Introduction to GPU Programming

Volodymyr (Vlad) Kindratenko

Innovative Systems Laboratory @ NCSA

Institute for Advanced Computing Applications and Technologies (IACAT)

Tutorial Goals

- Become familiar with NVIDIA GPU architecture
- Become familiar with the NVIDIA GPU application development flow
- Be able to write and run simple NVIDIA GPU kernels in CUDA
- Be aware of performance limiting factors and understand performance tuning strategies

Schedule

- 9:00-11:00 part I (introduction)
- 11:00-11:15 break
- 11:15-13:00 part II (examples)
- 13:00-14:00 lunch
- 14:00-16:00 part III (advanced topics)
- 16:00-16:15 break
- 16:15-18:00 part IV (lab time)

Part I

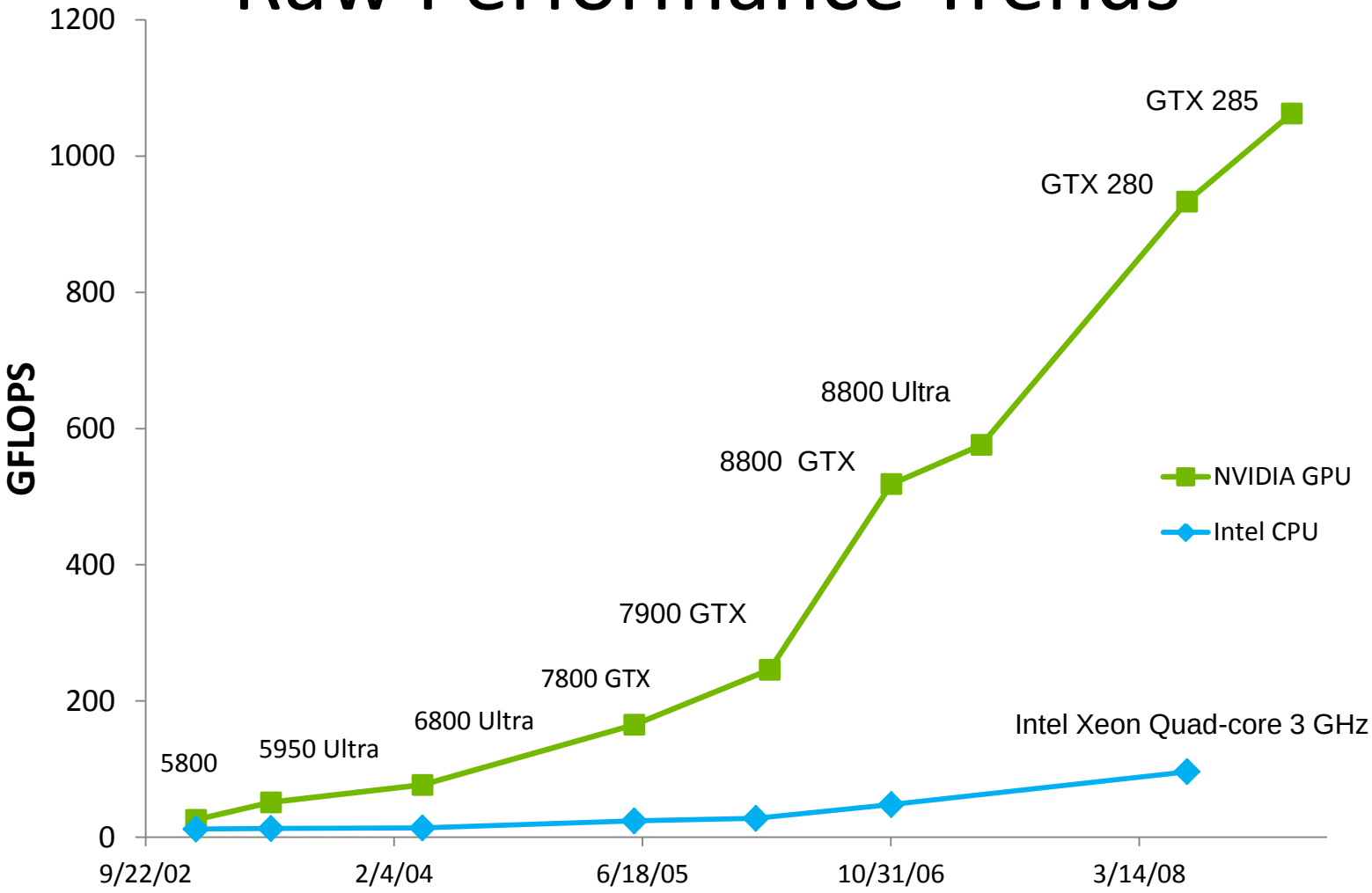
- Introduction
- Hands-on: getting started with NCSA GPU cluster
- Anatomy of a GPU application

- Break

Introduction

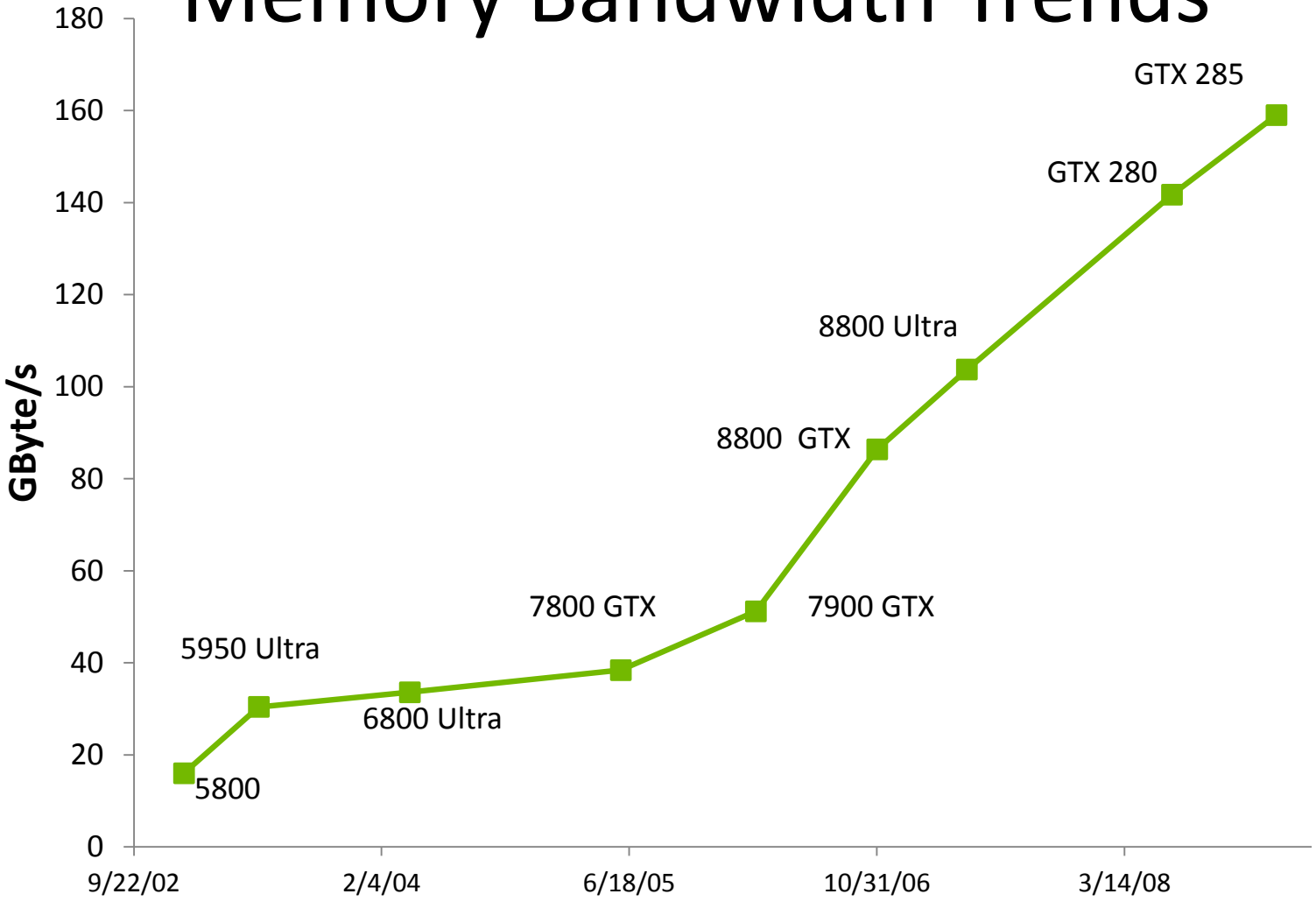
- Why use Graphics Processing Units (GPUs) for general-purpose computing
- Modern GPU architecture
  - NVIDIA
- GPU programming overview
  - Libraries
  - CUDA C
  - OpenCL
  - PGI x64+GPU

Why GPUs?
Raw Performance Trends



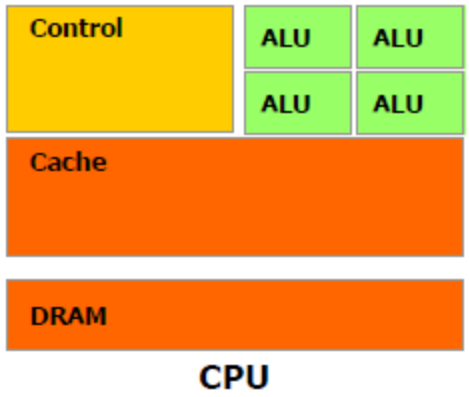
Graph is courtesy of NVIDIA

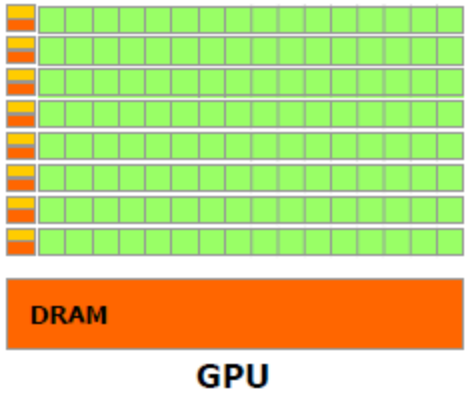
Why GPUs?
Memory Bandwidth Trends

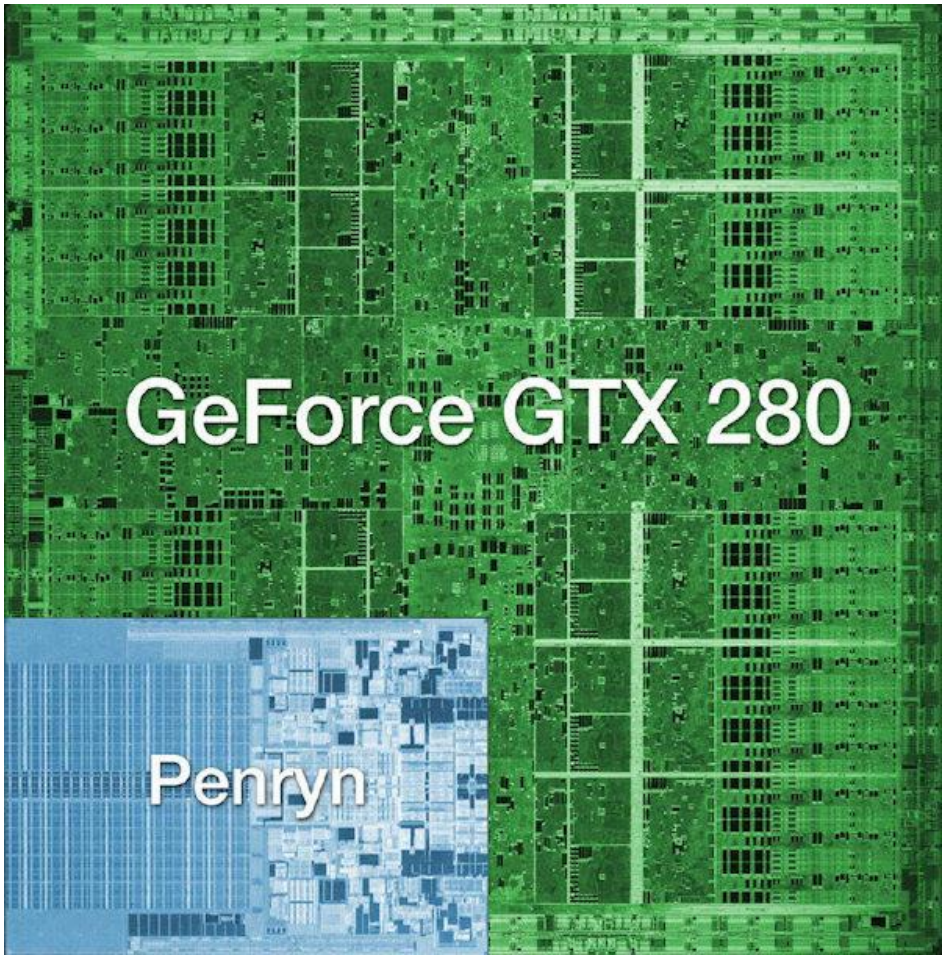


Graph is courtesy of NVIDIA

GPU vs. CPU Silicon Use







Graph is courtesy of NVIDIA

NVIDIA GPU Architecture

- A scalable array of multithreaded Streaming Multiprocessors (SMs), each SM consists of
  - 8 Scalar Processor (SP) cores
  - 2 special function units for transcendentals
  - A multithreaded instruction unit
  - On-chip shared memory
- GDDR3 SDRAM
- PCIe interface

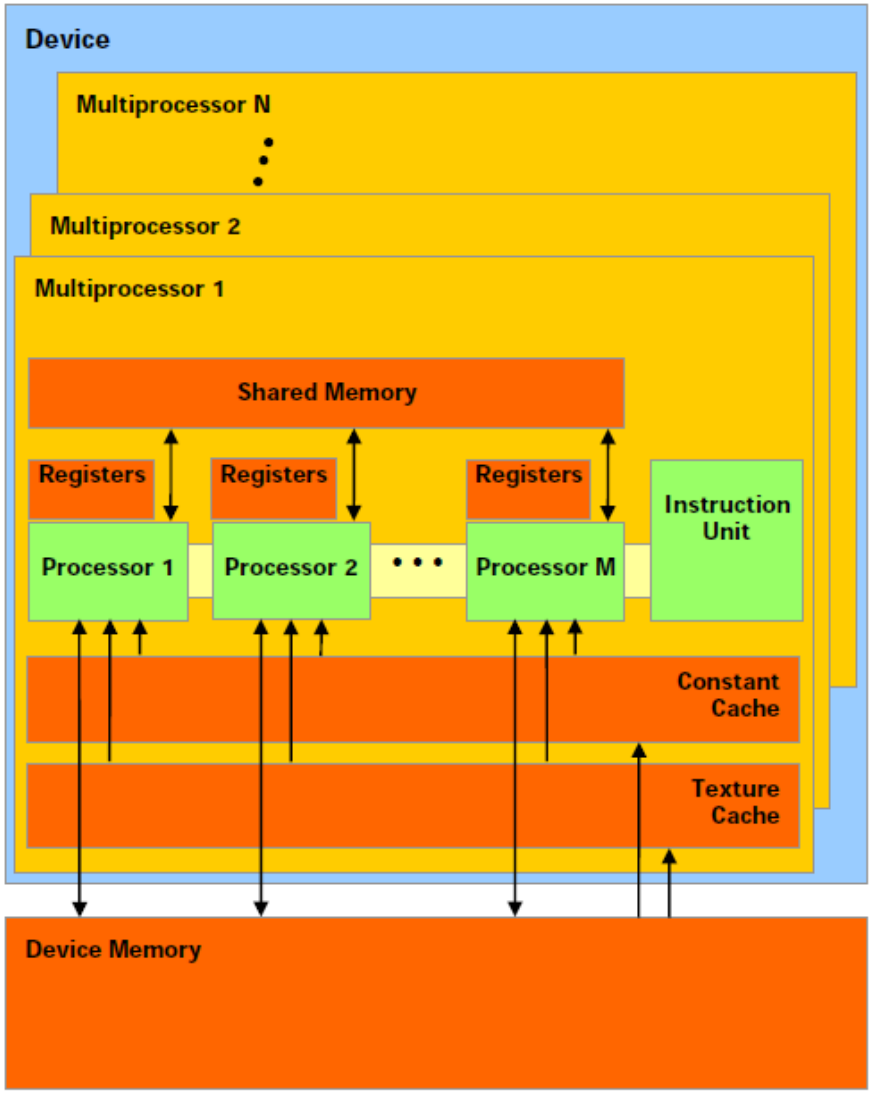
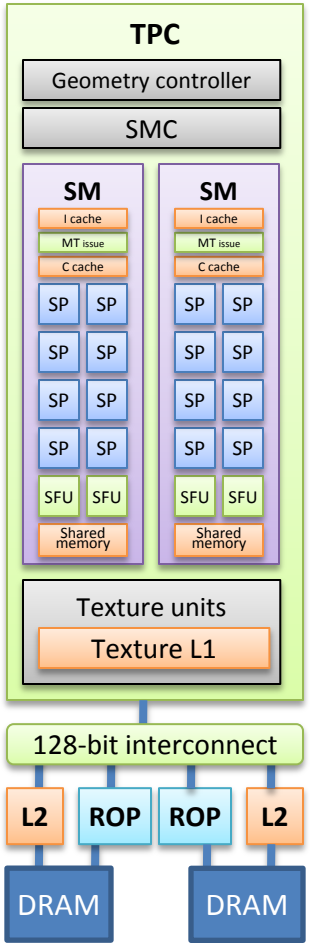


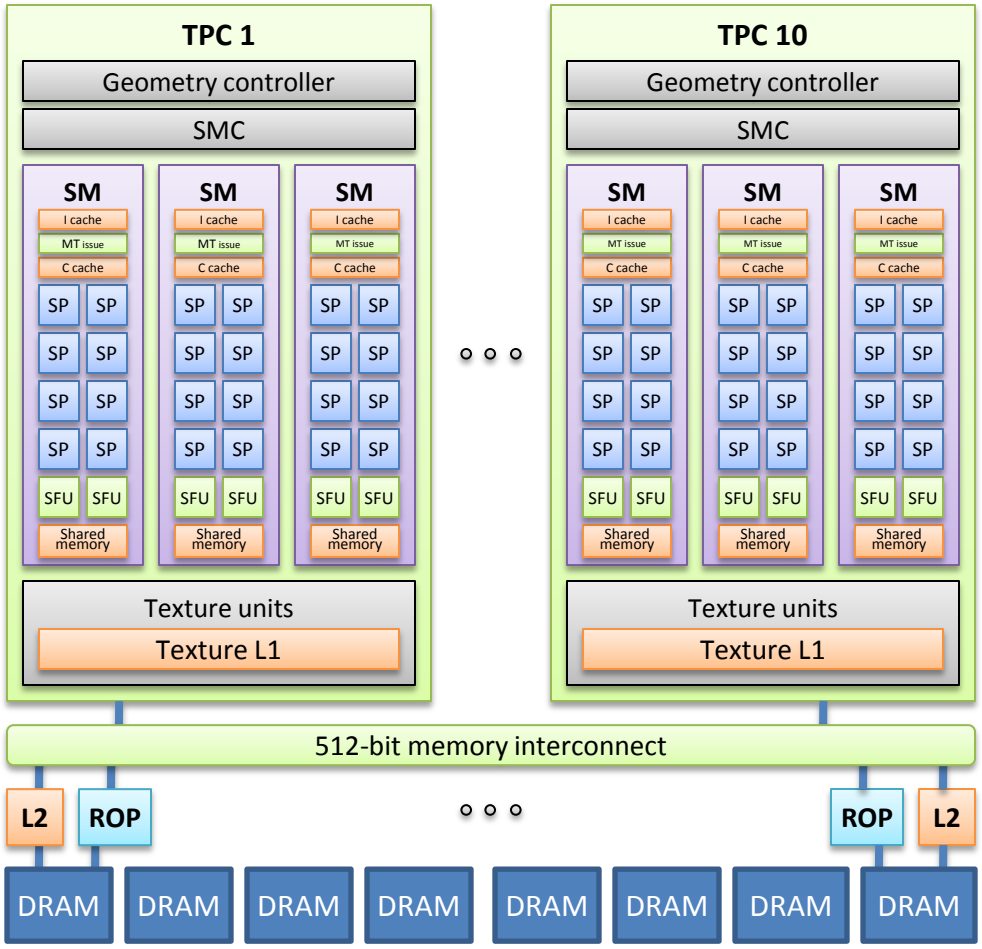
Figure is courtesy of NVIDIA

NVIDIA GeForce9400M G GPU



- 16 streaming processors arranged as 2 streaming multiprocessors
- At 0.8 GHz this provides
  - 54 GFLOPS in single-precision (SP)
- 128-bit interface to off-chip GDDR3 memory
  - 21 GB/s bandwidth

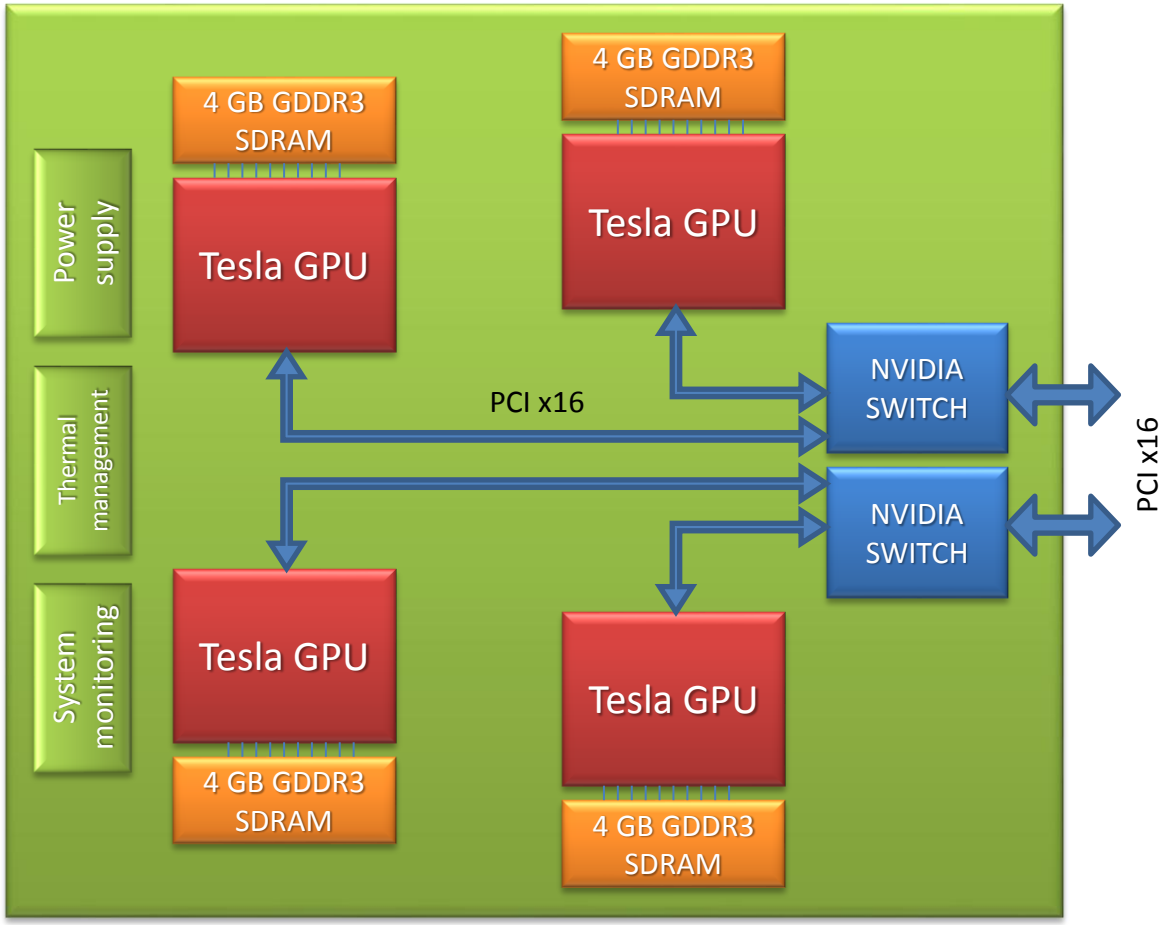
NVIDIA Tesla C1060 GPU



- 240 streaming processors arranged as 30 streaming multiprocessors
- At 1.3 GHz this provides
  - 1 TFLOPS SP
  - 86.4 GFLOPS DP
- 512-bit interface to off-chip GDDR3 memory
  - 102 GB/s bandwidth

NVIDIA Tesla S1070 Computing Server

- 4 T10 GPUs



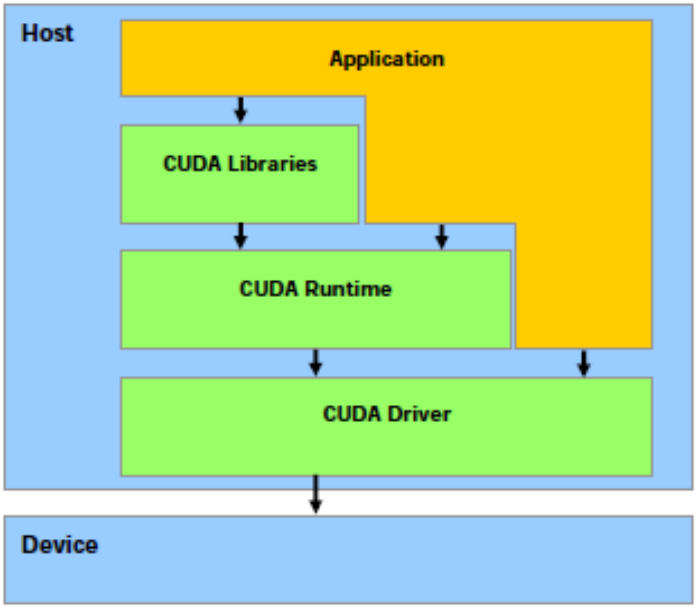
Graph is courtesy of NVIDIA

GPU Use/Programming

- GPU libraries
  - NVIDIA's CUDA BLAS and FFT libraries
  - Many 3rd party libraries
- Low abstraction lightweight GPU programming toolkits
  - CUDA C
  - OpenCL
- High abstraction compiler-based tools
  - PGI x64+GPU

CUDA C APIs

- higher-level API called the **CUDA runtime API**
  - myKernel<<<grid size>>>(args);



- low-level API called the **CUDA driver API**
  - cuModuleLoad(&module, binfile);
  - cuModuleGetFunction(&func, module, "mykernel");
  - …
  - cuParamSetv(func, 0, &args, 48);
  - …
  - cuParamSetSize(func, 48);
  - cuFuncSetBlockShape(func, ts[0], ts[1], 1);
  - cuLaunchGrid(func, gs[0], gs[1]);

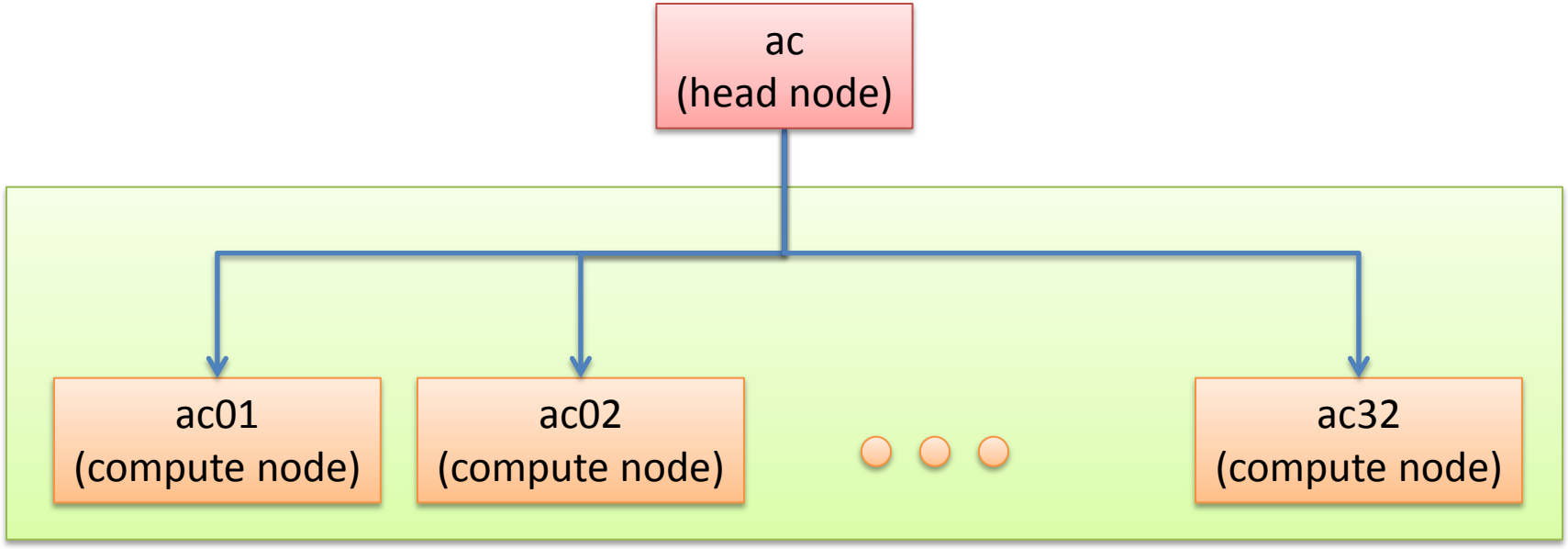
Getting Started with NCSA GPU Cluster

- Cluster architecture overview
- How to login and check out a node
- How to compile and run an existing application

NCSA **AC** GPU Cluster

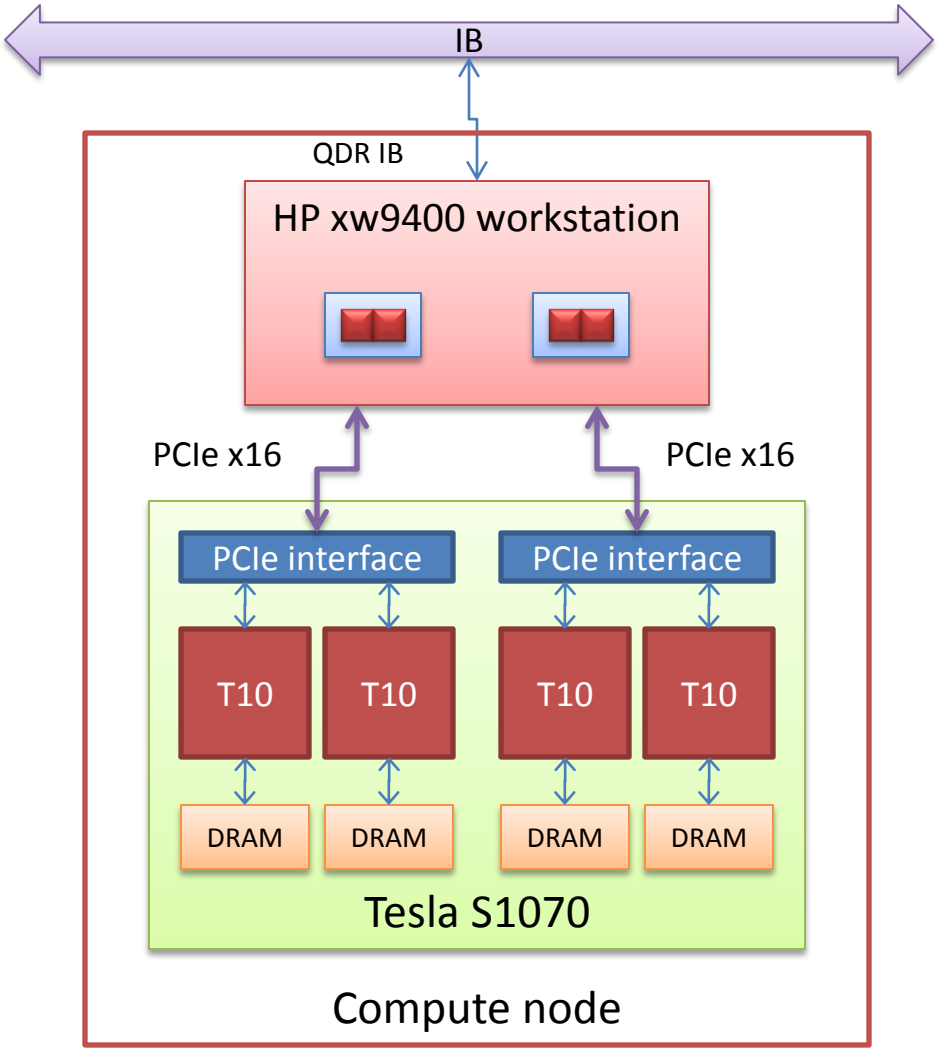
GPU Cluster Architecture

- Servers: 32
  - CPU cores: 128
- Accelerator Units: 32
  - GPUs: 128



GPU Cluster Node Architecture

- HP xw9400 workstation
  - 2216 AMD Opteron 2.4 GHz dual socket dual core
  - 8 GB DDR2
  - InfiniBand QDR
- S1070 1U GPU Computing Server
  - 1.3 GHz Tesla T10 processors
  - 4x4 GB GDDR3 SDRAM



Accessing the GPU Cluster

- Use Secure Shell (SSH) client to access AC
  - **ssh USER@ac.ncsa.uiuc.edu** (User: tra1 – tra54; Password: ???)
  - You will see something like this printed out:

```
Mar 30, 2010
ac and all compute nodes updated to the release version of CUDA 3.0 from the
beta version of 3.0, which has been in place for months.

May 11, 2010
GeForce 480 Fermi based GPU now available on ac login node, in addition to
a GeForce 280 (more similar to Tesla GPUs)

Questions?  Contact Jeremy Enos jenos@ncsa.uiuc.edu
(for password resets, please contact help@ncsa.uiuc.edu)

 13:52:30 up 10 days, 22:23, 32 users,  load average: 0.54, 0.29, 0.20
Job state_count = Transit:0 Queued:101 Held:0 Waiting:0 Running:111 Exiting:0

[tra1@ac ~]$ _
```

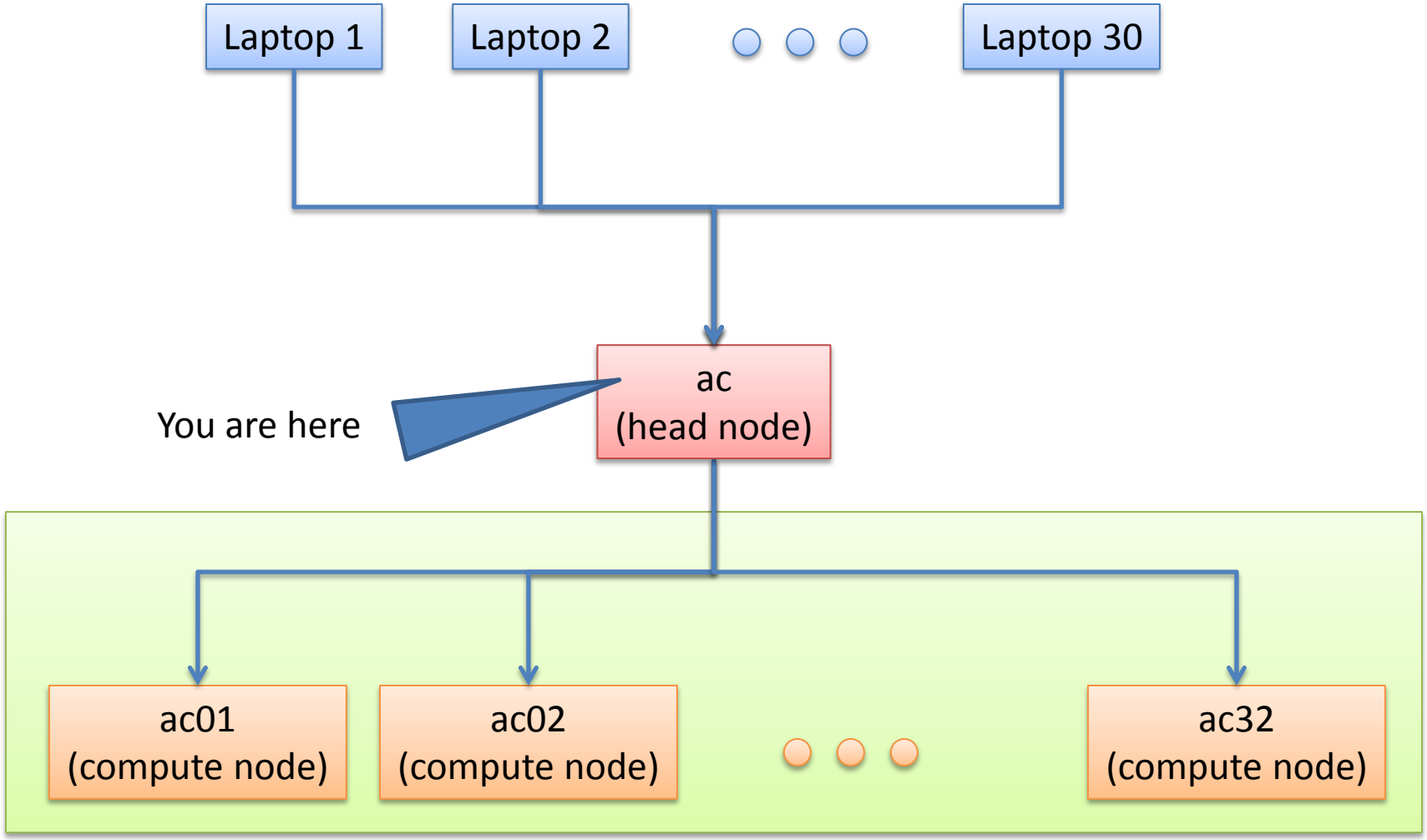
Installing Tutorial Examples

- Run this sequence to retrieve and install tutorial examples:

```
cd
cp /tmp/tutorial.tgz  .
tar -xvzf tutorial.tgz
cd tutorial
ls
```

```
src1  src2  src3
```

Accessing the GPU Cluster



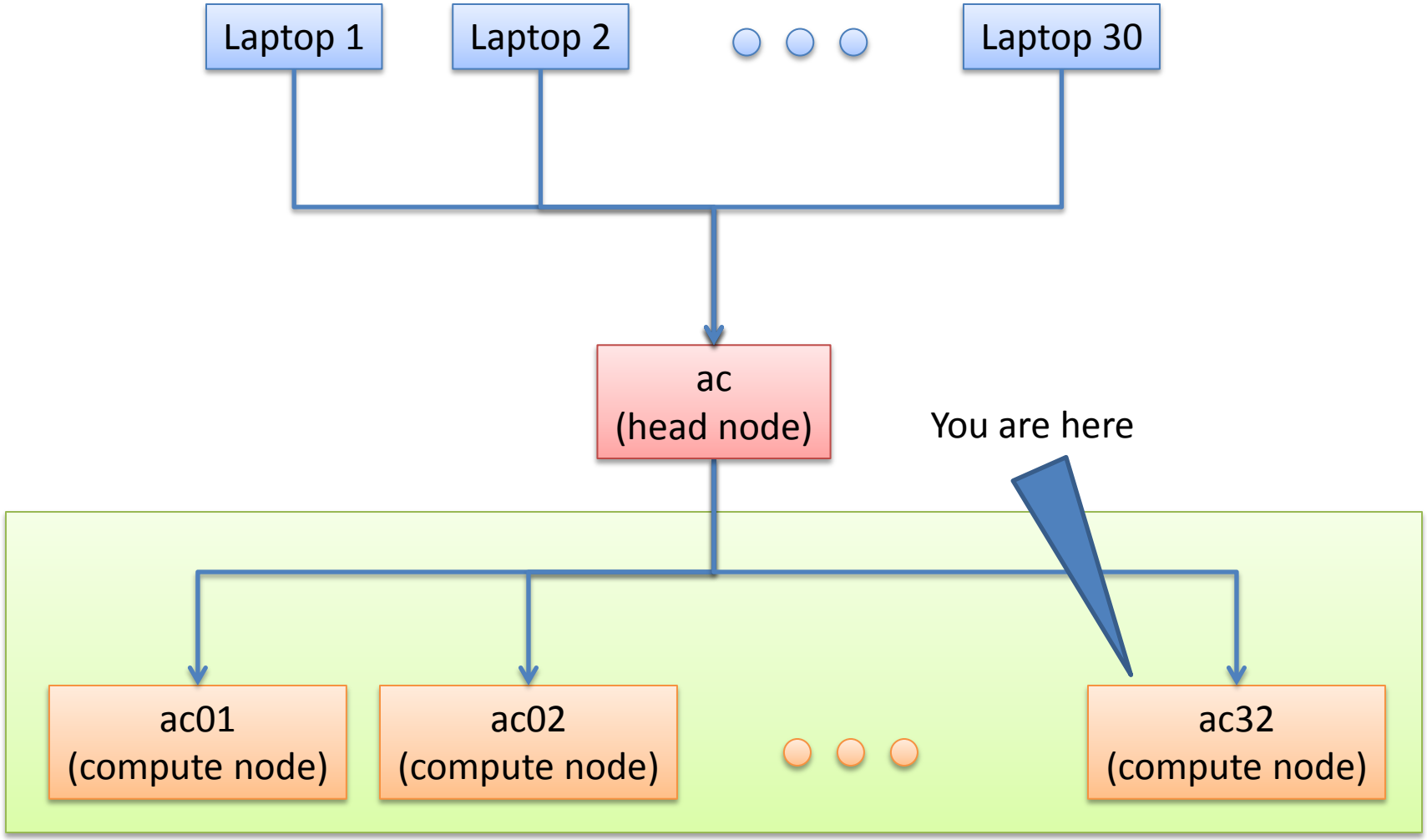
Requesting a Cluster Node for Interactive Use

- Run **qstat** to see what other users do

- Run **qsub -I -l walltime=02:00:00** to request a node with a single GPU for 2 hours of interactive use
  - You will see something like this printed out:

```
qsub: waiting for job 64424.acm to start
qsub: job 64424.acm ready
[gpuXYZ@ac32 ~]$ _
```

Requesting a Cluster Node



Some useful utilities installed on AC

- As part of NVIDIA driver
  - nvidia-smi (NVIDIA System Management Interface program)
- As part of NVIDIA CUDA SDK
  - deviceQuery
  - bandwidthTest
- As part of CUDA wrapper
  - wrapper_query
  - showgputime/showallgputime (works from the head node only)

nvidia-smi

```
Timestamp                       : Mon May 24 14:39:28 2010
Unit 0:
    Product Name          : NVIDIA Tesla S1070-400 Turn-key
    Product ID            : 920-20804-0006
    Serial Number         : 0324708000059
    Firmware Ver          : 3.6
    Intake Temperature    : 22 C
    GPU 0:
        Product Name      : Tesla T10 Processor
        Serial            : 2624258902399
        PCI ID            : 5e710de
        Bridge Port       : 0
        Temperature       : 33 C
    GPU 1:
        Product Name      : Tesla T10 Processor
        Serial            : 2624258902399
        PCI ID            : 5e710de
        Bridge Port       : 2
        Temperature       : 30 C
    Fan Tachs:
        #00: 3566 Status: NORMAL
        #01: 3574 Status: NORMAL
...
        #12: 3564 Status: NORMAL
        #13: 3408 Status: NORMAL
    PSU:
        Voltage           :  12.01 V
        Current           :  19.14 A
        State             : Normal
    LED:
        State             : GREEN
```

```
Unit 1:
    Product Name          : NVIDIA Tesla S1070-400 Turn-key
    Product ID            : 920-20804-0006
    Serial Number         : 0324708000059
    Firmware Ver          : 3.6
    Intake Temperature    : 22 C
    GPU 0:
        Product Name      : Tesla T10 Processor
        Serial            : 1930554578325
        PCI ID            : 5e710de
        Bridge Port       : 0
        Temperature       : 33 C
    GPU 1:
        Product Name      : Tesla T10 Processor
        Serial            : 1930554578325
        PCI ID            : 5e710de
        Bridge Port       : 2
        Temperature       : 30 C
    Fan Tachs:
        #00: 3584 Status: NORMAL
        #01: 3570 Status: NORMAL
        ...
        #12: 3572 Status: NORMAL
        #13: 3412 Status: NORMAL
    PSU:
        Voltage           :  11.99 V
        Current           :  19.14 A
        State             : Normal
    LED:
        State             : GREEN
```

deviceQuery

```
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA
Device 0: "Tesla T10 Processor"
  CUDA Driver Version:                           3.0
  CUDA Runtime Version:                          3.0
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 4294770688 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Exclusive (only one host thread at a time can use this device)
```

wrapper_query

```
cuda_wrapper info:
        version=2
        userID=21783
        pid=-1
        nGPU=1
        physGPU[0]=2
        key_env_var=
        allow_user_passthru=1
        affinity:
                GPU=0,    CPU=0 2
                GPU=1,    CPU=0 2
                GPU=2,    CPU=1 3
                GPU=3,    CPU=1 3
        cudaAPI = Unknown
        walltime = 10.228021 seconds
        gpu_kernel_time = 0.000000 seconds
        gpu_usage = 0.00%
```

- There are 4 GPUs per cluster node
- When requesting a node, we can specify how may GPUs should be allocated
  - e.g., `-l nodes=1:ppn=4` in **qsub** resources string will result in all 4 GPUs allocated
- By default, only one GPU per node is allocated

Compiling and Running an Existing Application

- cd tutorial/src1
  - vecadd.c - reference C implementation
  - vecadd.cu – CUDA implementation

- Compile & run CPU version

  gcc vecadd.c -o vecadd_cpu

  ./vecadd_cpu

  Running CPU vecAdd for 16384 elements

  C[0]=2147483648.00 ...

- Compile & run GPU version

  nvcc vecadd.c**u** -o vecadd_**g**pu

  ./vecadd_**g**pu

  *Running **G**PU vecAdd for 16384 elements*

  C[0]=2147483648.00 ...

nvcc

- Any source file containing CUDA C language extensions must be compiled with nvcc
- nvcc is a compiler driver that invokes many other tools to accomplish the job
- Basic nvcc usage
  - nvcc <filename>.cu [-o <executable>]
    - Builds release mode
  - nvcc -deviceemu <filename>.cu
    - Builds device emulation mode (all code runs on CPU)
  - -g flag allows to build debug mode for gdb debugger
  - nvcc --version

Anatomy of a GPU Application

- Host side
- Device side

Reference CPU Version

```
void vecAdd(int N, float* A, float* B, float* C) {
    for (int i = 0; i < N; i++)  C[i] = A[i] + B[i];
}
```

Computational kernel

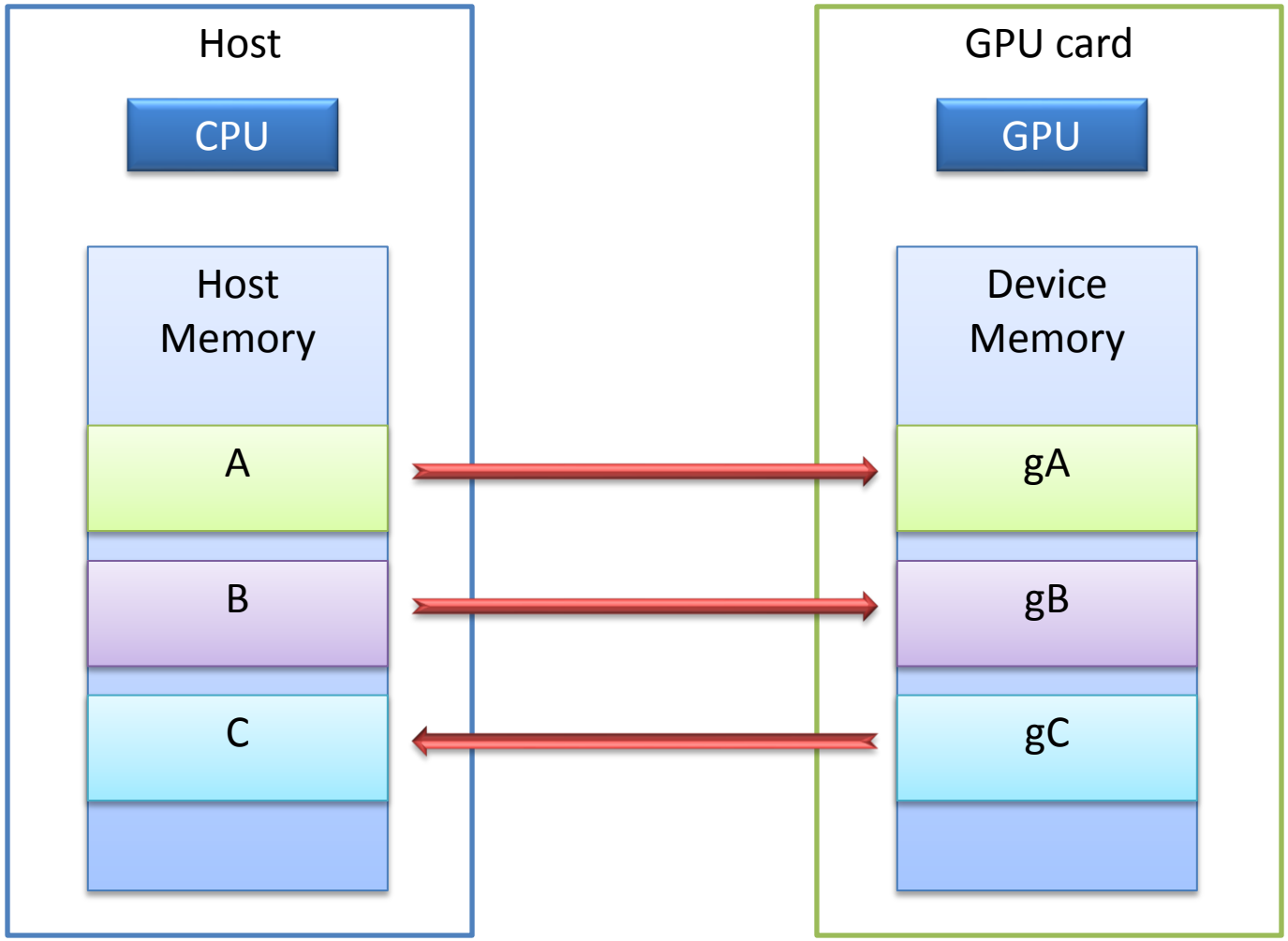
```
int  main(int argc, char **argv)
{
    int N = 16384;  // default vector size

    float *A = (float*)malloc(N * sizeof(float));
    float *B = (float*)malloc(N * sizeof(float));
    float *C = (float*)malloc(N * sizeof(float));

    vecAdd(N, A, B, C);  // call compute kernel

    free(A); free(B); free(C);
}
```

Memory allocation

Kernel invocation

Memory de-allocation

Adding GPU support



Memory Spaces

- **CPU and GPU have separate memory spaces**
  - **Data is moved across PCIe bus**
  - **Use functions to allocate/set/copy memory on GPU**
- **Host (CPU) manages device (GPU) memory**
  - **cudaMalloc(void** pointer, size_t nbytes)**
  - **cudaFree(void* pointer)**
  - **cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction);**
    - **returns after the copy is complete**
    - **blocks CPU thread until all bytes have been copied**
    - **does not start copying until previous CUDA calls complete**
  - **enum cudaMemcpyKind**
    - **cudaMemcpyHostToDevice**
    - **cudaMemcpyDeviceToHost**
    - **cudaMemcpyDeviceToDevice**

Adding GPU support

```
int  main(int argc, char **argv)
{
   int  N = 16384;  // default vector size

   float *A = (float*)malloc(N * sizeof(float));
   float *B = (float*)malloc(N * sizeof(float));
   float *C = (float*)malloc(N * sizeof(float));

   float *devPtrA, *devPtrB, *devPtrC;

   cudaMalloc((void**)&devPtrA, N * sizeof(float));
   cudaMalloc((void**)&devPtrB, N * sizeof(float));
   cudaMalloc((void**)&devPtrC, N * sizeof(float));

   cudaMemcpy(devPtrA, A, N * sizeof(float),  cudaMemcpyHostToDevice);
   cudaMemcpy(devPtrB, B, N * sizeof(float),  cudaMemcpyHostToDevice);
```

Memory allocation on the GPU card

Copy data from the CPU (host) memory to the GPU (device) memory

Adding GPU support

```
vecAdd<<<N/512, 512>>>(devPtrA, devPtrB, devPtrC);
```

Kernel invocation

```
cudaMemcpy(C, devPtrC, N * sizeof(float),  cudaMemcpyDeviceToHost);
```

Copy results from device memory to the host memory

```
cudaFree(devPtrA);
cudaFree(devPtrB);
cudaFree(devPtrC);
```

Device memory de-allocation

```
  free(A);
  free(B);
  free(C);
}
```

GPU Kernel

- **CPU version**

```
void vecAdd(int N, float* A, float* B, float* C)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

- **GPU version**

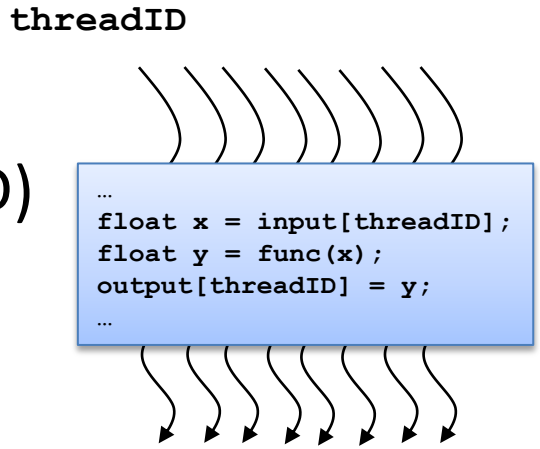
```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Introduction to GPU programming

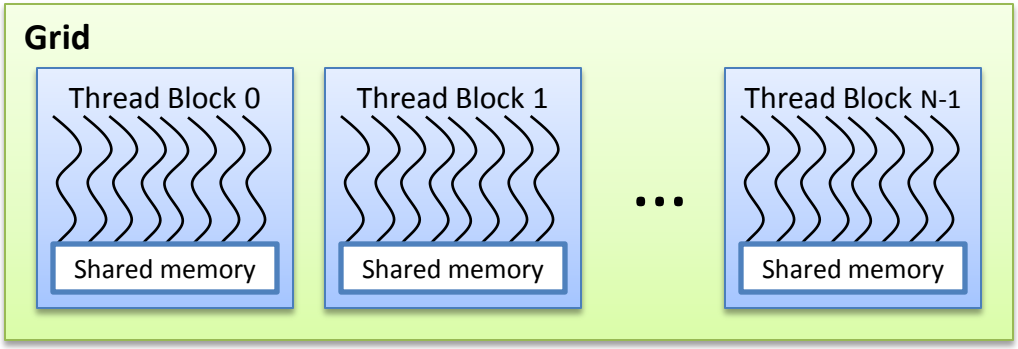
- CUDA programming model
- GPU memory hierarchy
- CUDA C

CUDA Programming Model

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



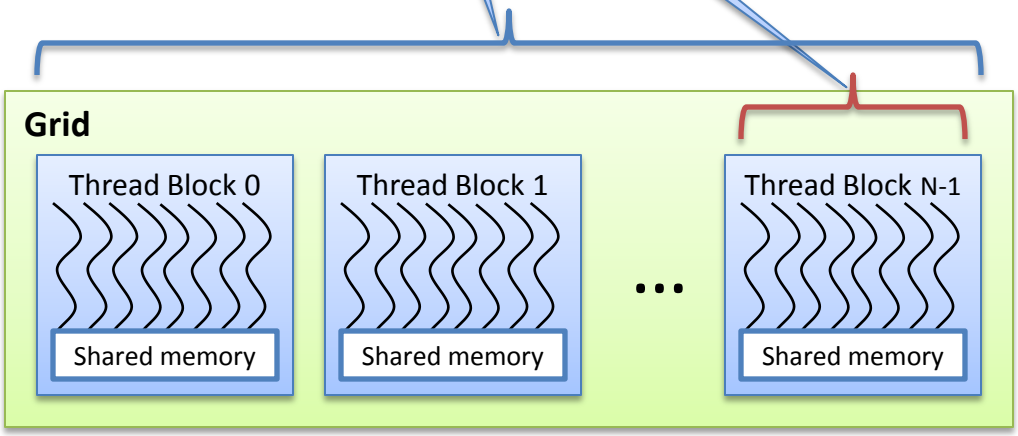
- Threads are arranged as a grid of thread blocks
  - Threads within a block have access to a segment of shared memory



Kernel Invocation Syntax

grid & thread block dimensionality

```
vecAdd<<<32, 512>>>(devPtrA, devPtrB, devPtrC);
```




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

block ID within a grid

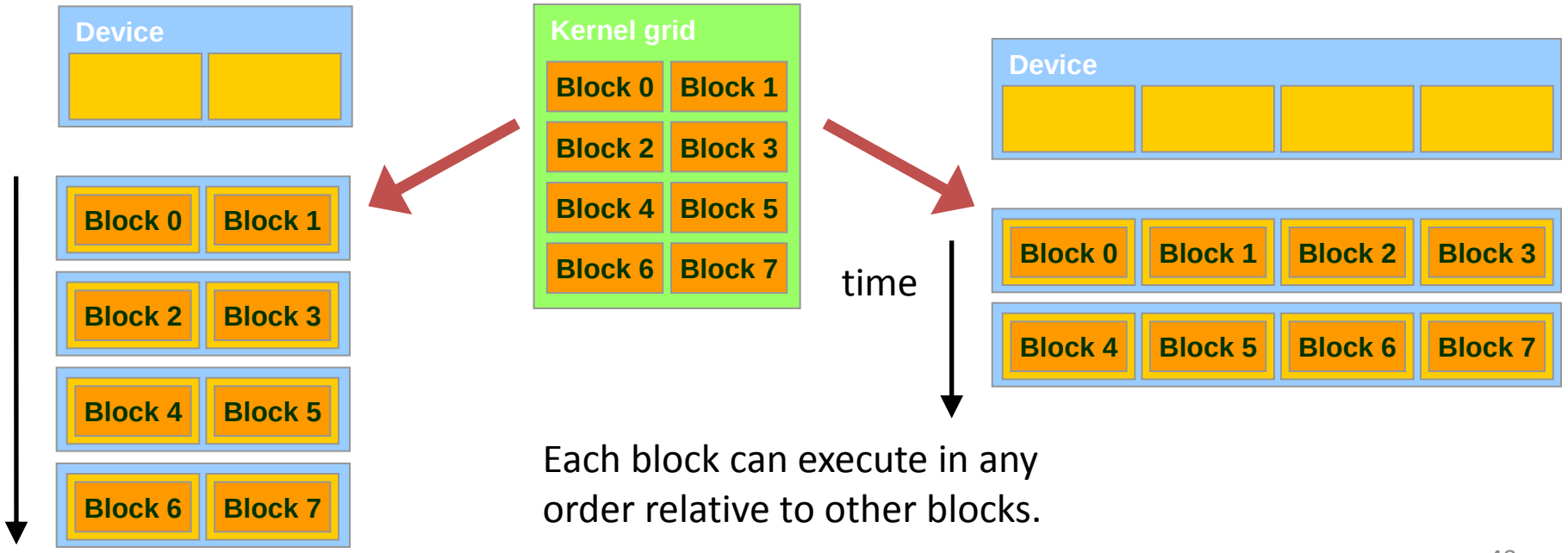
number of threads per block

thread ID within a thread block

Mapping Threads to the Hardware

- Blocks of threads are transparently assigned to SMs
  - A block of threads executes on one SM & does not migrate
  - Several blocks can reside concurrently on one SM

- Blocks must be independent
  - Any possible interleaving of blocks should be valid
  - Blocks may coordinate but not synchronize
  - Thread blocks can run in any order

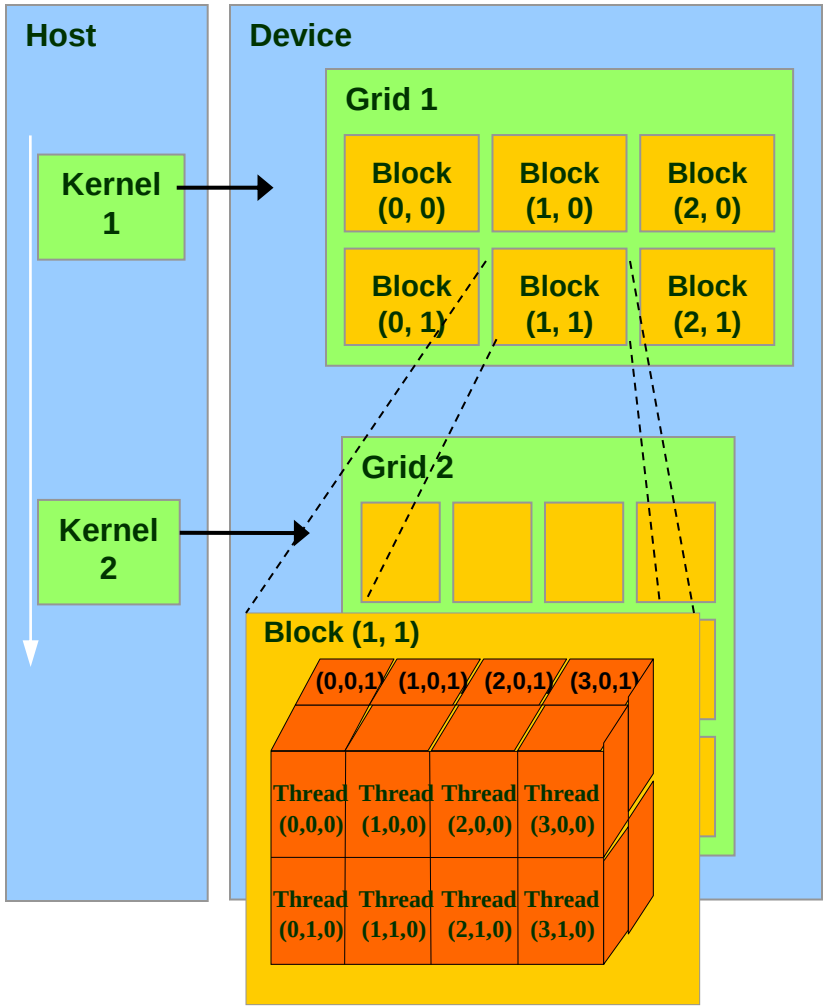


Each block can execute in any order relative to other blocks.

Slide is courtesy of NVIDIA

CUDA Programming Model

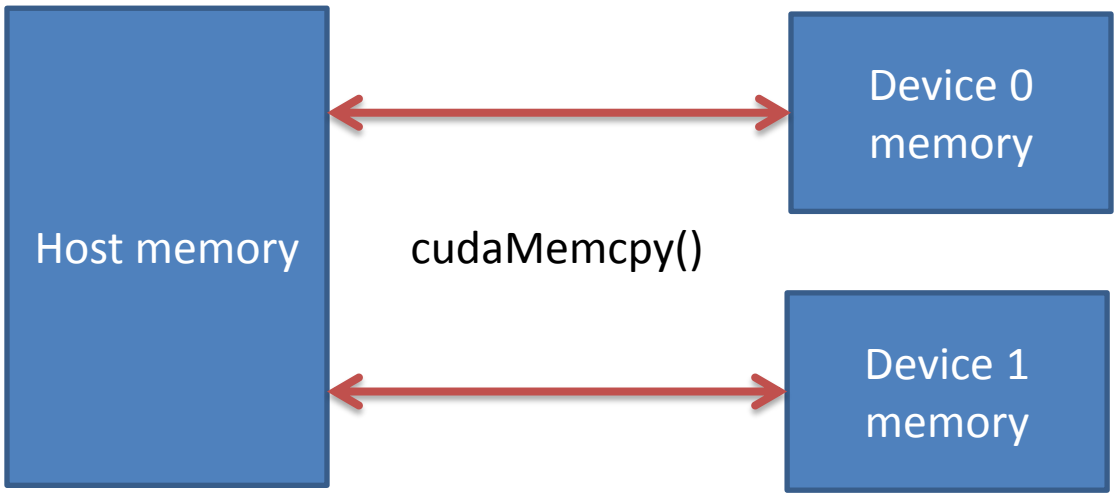
- A kernel is executed as a grid of thread blocks
  - Grid of blocks can be 1 or 2-dimentional
  - Thread blocks can be 1, 2, or 3-dimensional
- Different kernels can have different grid/block configuration
- Threads from the same block have access to a shared memory and their execution can be synchronized



Slide is courtesy of NVIDIA

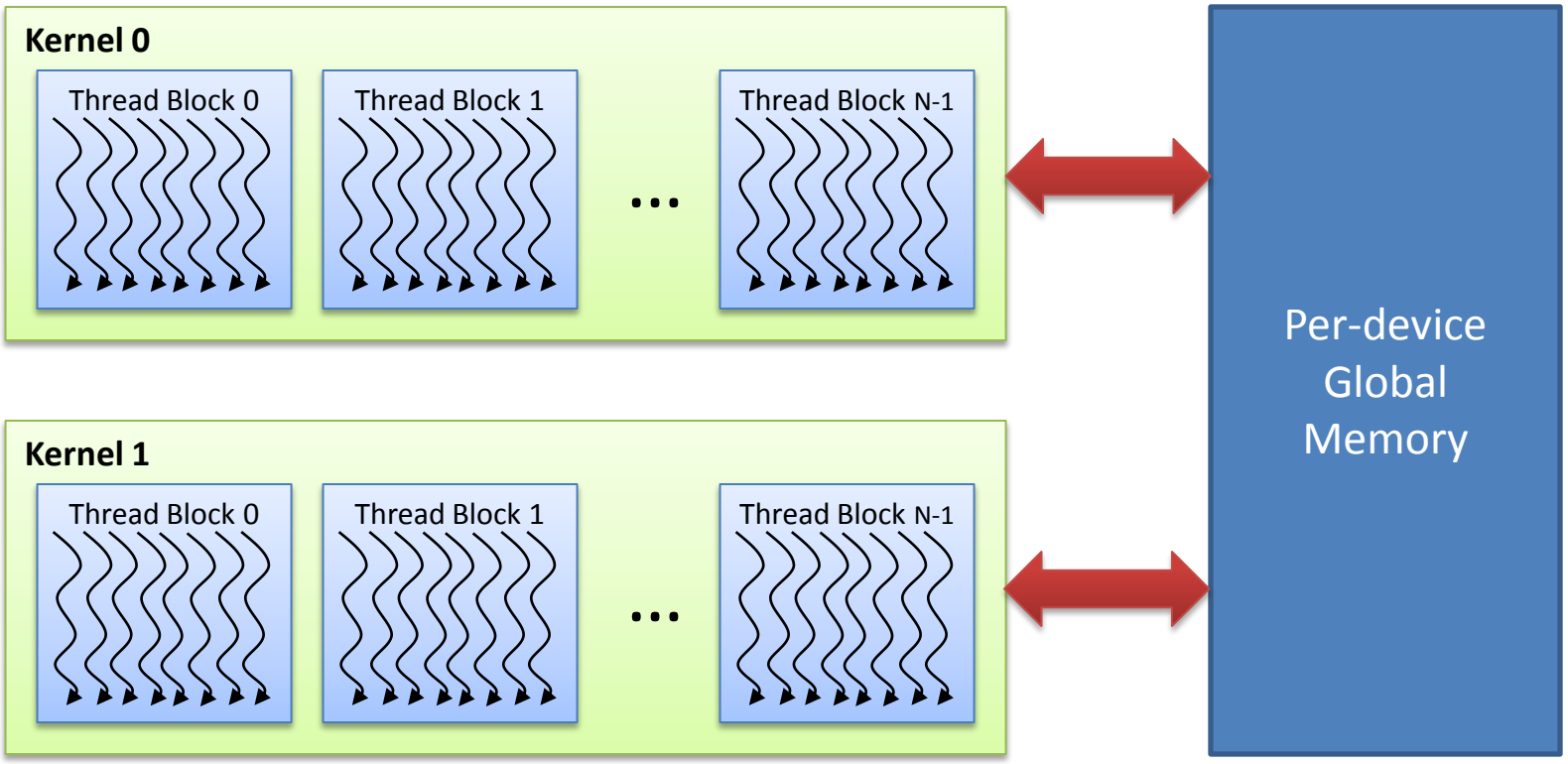
GPU Memory Hierarchy

- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime is from allocation to deallocation



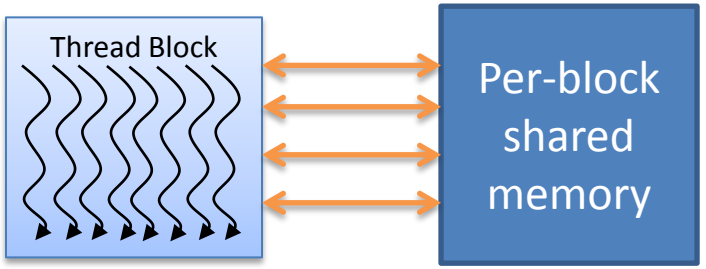
GPU Memory Hierarchy

- Global (device) memory



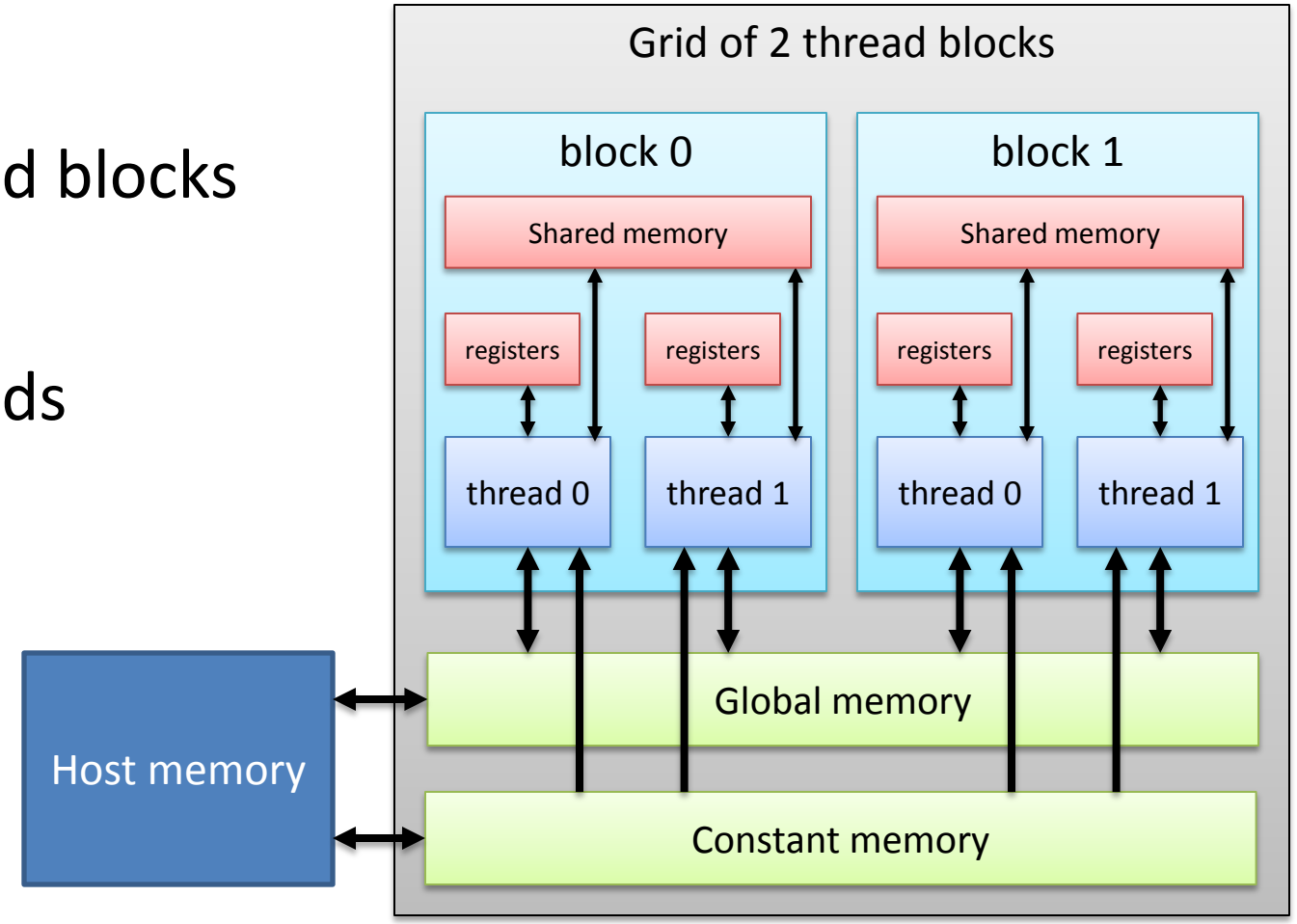
GPU Memory Hierarchy

- Local storage
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
  - Data lifetime = thread lifetime
- Shared memory
  - Each thread block has own shared memory
    - Accessible only by threads within that block
  - Data lifetime = block lifetime

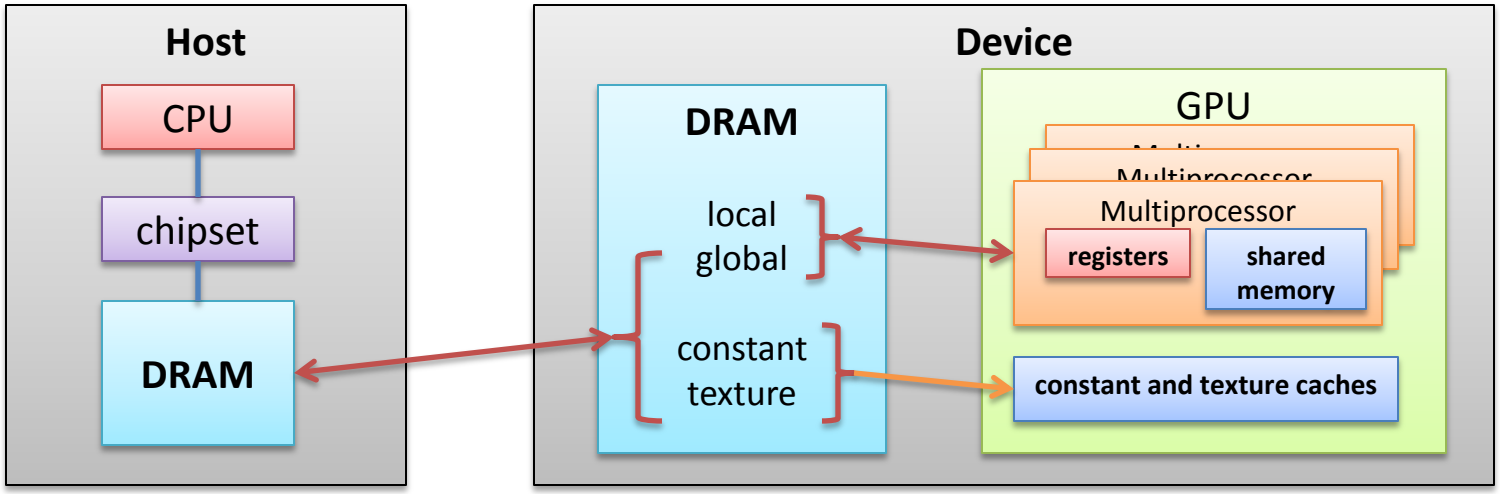


GPU Memory Hierarchy

- 1D grid
  - 2 thread blocks
- 1D block
  - 2 threads



GPU Memory Hierarchy



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application