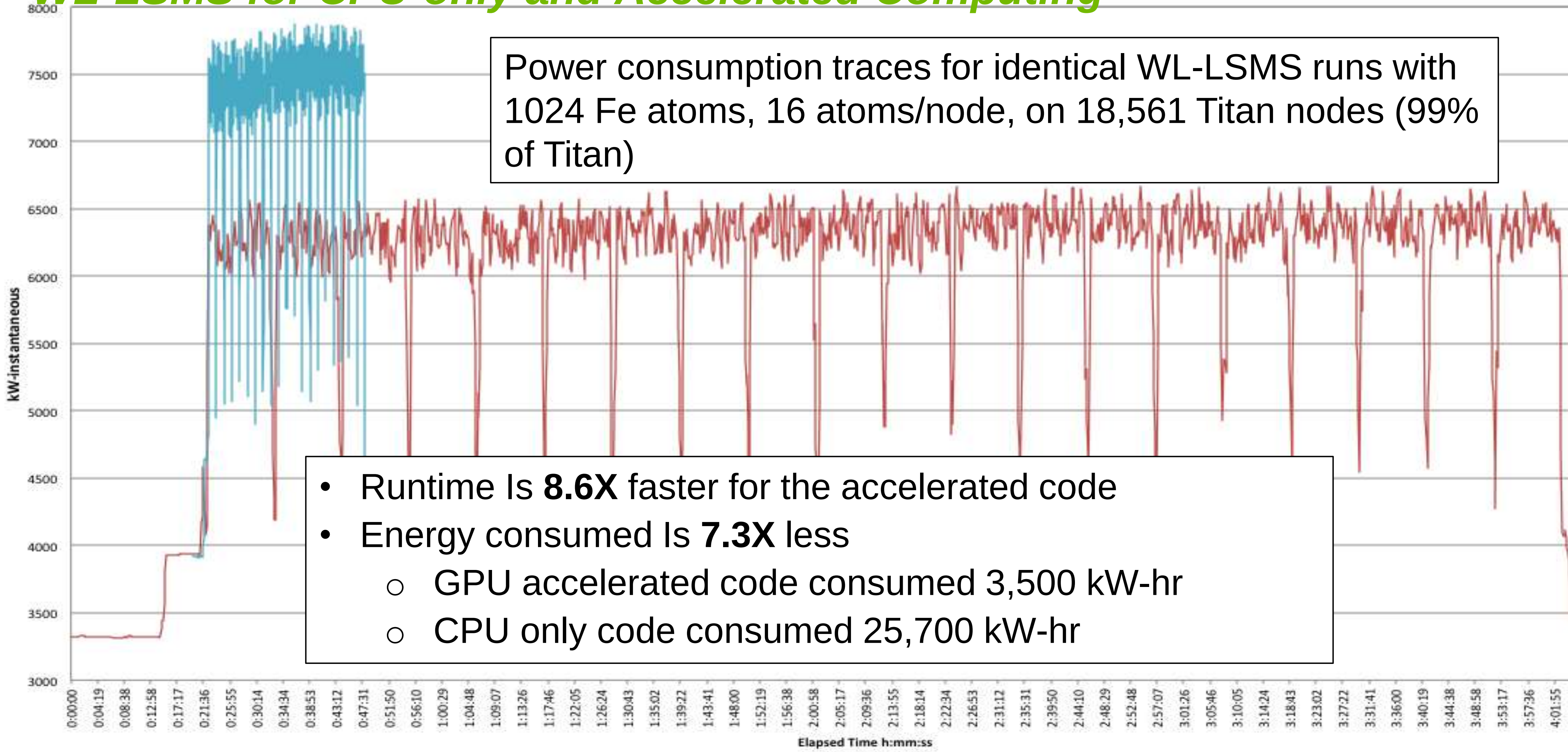
"CPU" node = XK7 (1x Interlagos)
"M2090" node = XK6 (1x M2090 + 1x Interlagos)
"K20X" node = XK7 (1x K20X + 1x Interlagos)



Preliminary, NVIDIA Confidential – not for distribution

Application Power Efficiency of the Cray XK7

WL-LSMS for CPU-only and Accelerated Computing



Power consumption traces for identical WL-LSMS runs with 1024 Fe atoms, 16 atoms/node, on 18,561 Titan nodes (99% of Titan)

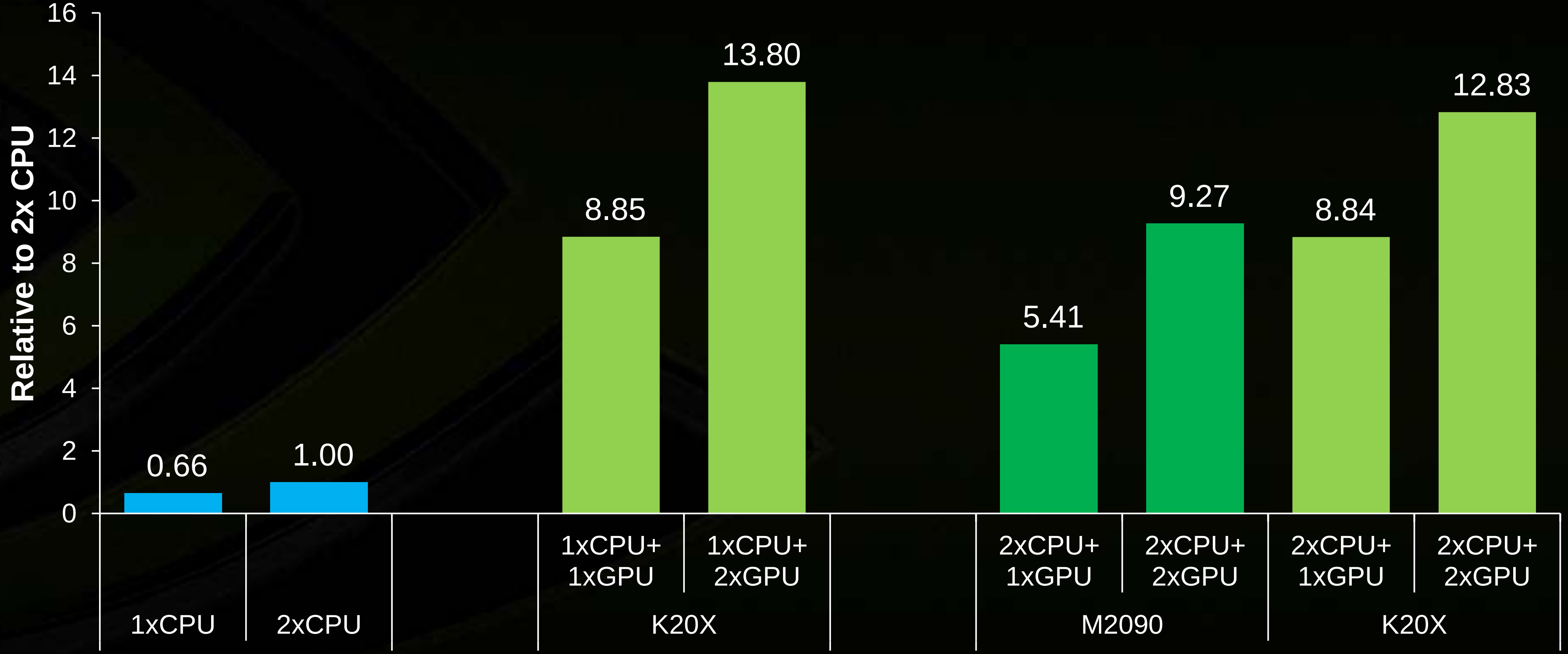
- Runtime Is **8.6X** faster for the accelerated code
- Energy consumed Is **7.3X** less
  - GPU accelerated code consumed 3,500 kW-hr
  - CPU only code consumed 25,700 kW-hr

SPECFEM3D – Earth Science

SPECFEM3D

meshfed3D-256x128x15

Relative Performance (solver time) vs. E5-2687w 3.10 GHz Sandy Bridge

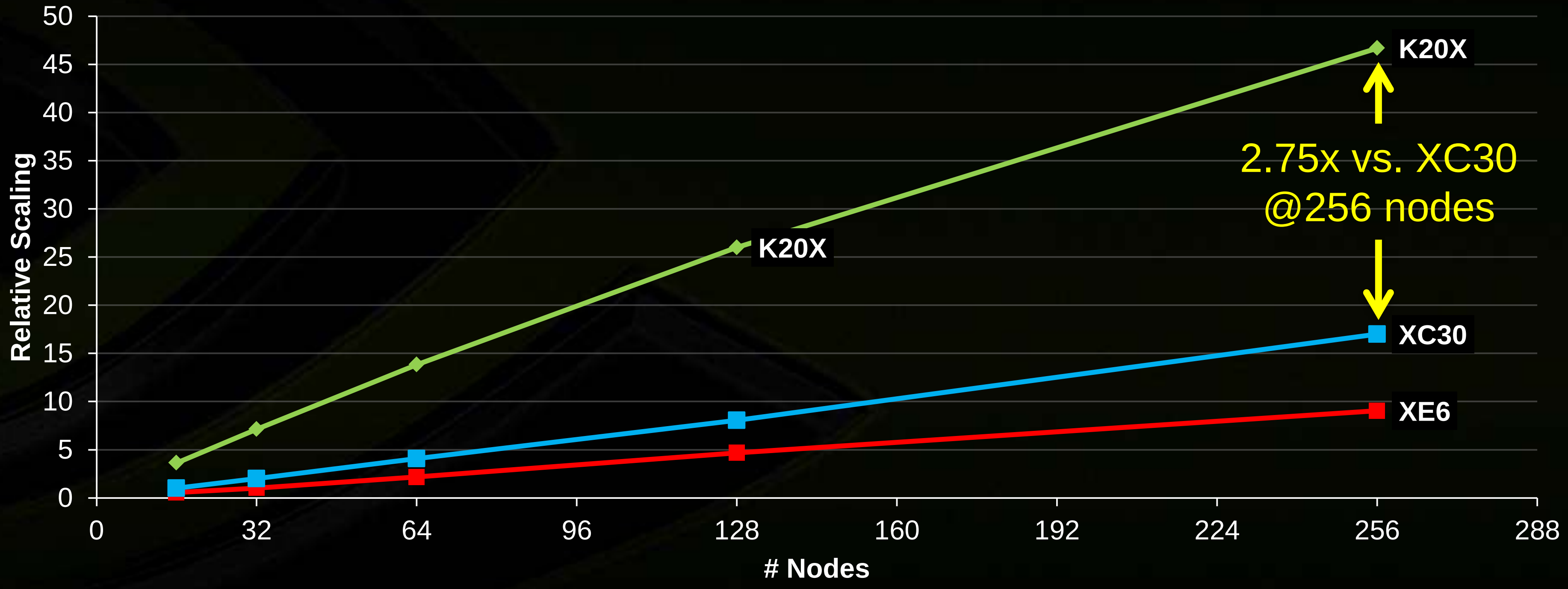


Preliminary, NVIDIA Confidential – not for distribution

SPECFEM3D – Earth Science

SPECFEM3D on Sandy Bridge
512x512x20, 5 M Elastic cells, 1000 timesteps
Relative Scaling (solver time)

"K20X" node = XK7 (1x K20X + 1x Interlagos)
"XE6" node = XE6 (2x Interlagos)
"XC30" node = XC30 (2x Sandy Bridge)



Preliminary, NVIDIA Confidential – not for distribution

SPECFEM3D – Earth Science

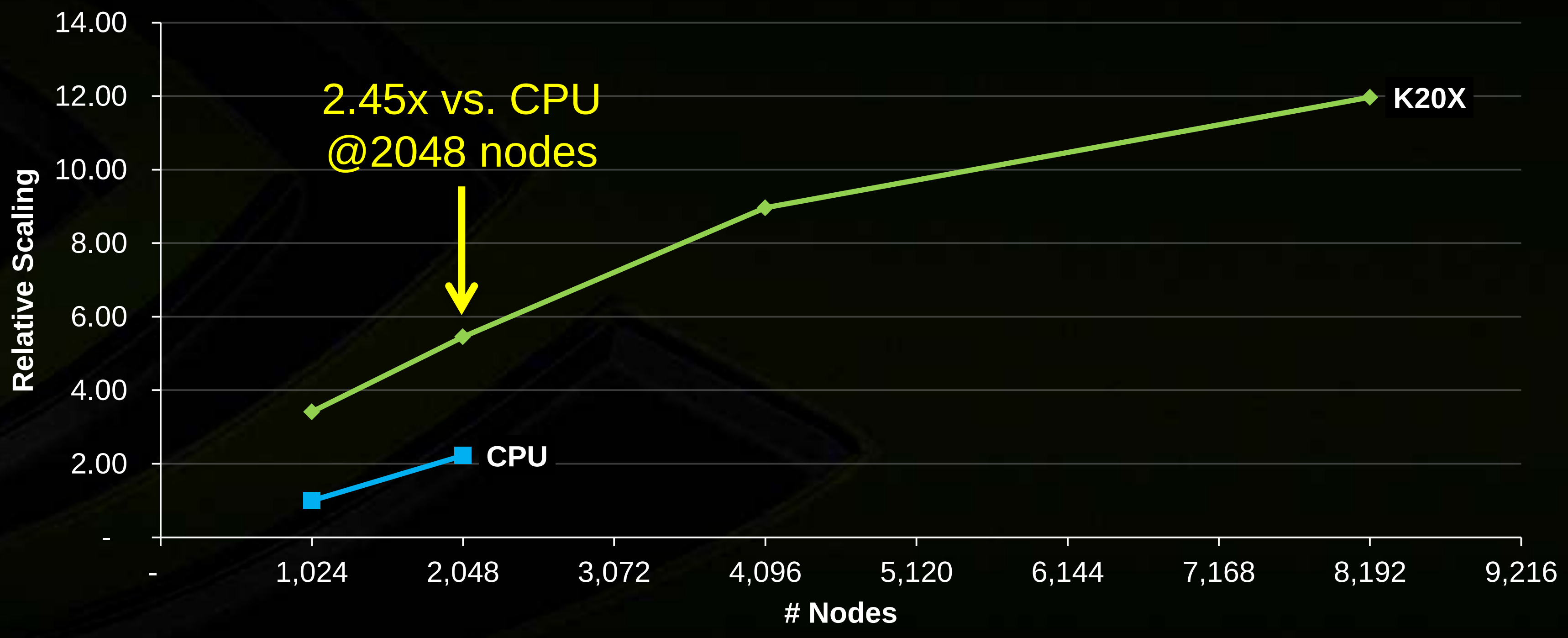
SPECFEM3D on Titan

2048x2048x15, 33 M cells, 2000 timesteps

Relative Scaling (solver time) on Cray XK7

“CPU” node = XK7 (1x Interlagos)

“K20X” node = XK7 (1x K20X + 1x Interlagos)



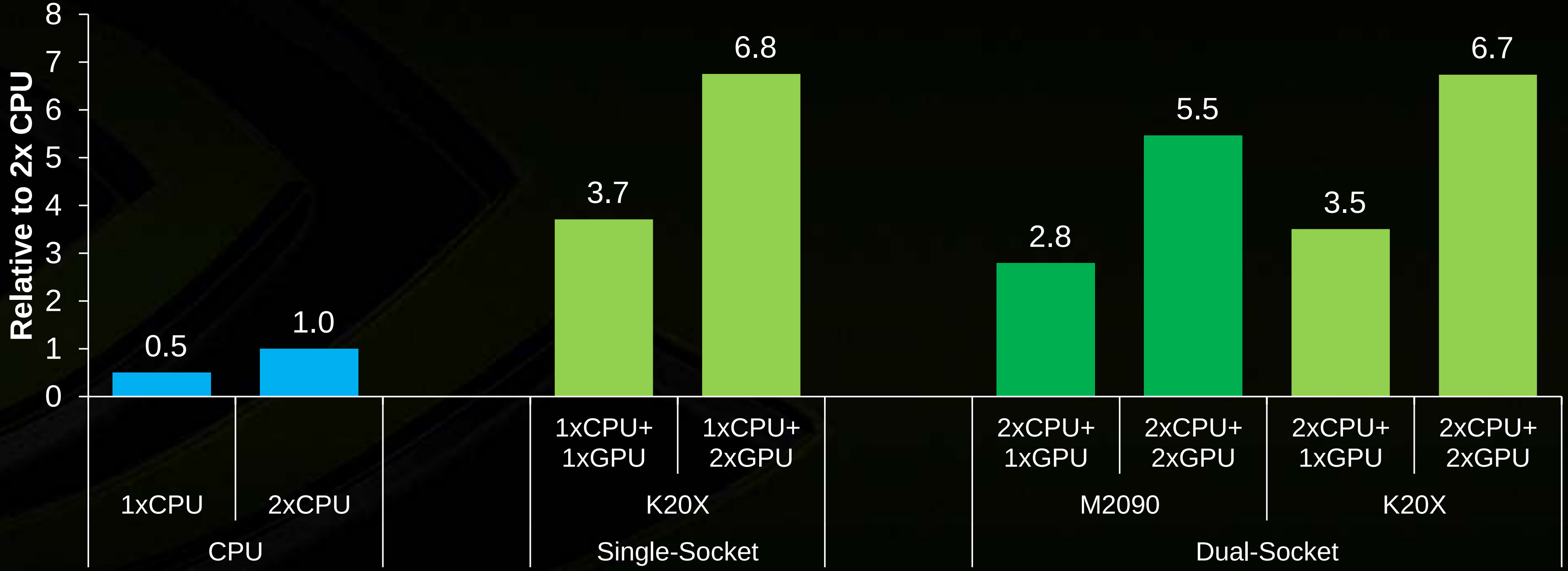
Preliminary, NVIDIA Confidential – not for distribution

Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma

24^3x128 lattice

Relative Performance (Propagator) vs. E5-2687w 3.10 GHz Sandy Bridge



Preliminary, NVIDIA Confidential – not for distribution

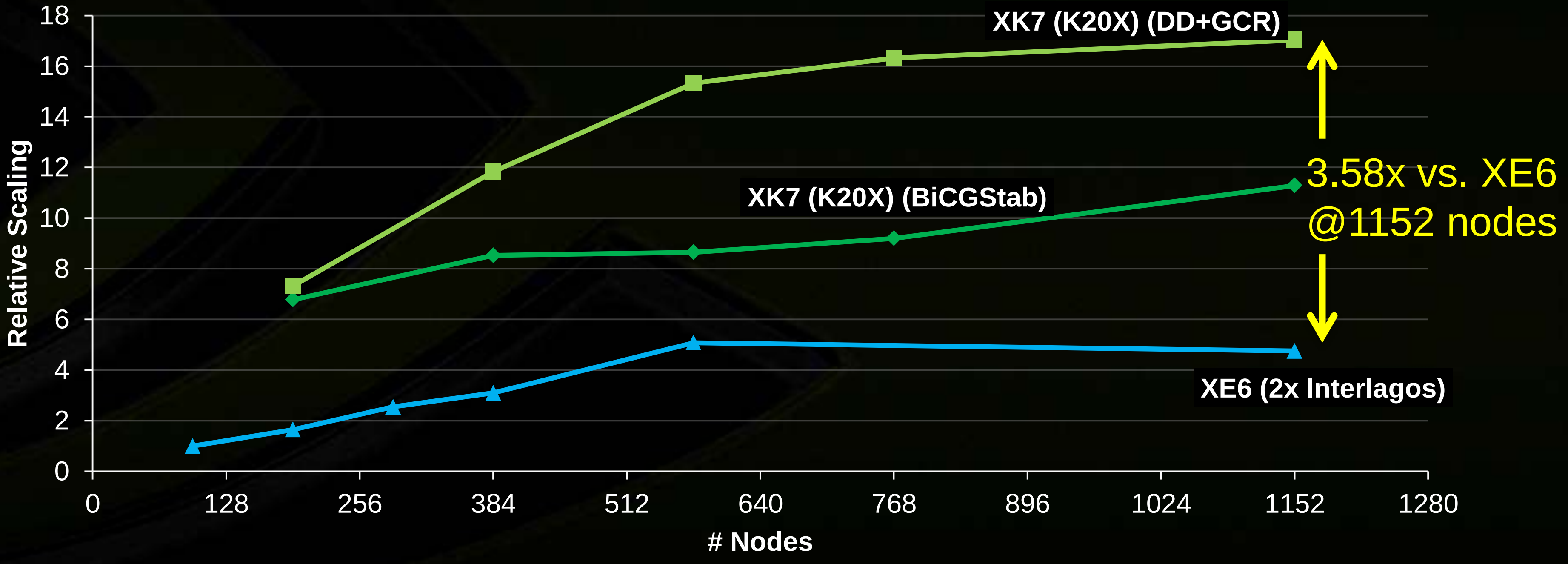
Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma

48^3x512 lattice

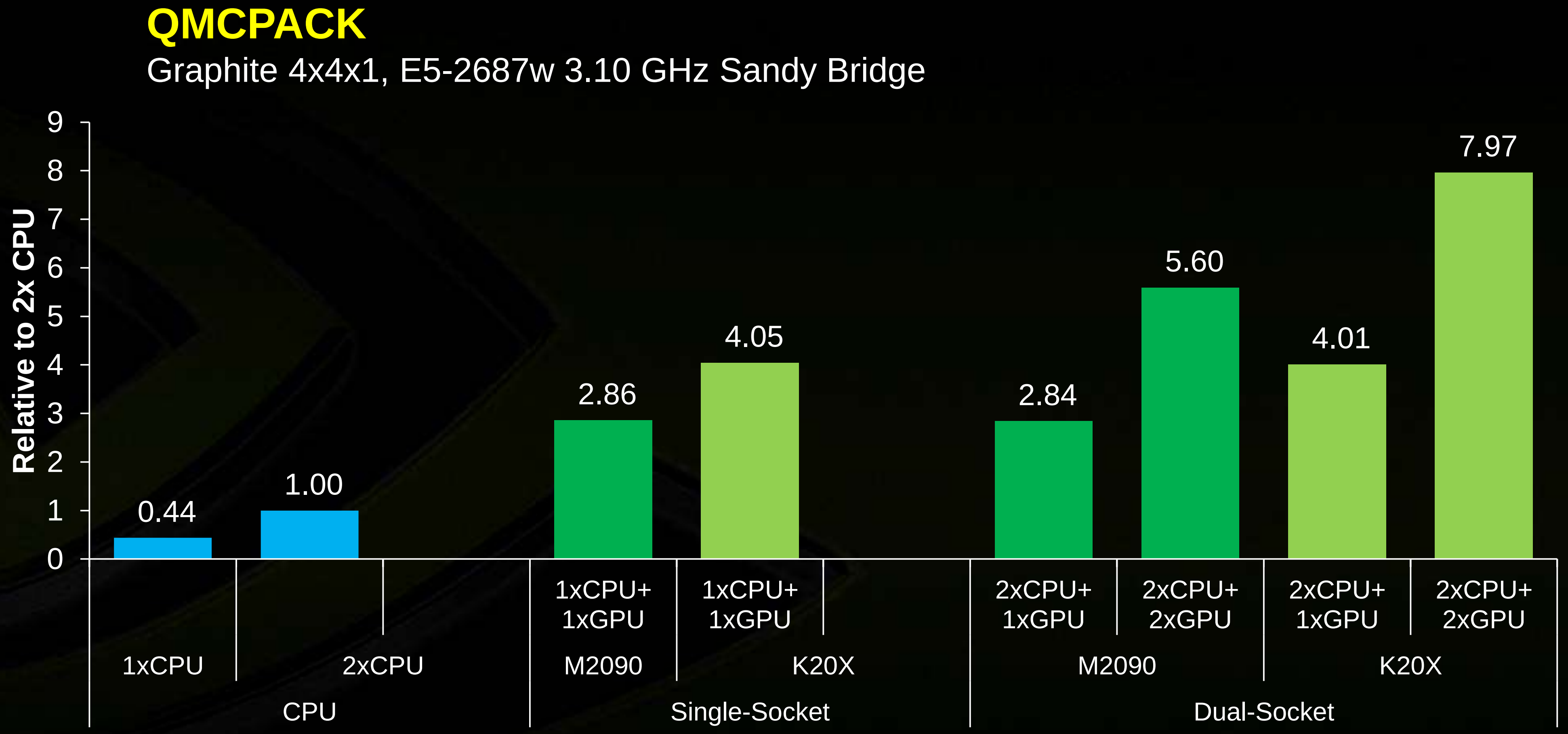
Relative Scaling (Application Time)

“XK7” node = XK7 (1x K20X + 1x Interlagos)
“XE6” node = XE6 (2x Interlagos)



Preliminary, NVIDIA Confidential – not for distribution

QMCPACK – Materials Science

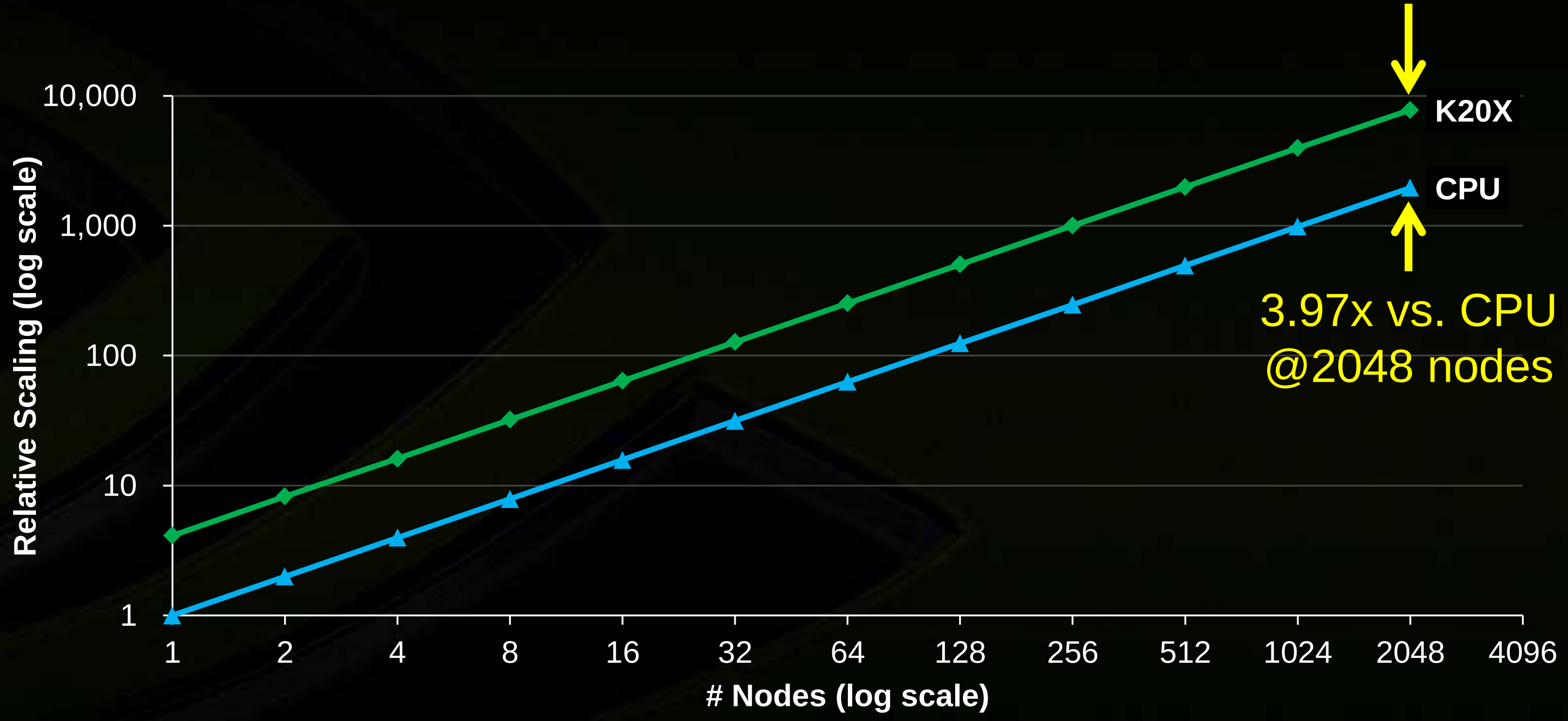


Preliminary, NVIDIA Confidential – not for distribution

QMCPACK – Materials Science

QMCPACK
3x3x1 Graphite
Relative Scaling (Efficiency) on Cray XK7

"CPU" node = XK7 (1x Interlagos)
"K20X" node = XK7 (1x K20X + 1x Interlagos)

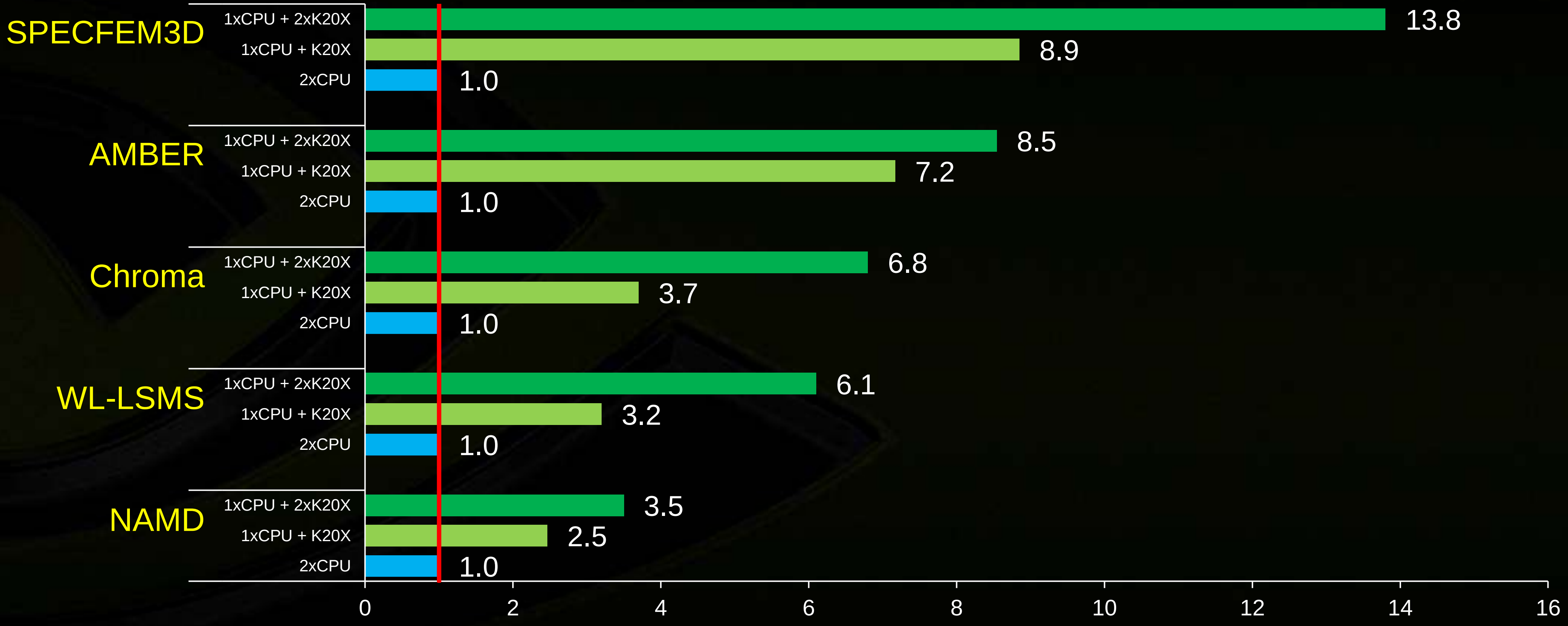


Preliminary, NVIDIA Confidential – not for distribution

Single-node Performance Summary

Relative Performance vs. dual-socket Sandy Bridge

E5-2687w 3.10 GHz Sandy Bridge



Preliminary, NVIDIA Confidential – not for distribution

Summary

- Performance
- Scalability
- Variety of programming approaches
- Futures