An Introduction to CUDA for HPC

University of Maryland, Baltimore County

April 12, 2013

Frank Willmore

willmore@tacc.utexas.edu

An Introduction to CUDA for HPC

- Some Definitions
- Comparison of CPU and GPU architectures
- CUDA programming model
- Lab exercise
- Lab exercise walk-through

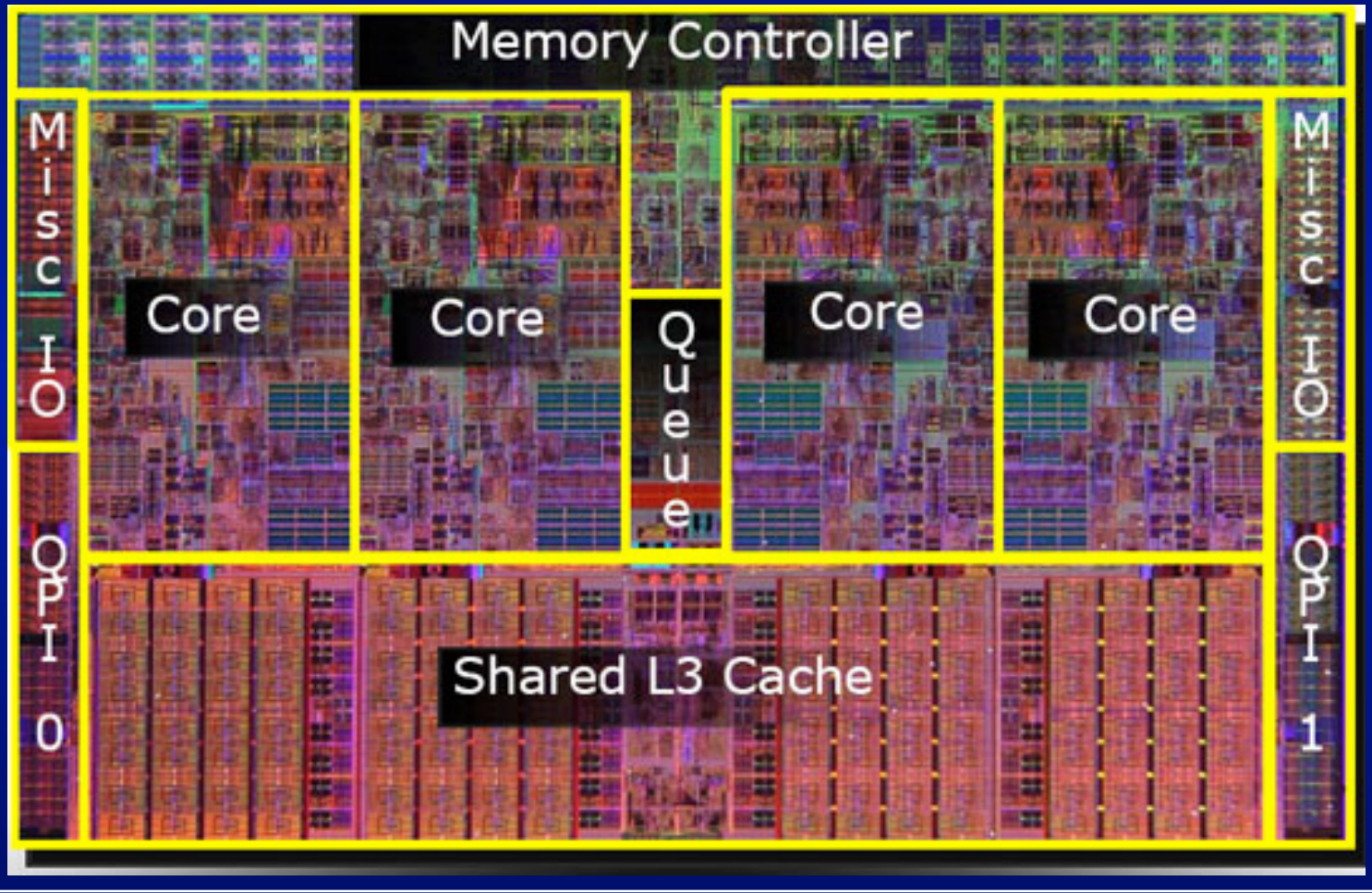
What is CUDA?

- *"Compute Unified Device Architecture"*
- CUDA is an API for programming NVIDIA GPUs.
- CUDA can be used with C/C++ or FORTRAN (we'll be working with the C interface).
- CUDA has a high-level/runtime API and a low-level/driver API.
  (We'll be working with the runtime API).
- OpenCL has similar syntax to the driver API, usable for GPUs and CPUs (and other devices)

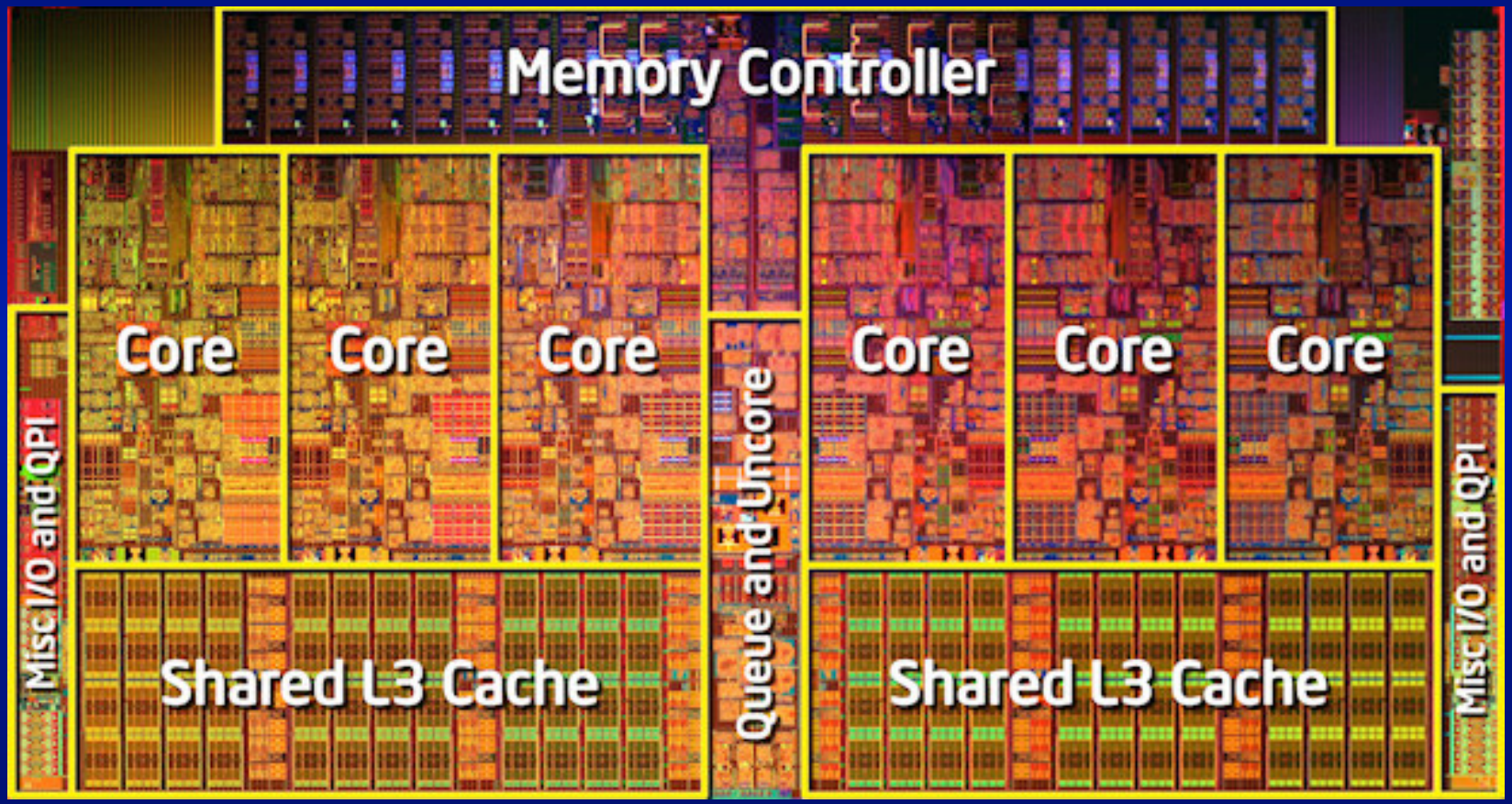
What is a CPU?

- CPU = Central Processing Unit
- A CPU is what we typically think of as the 'brain' part of the computer, or in the context of heteregenous computing, 'the decider'.
- Modern CPUs can have multiple cores
  - The Nehalem chips in Longhorn have four cores each
  - The Westmere chips in Lonestar have six cores each
- The CPU part of a node is the *host*
  - Processors and associated RAM

Intel Nehalem
(Longhorn nodes)



Intel Westmere
(Lonestar nodes)

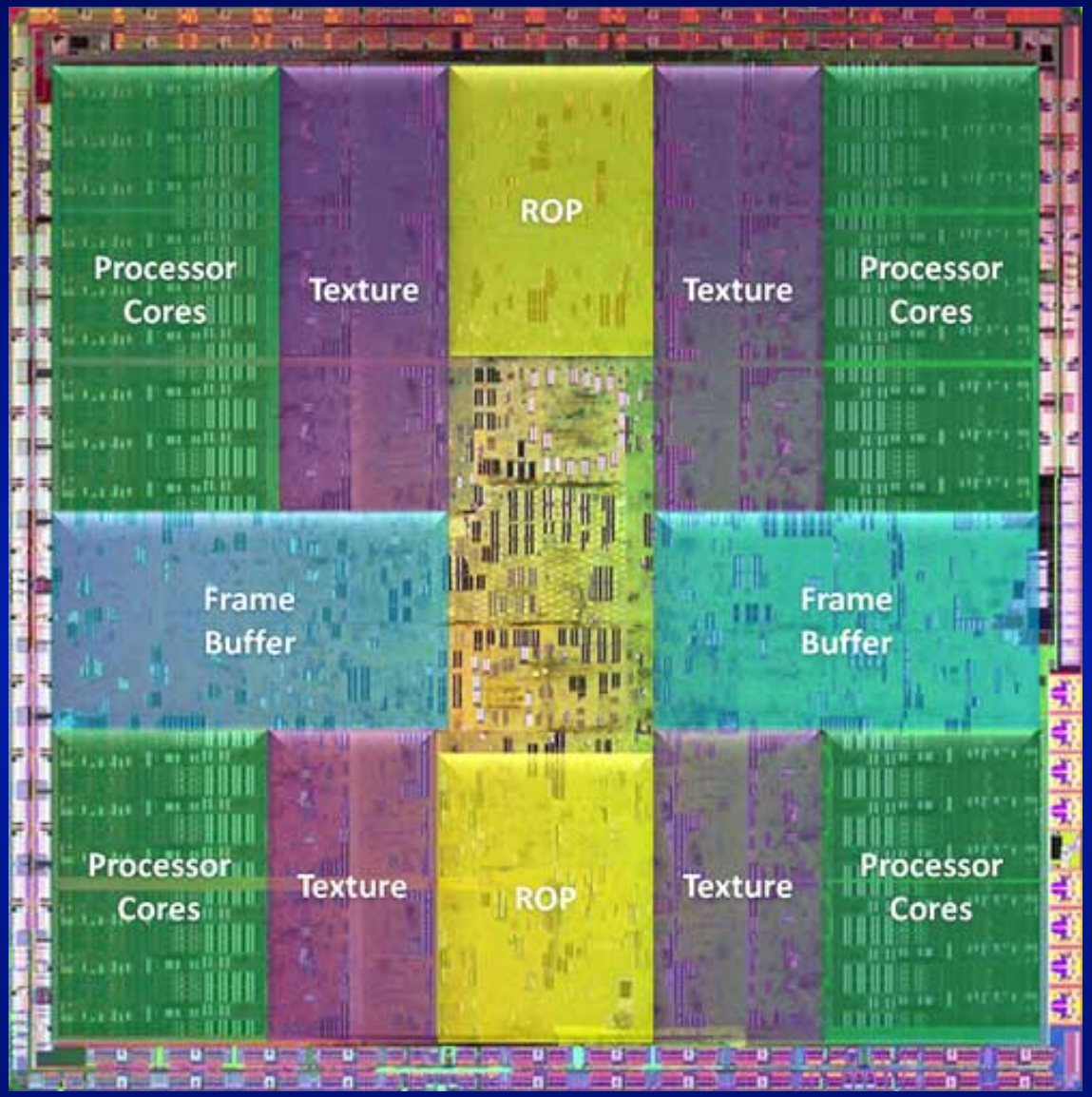


TACC

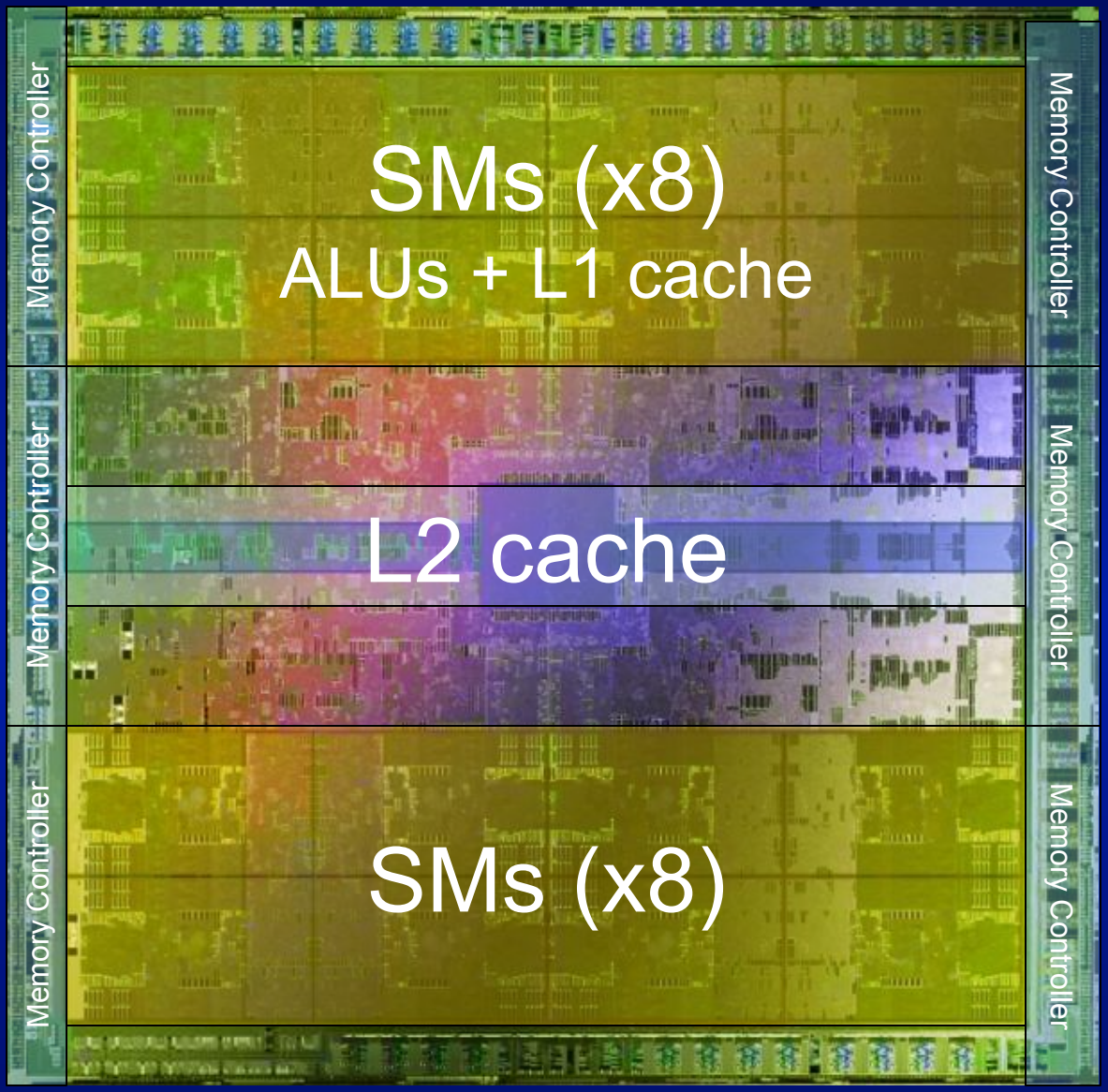
What is a GPU?

- GPU = Graphics Processing Unit
- A GPU is an extension or add-on to the CPU
- GPGPU = general purpose GPU
- Each GPU has many functional units
  (Streaming Multiprocessors or SMs in NVIDIA devices)
  - The Tesla-class devices on Longhorn have 30 SMs
  - The Fermi-class devices on Lonestar have 14 SMs
  - Each SM is like a CPU core, but much simpler logic
- The GPU parts of a node are *devices*
  - GPU SMs and associated GRAM (graphics RAM)

NVIDIA GT200 (Longhorn nodes)



NVIDIA GF100
Fermi
(Lonestar nodes)



TACC

Hardware Comparison
(Longhorn- and Lonestar-deployed versions)

	Nehalem E5540	Westmere X5680	Tesla Quadro FX 5800	Fermi Tesla M2070
Functional Units	4	6	30	14
Speed (GHz)	2.53	3.33	1.30	1.15
SIMD / SIMT width	4	4	8	32
Instruction Streams	16	24	240	448
Peak Bandwidth DRAM->Chip (GB/s)	35	35	102	150

What is a CUDA kernel?

- A CUDA kernel is a *function* that is called by the host or another CUDA kernel.
- A CUDA kernel executes on the *device*.
- A CUDA kernel is parallel code executed simultaneously by many threads.

- Mainly Tesla capability today (on Longhorn)
  - Some Fermi details, see the CUDA C Programming Guide for more

CPU vs. GPU characteristics

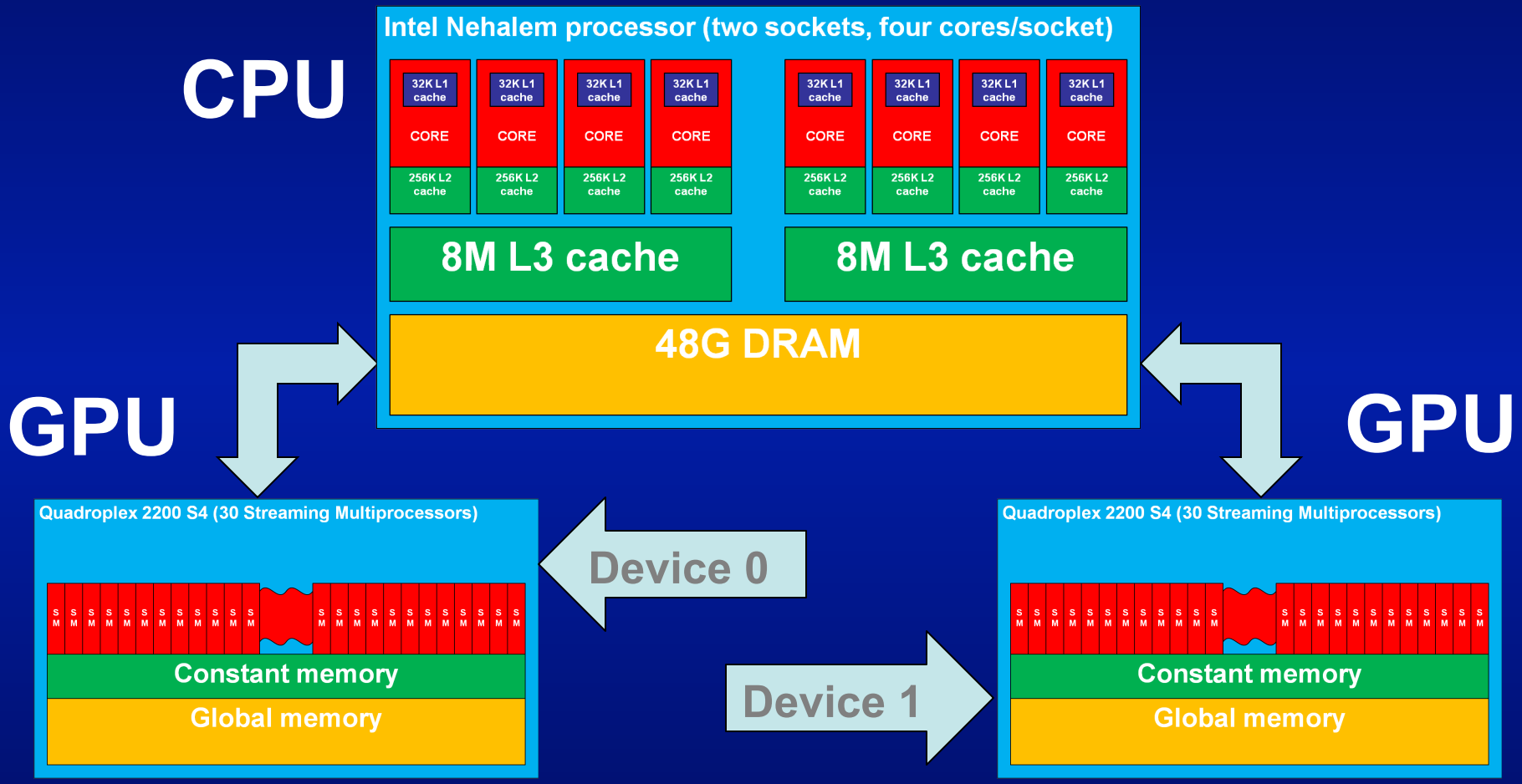
CPU

- Few computation cores
  - Supports many instruction streams, but keep few for performance
- More complex pipeline
  - Out-of-order processing
  - Deep (tens of stages)
  - Became simpler (Pentium 4 was complexity peak)
- Optimized for serial execution
  - SIMD units less so, but lower penalty for branching than GPU

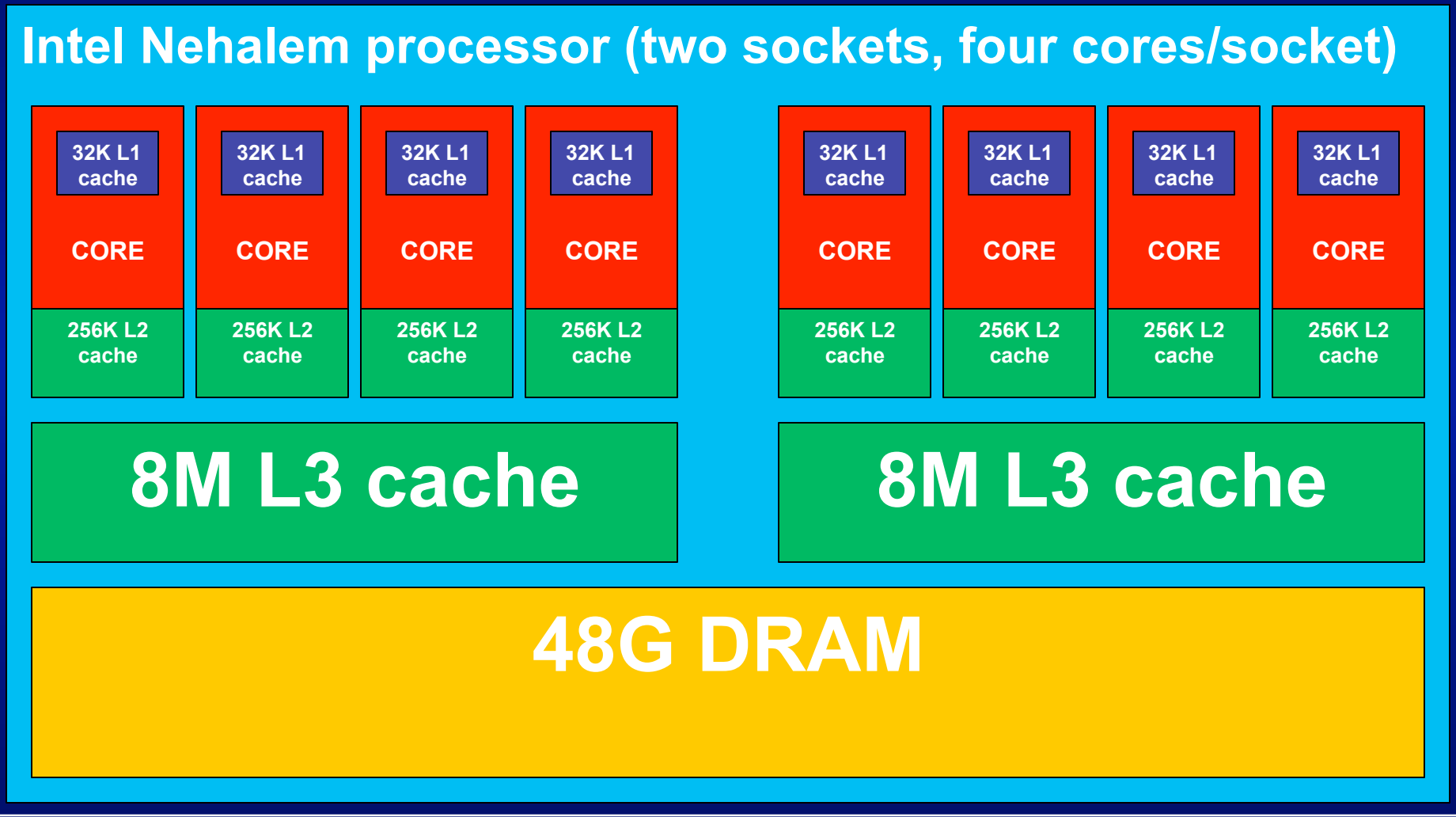
GPU

- Many computation cores
  - Few instruction streams
- Simple pipeline
  - In-order processing
  - Shallow (< 10 stages)
  - Became more complex
- Optimized for parallel execution
  - Potentially heavy penalty for branching

Longhorn architecture

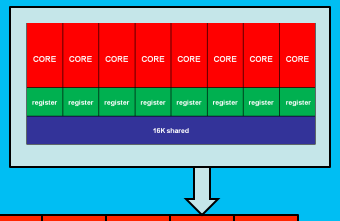


Longhorn CPU architecture



Longhorn GPU architecture

Quadro FX 5800 (30 Streaming Multiprocessors)

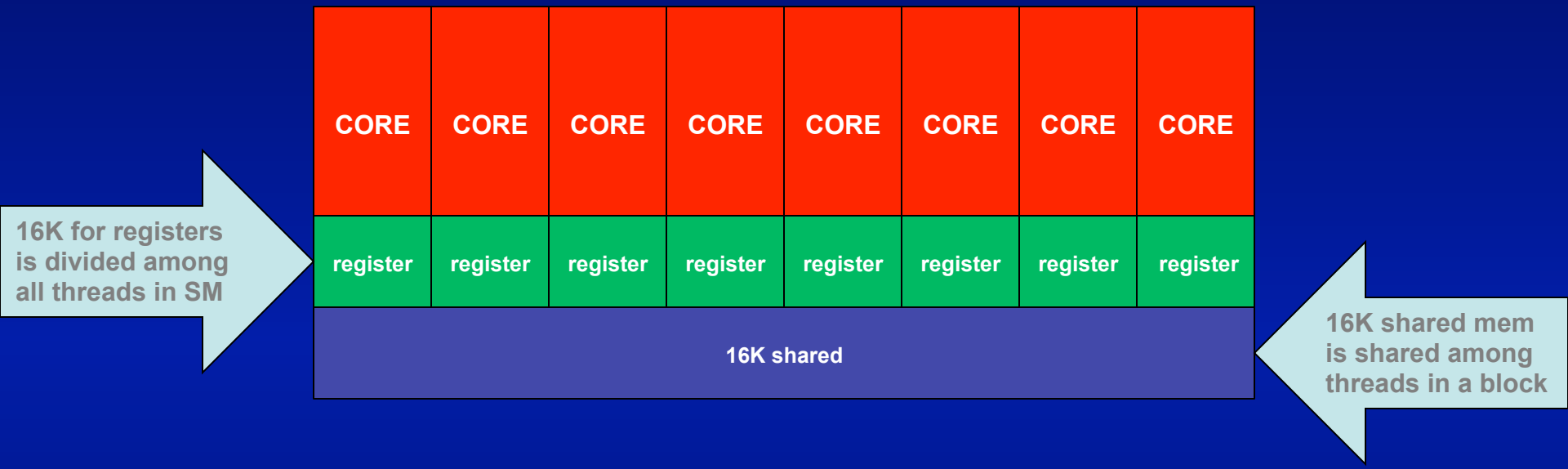


SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM SM

Constant memory

Global memory

Longhorn Streaming Multiprocessor



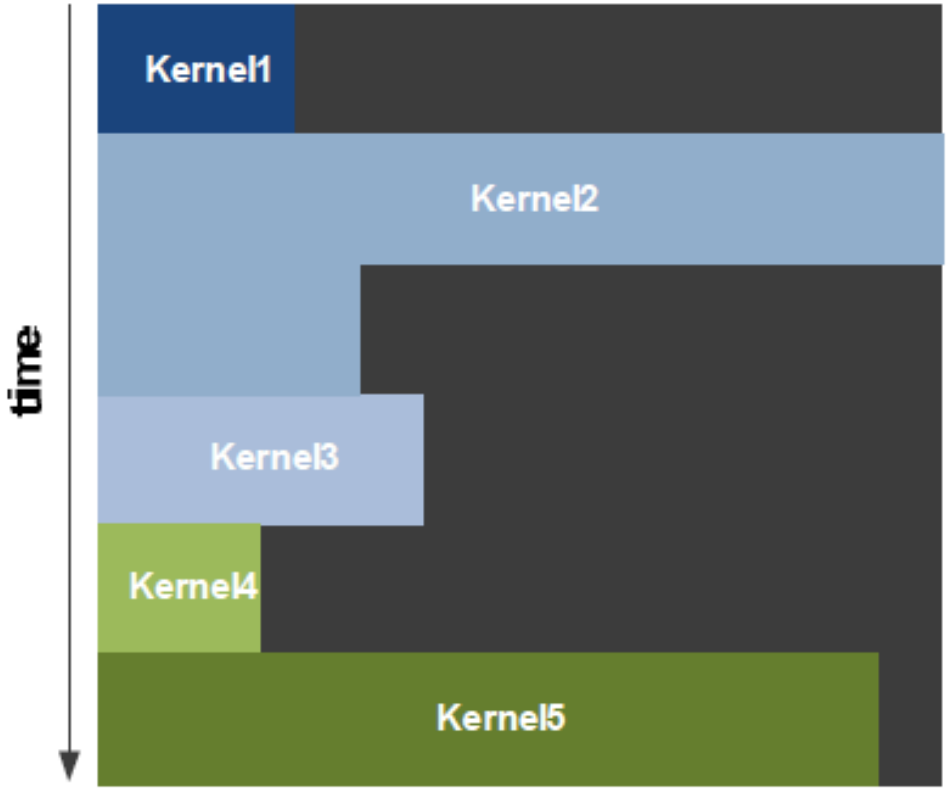
- A Streaming Multiprocessor has 8 “CUDA cores”
- A block of threads executes on one SM
- Blocks are divided into “warps” of 32 threads
  - Each block should have a multiple of 32 threads!
- Note: Lonestar’s GPUs have 32 CUDA cores per SM

Comparing Tesla versus Fermi

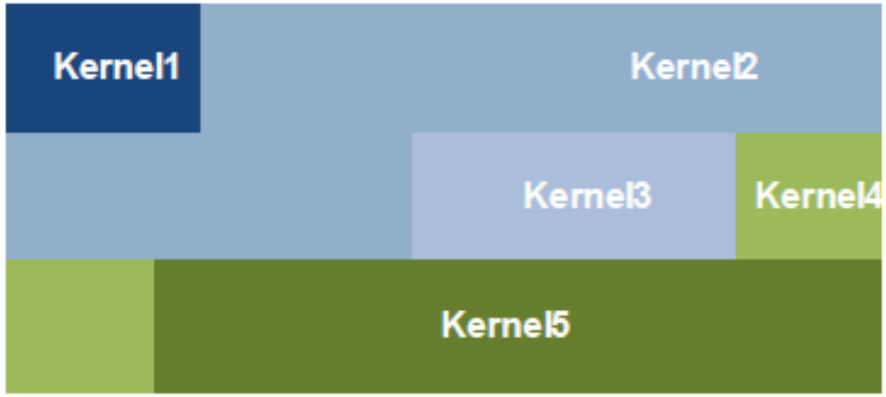
GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Kernel Execution

GT200
single kernel at a time



GF100
multiple kernels simultaneously



Up to one per SM

Other Important Changes in Fermi

- ECC (Error Correction Code) memory
  - Detects and corrects soft memory errors automatically
  - Parity bits reduce memory capacity: 6 GB -> 5.25 GB
  - Definite nod to HPC, but not enabled by default
- Faster Double-Precision
  - GT200 took 8x penalty, GF100 only 2x (same as CPUs)
- Faster context switching
  - 10x improvement makes multitasking practical
- IEEE 754-2008 (floating point standard) compliance

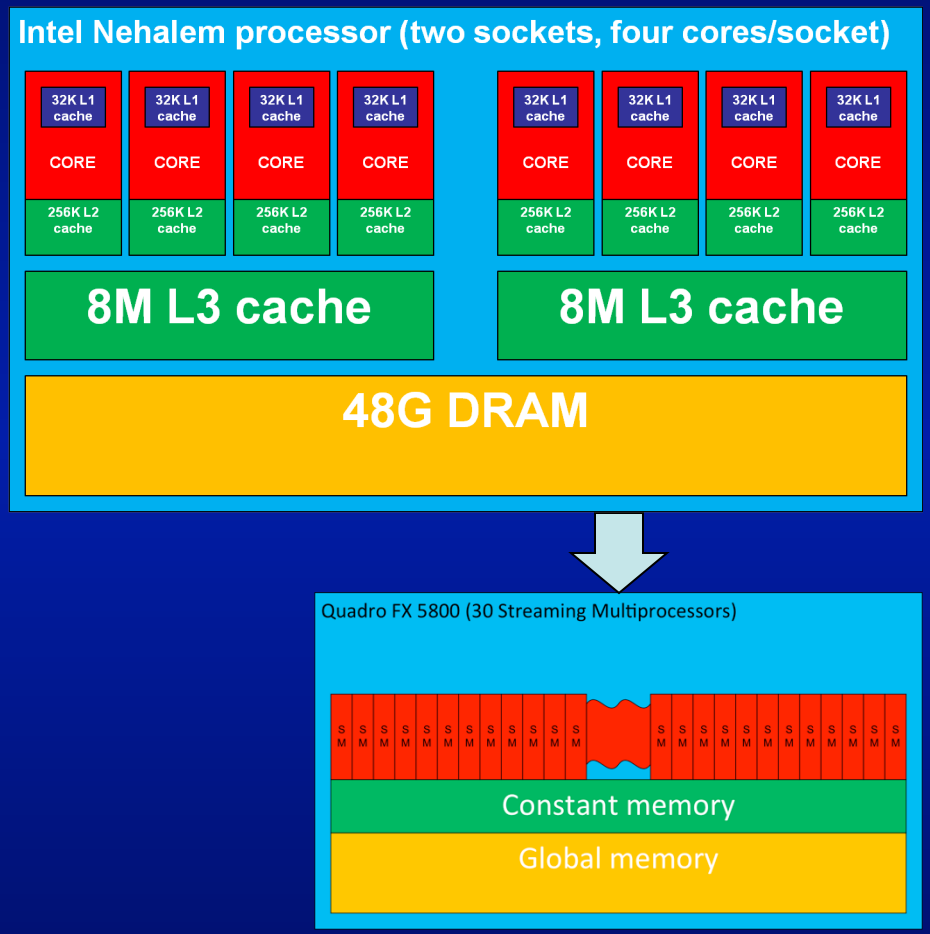
Other Important Changes in Fermi

- Faster atomic operations
  - Ensures protected access to data
  - More hardware support and unified L2 cache
  - 20x faster than GT200
- Unified host/device address space
  - Allows full implementation of pointers in C and C++
  - Supports device-size memory allocation
    - Possible, but expensive! Reuse buffers if possible.

The CUDA programming model

- Runtime API
- Driver API

- Host thread calls CUDA functions and launches CUDA kernels
- Flow control and synchronization are the responsibility of the host thread.



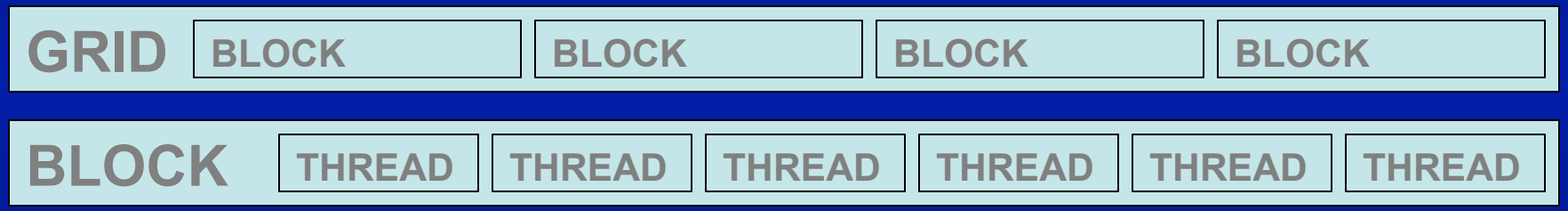
CUDA Programming Model

- Data is decomposed into a hierarchy of grids and threads.

- A Grid is composed of multiple thread blocks
- A Grid can be either 1-D or 2-D (or 3-D on Fermis)

CUDA Programming Model

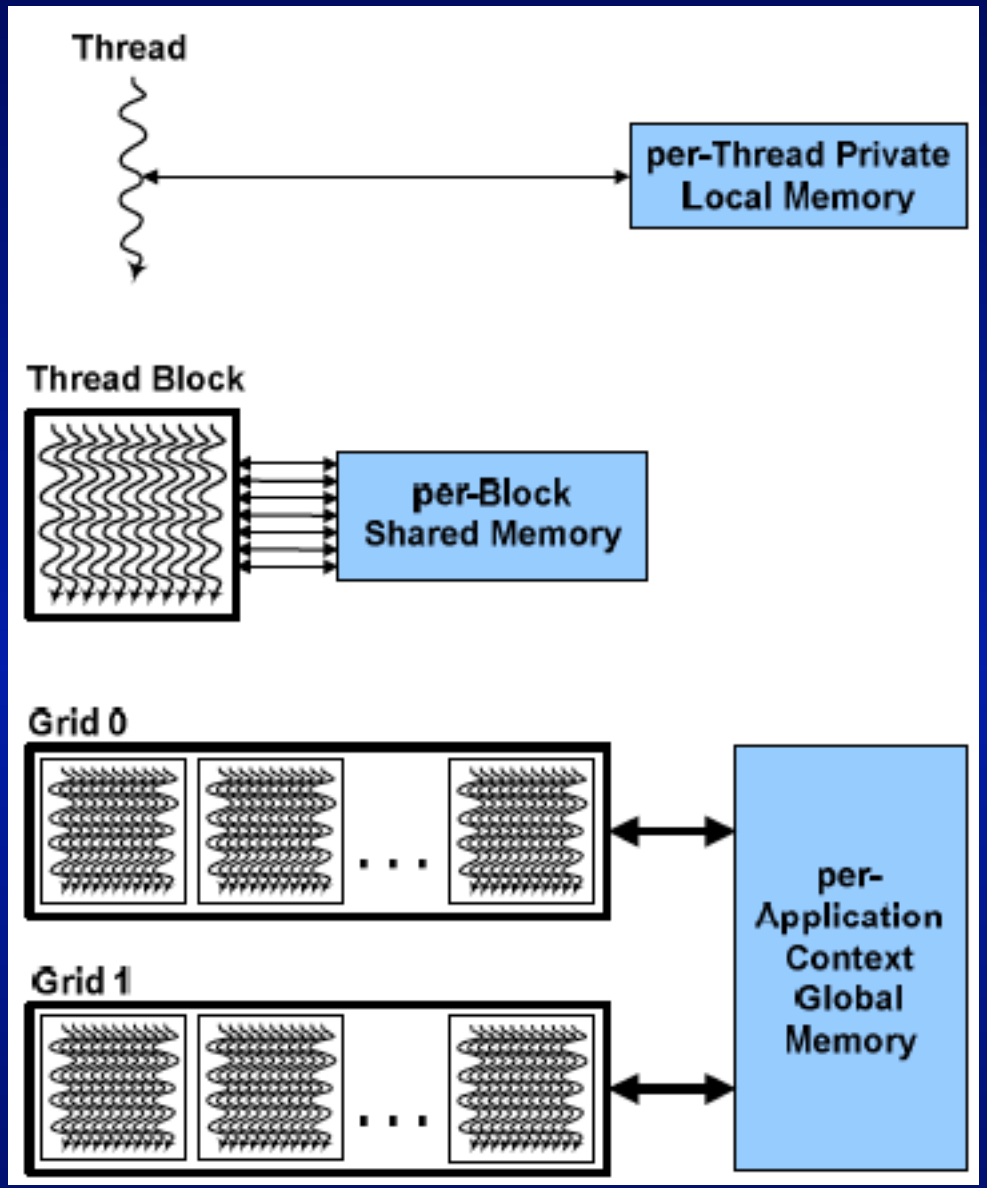
- A thread block contains many threads.
- A thread block can be 1-D, 2-D, or 3-D.



Hierarchy of Threads, Blocks, and Grids

- Thread -> One SIMT Lane -> Registers
- Thread Block -> SM -> Shared Memory (& L1 Cache on Fermi)
- Grid -> all SMs -> Global Memory (& L2 Cache on Fermi)

Threads control access to all memories



Compiler Flags:

- Debugging:
  - -g generates debugging information for host
  - -G generates debugging information for device
  - Use `cuda-gdb` to debug both host and device code
  - More info: `$TACC_CUDA_DIR/doc/cuda-gdb.pdf`
- Profiling
  - Use NVIDIA computeprof to profile device code
  - More info: `$TACC_CUDA_DIR/computeprof/doc`

Example: Free Volume *via* Widom insertion

$$F(N,V,T) = -kT \ln Z(N,V,T)$$

$$Z = \sum_i e^{-E_i / kT}$$

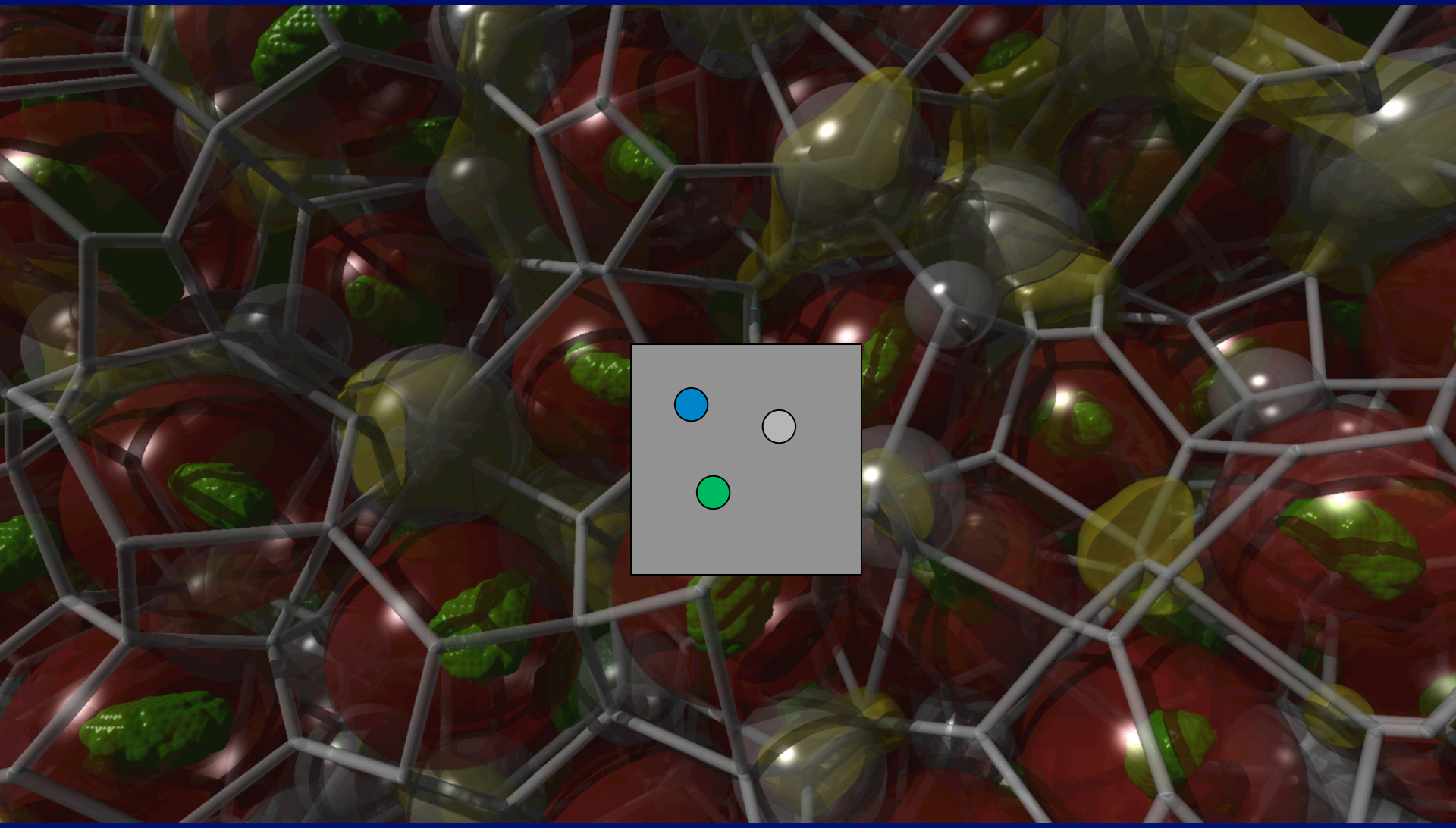
$$\mu = \left.\frac{\partial F}{\partial N}\right|_{T,V} = -kT \left.\frac{\partial \ln Z}{\partial N}\right|_{T,V} = -kT \ln \frac{Z(N+1,V,T)}{Z(N,V,T)} = -kT \ln B_{insertion}$$

$$B_{insertion} = \left\langle e^{-\frac{\psi_{insertion}}{kT}} \right\rangle$$

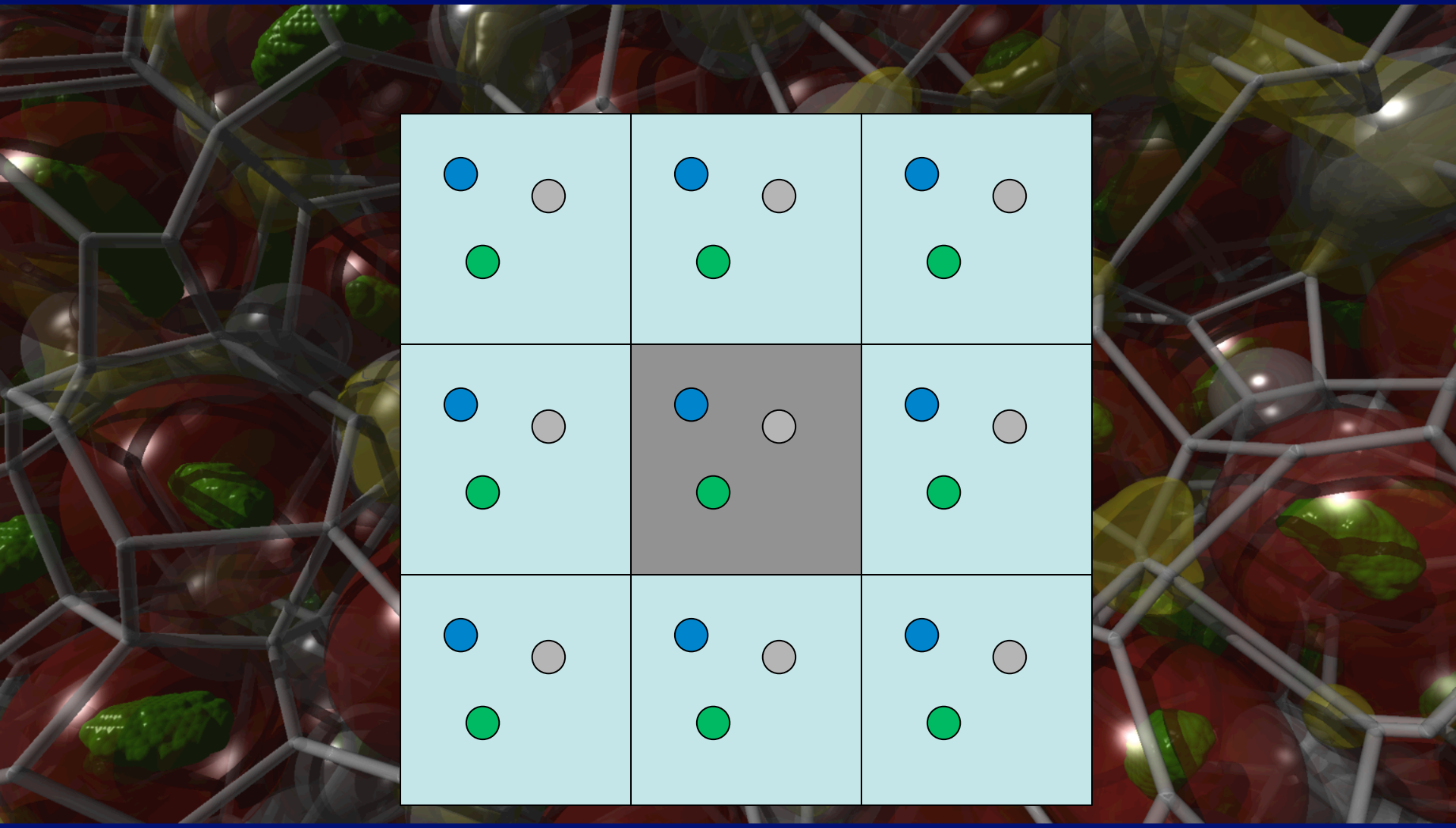
$$FVI(x,y,z,t) \equiv e^{-\frac{\psi_{repulsive}}{kT}}$$

Insertion parameter as a free volume metric:

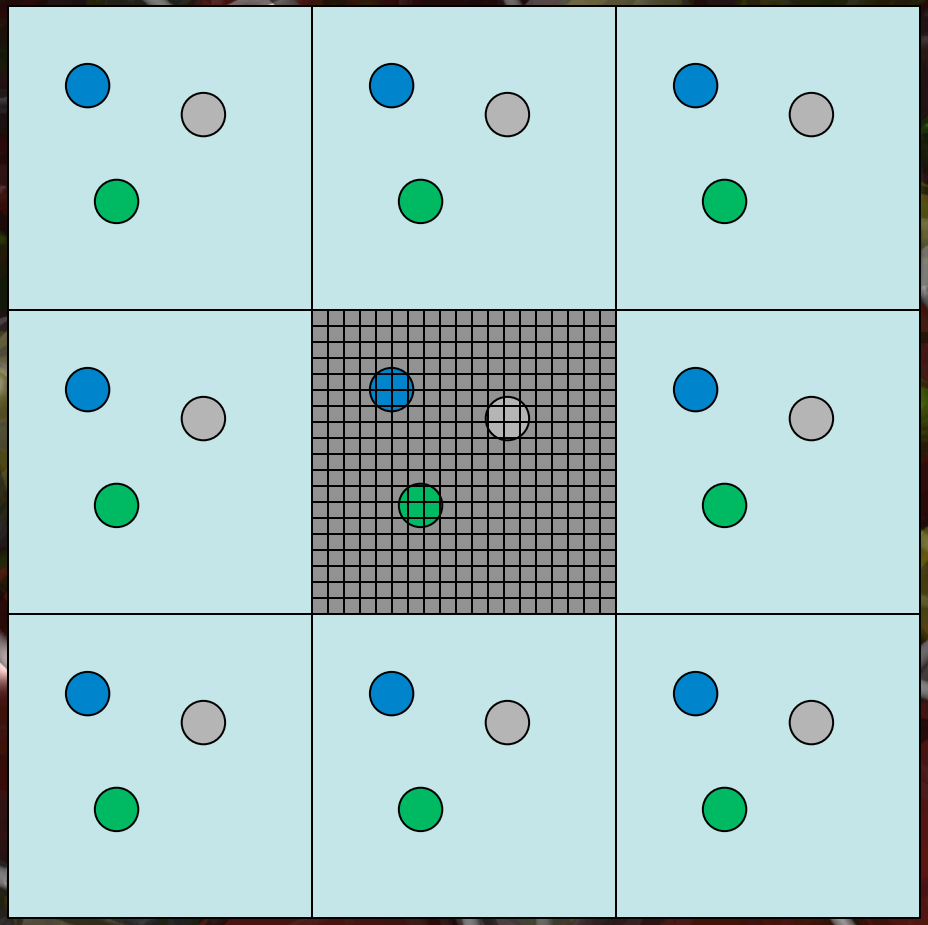
- Works with *any* forcefield model
- Well-behaved function, bounded between [0..1)
- Represents solubility
- Continuity with statistical mechanics
- Easily computable (particularly with GPUs)
- Defined over *all* of space



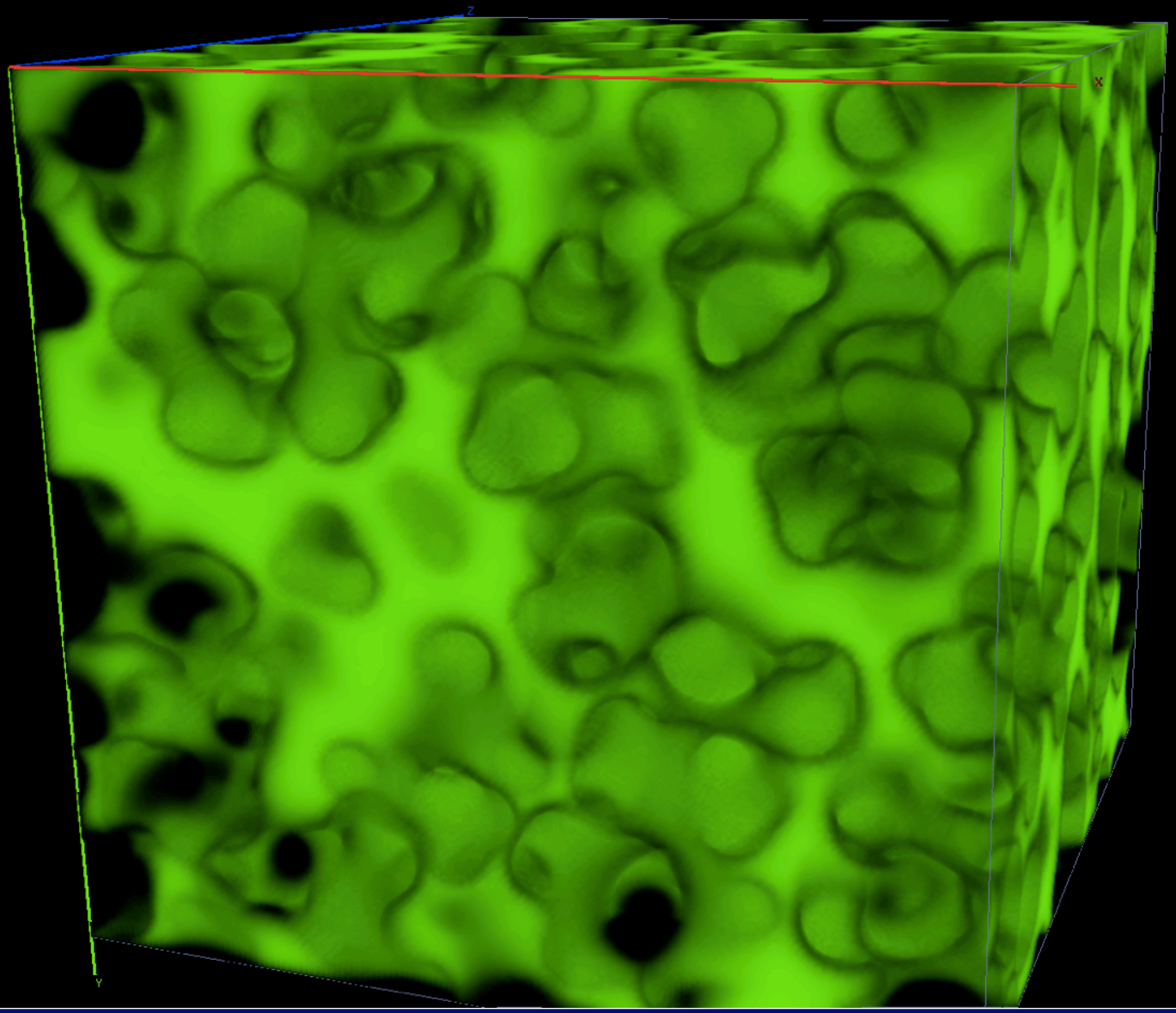
A GPU example: Sampling free volume



A GPU example: Sampling free volume



A GPU example: Sampling free volume



TACC

polydimethylsiloxane

Why use GPUs for this?

- Feasability of calculation
  - Widom first formulated insertion method in 1963. For most systems, calculation was not feasible.
  - 49 years of increases in computing power / ~4 billion-fold increase per Moore's Law (1965)
  - With the current formulation, it now takes ~one hour on a single NVIDIA Tesla M2070 (R_{peak} = 1TFLOP single precision) to calculate energies in a system of 1600 atoms at 1 billion (*1024 x 1024 x 1024*) grid points.
- GPUs excel at performing the same calculation for different values of x, y, z. For example:
  - rendering an image
  - Monte Carlo sampling
  - Stencil calculations

Lab I: My first CUDA program

- Reads from standard input
- Converts each line of text received into ALL CAPITALS
- Prints the converted line of text to standard output.

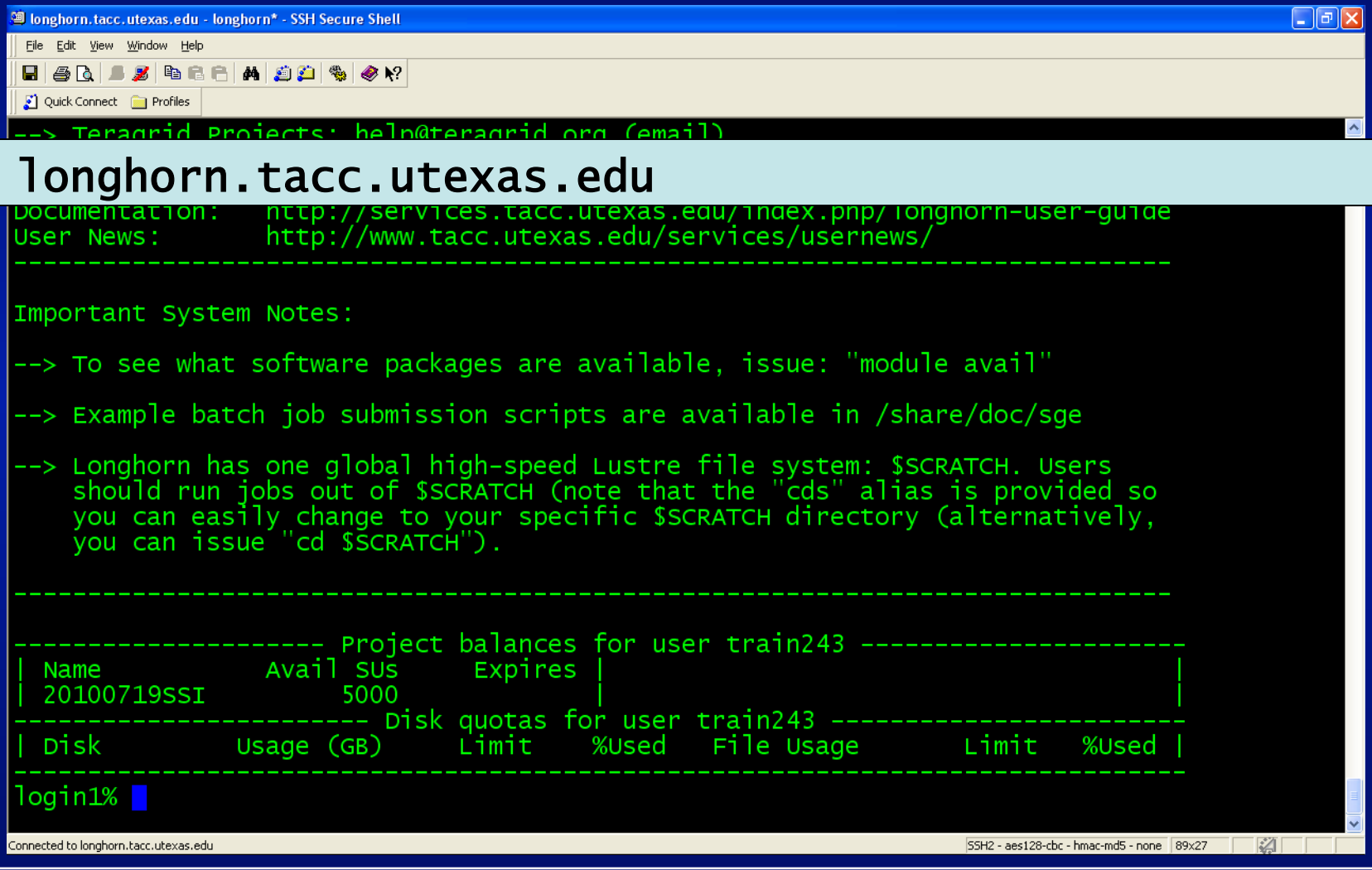
Lab I: My first CUDA program

- Compile the program
- Run the program interactively
- Run the program with a large text file
- Time program execution
- Compile program again for device emulation
- Time program execution for device emulation

Setting up the lab

- Login
- Unpack the archive
- Submit the batch job to reserve a node
- Retrieve the node name
- Connect to your node

Log in to longhorn.tacc.utexas.edu

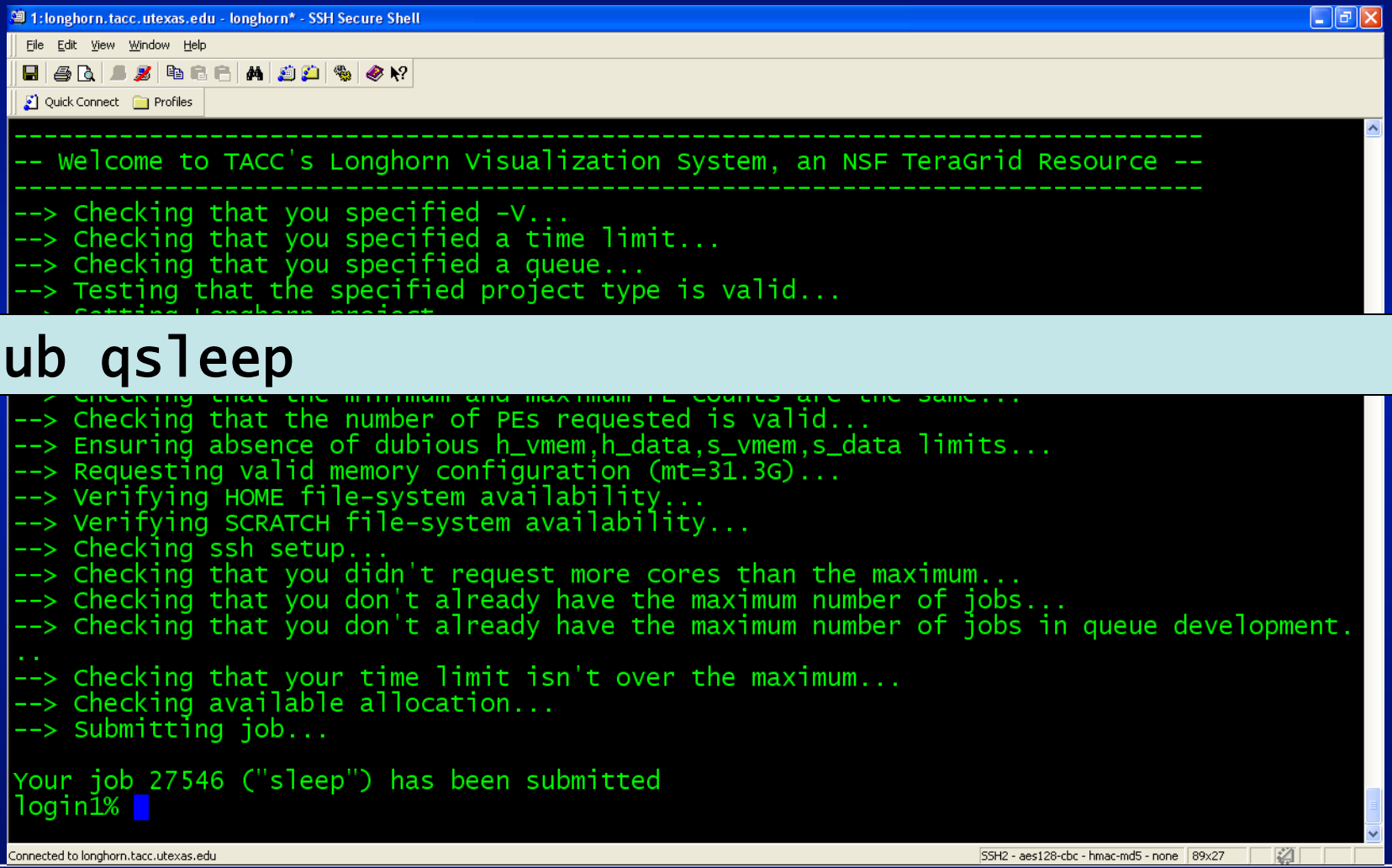


Copy and unpack the archive

```
cp ~train00/cuda4hpc.tar .
tar -xvf cuda4hpc.tar
cd cuda4hpc
```

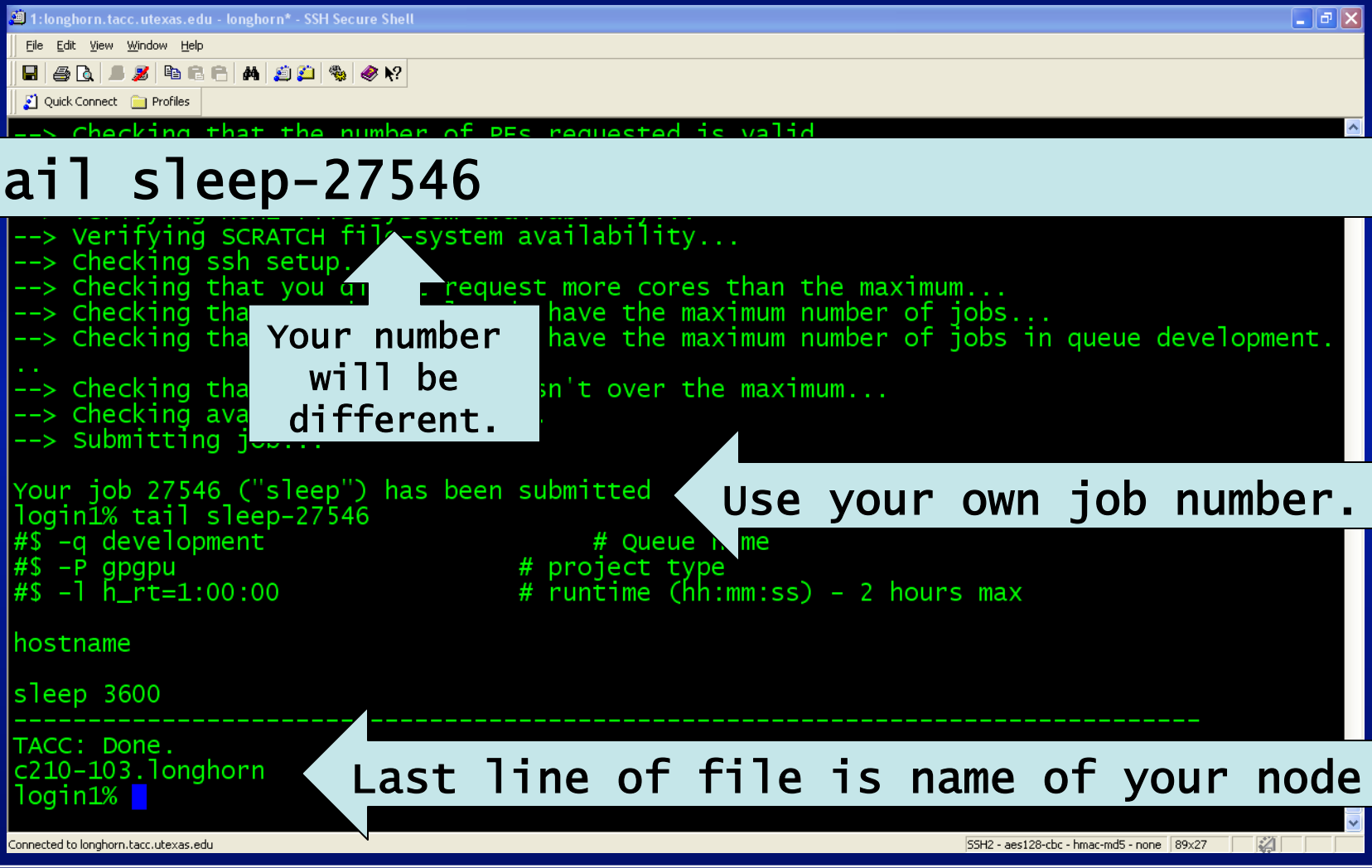
```
---------------------- Project balances for user train243 ----------------------
| Name           Avail SUs     Expires |
| 20100719SSI         5000             |
------------------------ Disk quotas for user train243 -------------------------
| Disk         Usage (GB)     Limit    %Used   File Usage       Limit   %Used
-------------------------------------------------------------------------------
login1% cp ~train00/cuda4hpc.tar .
login1% tar -xvf cuda4hpc.tar
./cuda4hpc/Makefile
./cuda4hpc/printDeviceInfo.cu
./cuda4hpc/processDistributions.cu
./cuda4hpc/qsleep
./cuda4hpc/shiftUpperCase.cu
./cuda4hpc/War_and_Peace.txt
login1% cd cuda4hpc
login1%
```

Submit the batch job to reserve a node

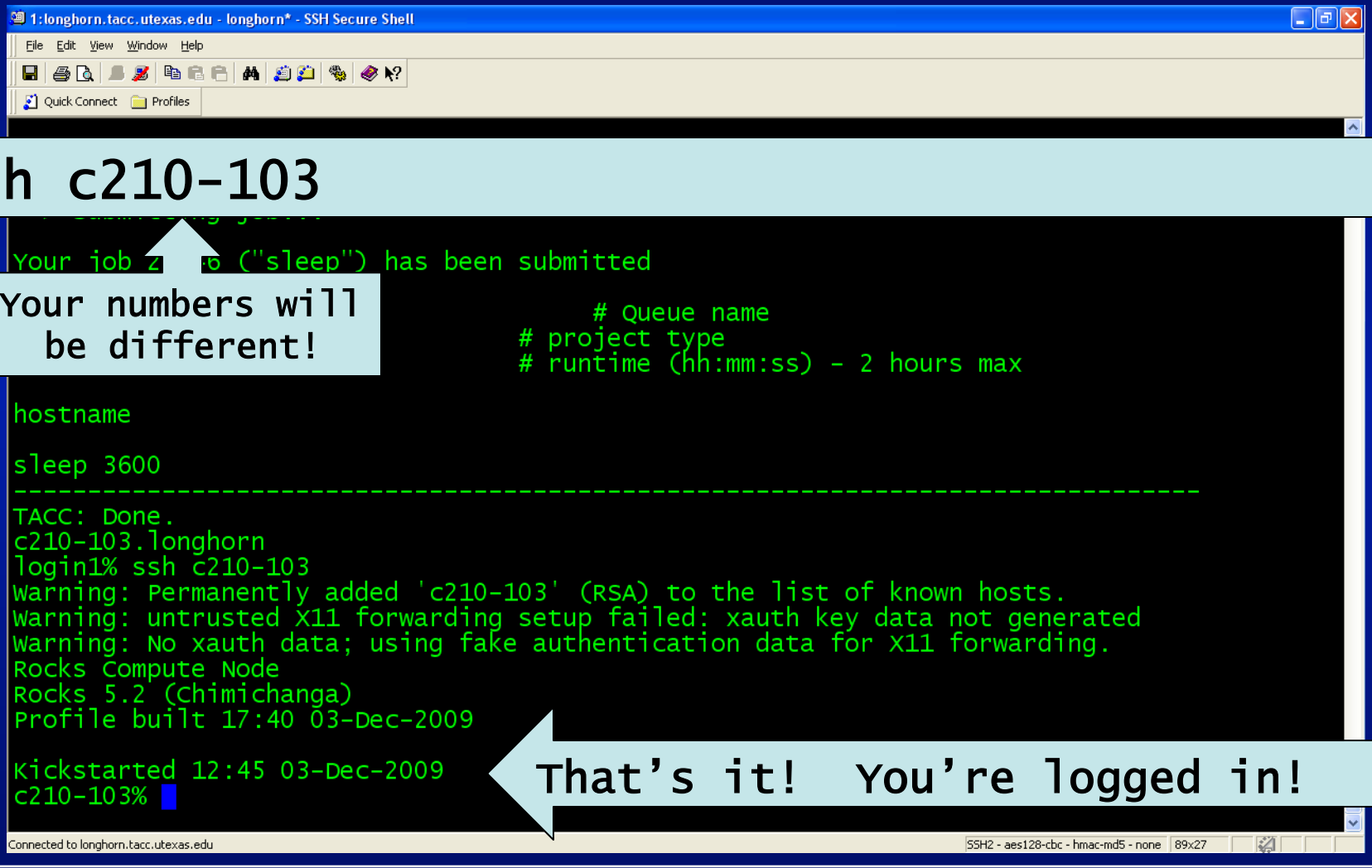


`qsub qsleep`

Retrieve the name of your node



Connect to your node



Examine source code

1:longhorn.tacc.utexas.edu - longhorn* - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

```
c210-103% cd cuda4hpc
c210-103% cat shift.cu
```

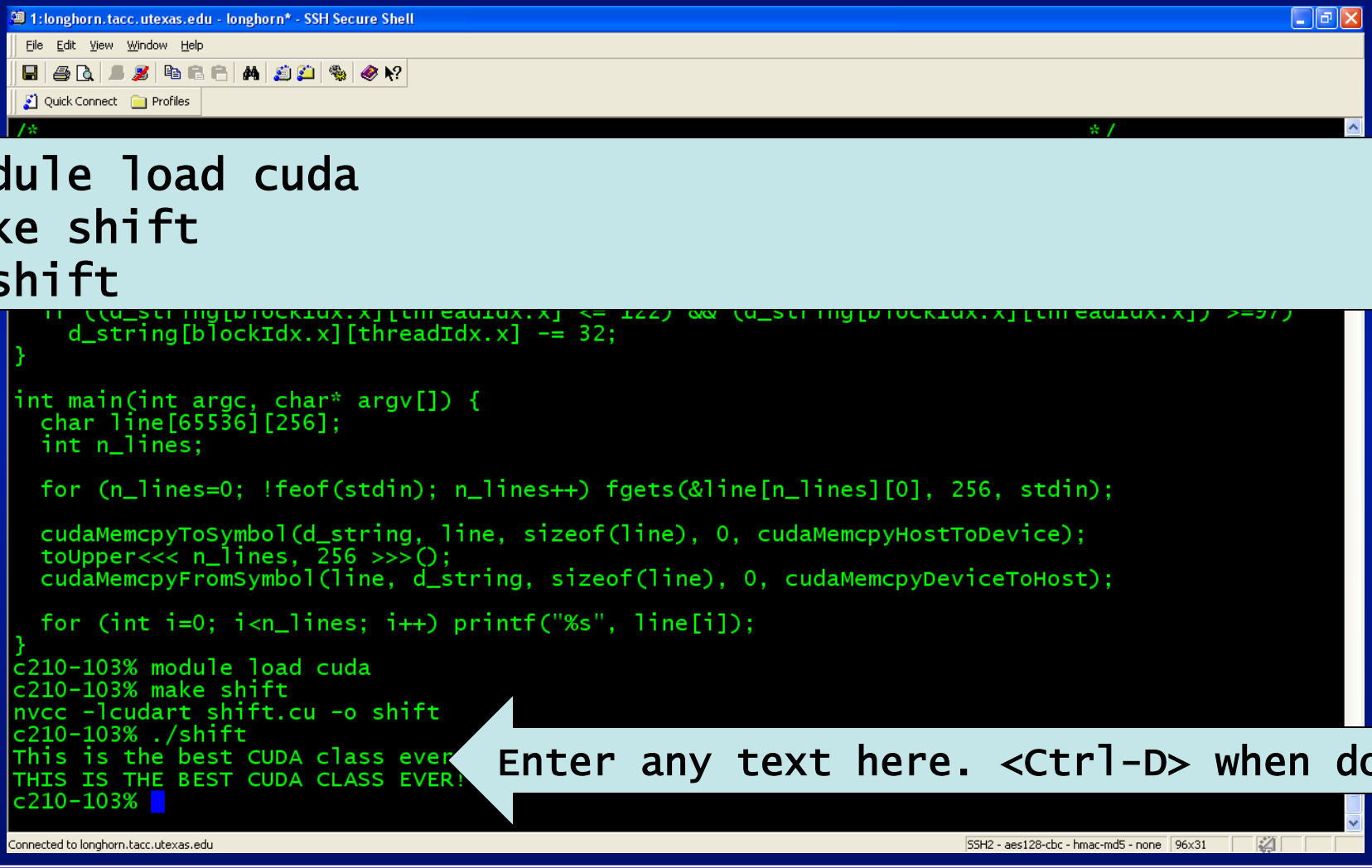
```
cd cuda4hpc
cat shift.cu
```

```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122) && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
  char line[65536][256];
  int n_lines;

  for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

  cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
  toUpper<<< n_lines, 256 >>>();
  cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

  for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
c210-103%
```

Connected to longhorn.tacc.utexas.edu SSH2 - aes128-cbc - hmac-md5 - none 96x31

TACC

Compile and run the program



Walking through the program

```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122)
    && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
  char line[65536][256];
  int n_lines;

  for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

  cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
  toUpper<<< n_lines, 256 >>>();
  cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

  for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```

Read text from standard input

```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122)
    && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
  char line[65536][256];
  int n_lines;

  for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

  cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
  toUpper<<< n_lines, 256 >>>();
  cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

  for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```

TACC

Copy data to device

```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122)
    && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
  char line[65536][256];
  int n_lines;

  for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

  cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
  toUpper<<< n_lines, 256 >>>();
  cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

  for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```

TACC

Launch the CUDA kernel

```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122)
    && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
  char line[65536][256];
  int n_lines;

  for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

  cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
  toUpper<<< n_lines, 256 >>>();
  cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

  for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```

TACC

Copy the result from the device

```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122)
    && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
  char line[65536][256];
  int n_lines;

  for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

  cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
  toUpper<<< n_lines, 256 >>>();
  cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

  for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```

TACC

Print the result

```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122)
    && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
  char line[65536][256];
  int n_lines;

  for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

  cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
  toUpper<<< n_lines, 256 >>>();
  cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

  for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```

A CUDA kernel: toUpper()

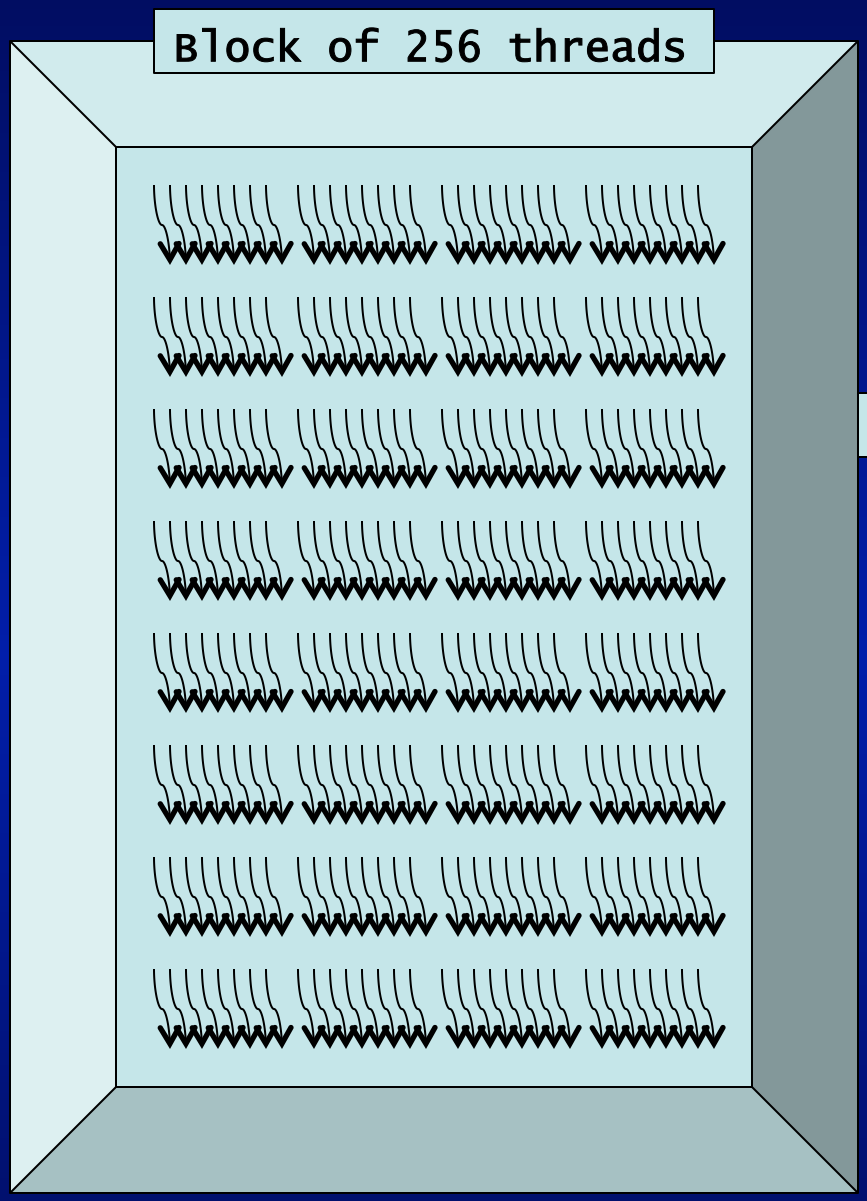
```
__global__ void toUpper() {
  if ((d_string[blockIdx.x][threadIdx.x] <= 122)
    && (d_string[blockIdx.x][threadIdx.x]) >=97)
    d_string[blockIdx.x][threadIdx.x] -= 32;
}
```

- This simple CUDA kernel changes all lower case letters in a string to upper case.
- Each letter is examined simultaneously by a separate thread.
- The built-in variable `threadIdx.x` tells us which thread is referenced / which character to examine.

Launching a CUDA kernel:

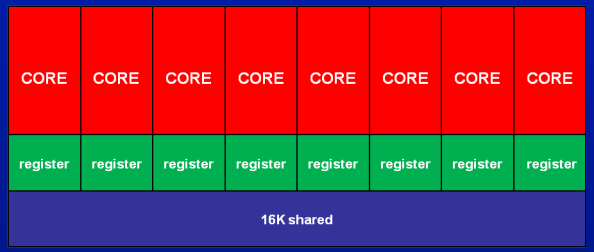
```
toUpper<<< n_lines, 256 >>>();
```

- CUDA kernels require parameters for grid size (`n_lines`) and block size (`256`)
- Zero or more function arguments can be passed
- Multidimensional grids and blocks must be declared of type `dim3`



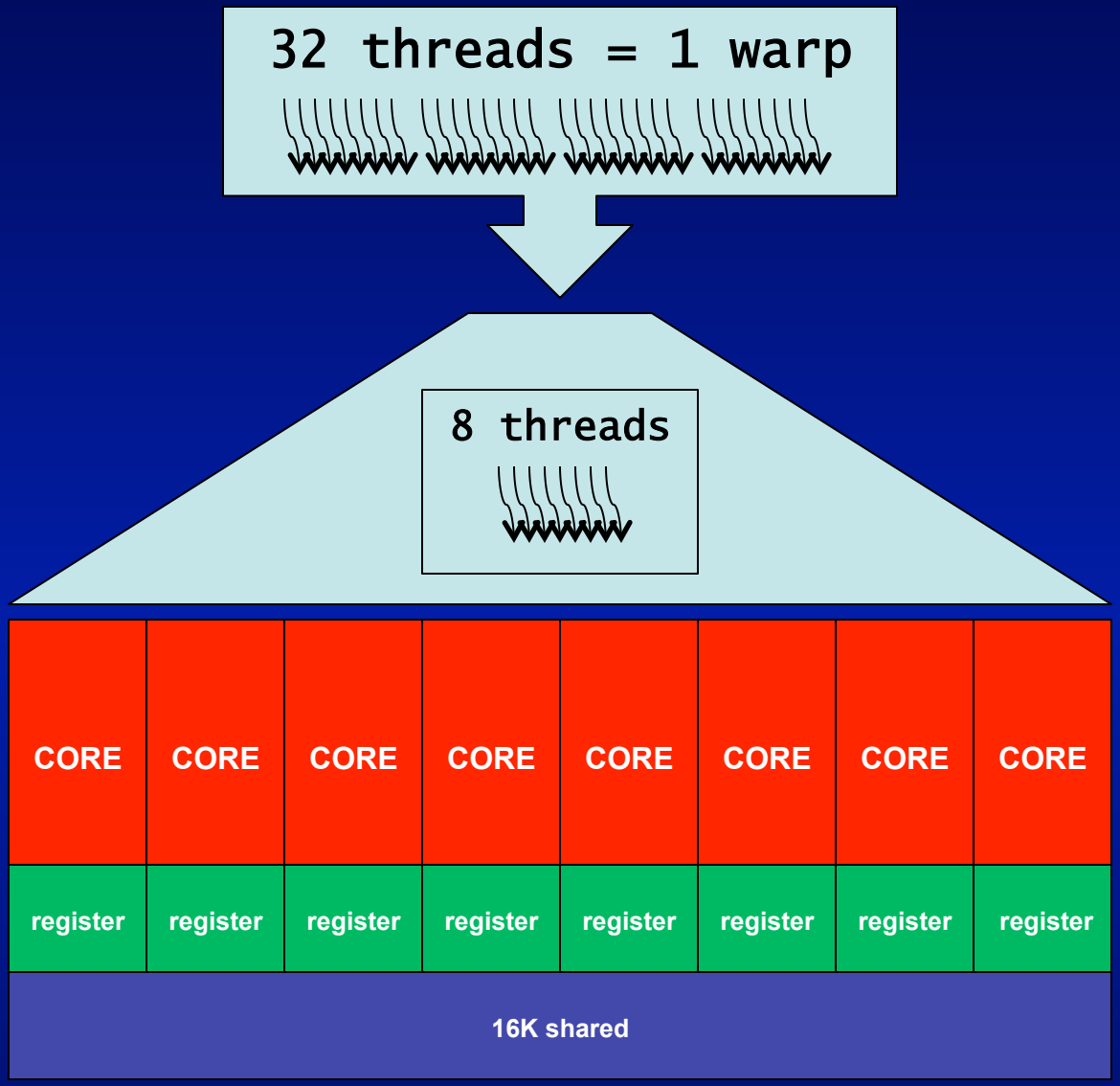
A set of threads (a WARP) is assigned to a multiprocessor.

32 threads at a time



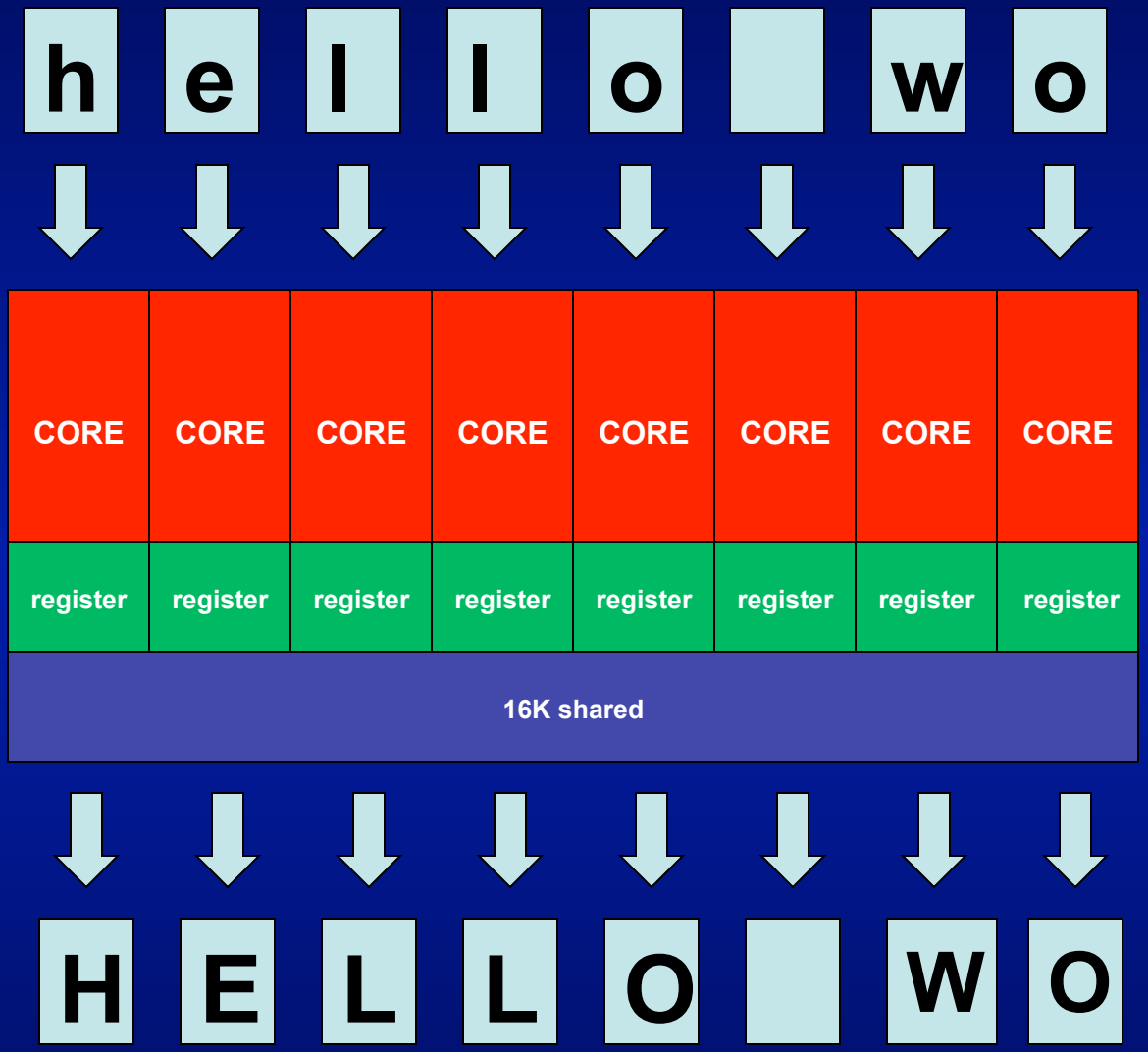
Streaming Multiprocessor

TACC



One warp executes in four cycles of eight threads.

TACC


h e l l o w o
CORE CORE CORE CORE CORE CORE CORE CORE
register register register register register register register register
16K shared
H E L L O W O

Summary

- Using GPU acceleration is beneficial only if there is enough work for the GPU to do!
  - Must amortize cost of data transfer between device and host
  - Can use streams or zero-copy transfer to improve performance (see CUDA Programming Guide for details)
- Best if there is (1) a lot of data to process, and (2) a lot of work to do for each piece of data
  - Regular distribution of work is best
  - Few conditionals, lots of FLOPs

Additional reading

- http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf