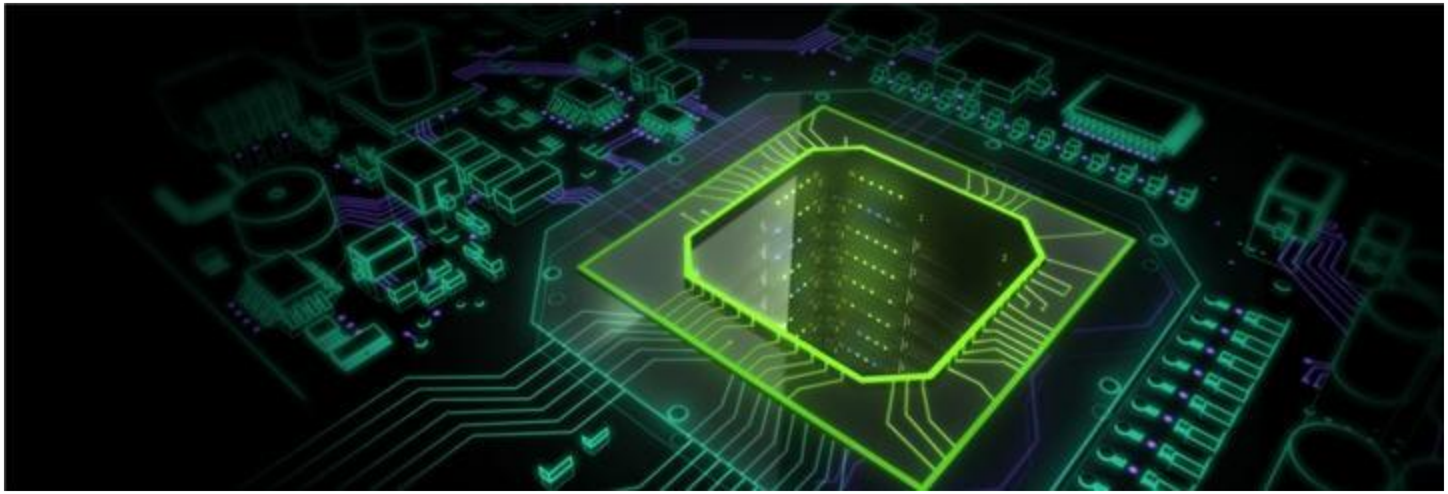
CSCI-GA.3033-012

Graphics Processing Units (GPUs): Architecture and Programming

Lecture 9: Multi-GPU Systems

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



Tianhe-1A: #2 in latest Top500

Intel Xeon X5670 and NVIDIA 2050 GPU

Nebulae: #4 in Top 500 list

Intel Xeon X5650 and Nvidia GPU Tesla c2050

Tsubame 2.0: #5 in Top 500 list

Intel Xeon X5670 and Nvidia GPU

From the Nov 2011 Top500 supercomputers

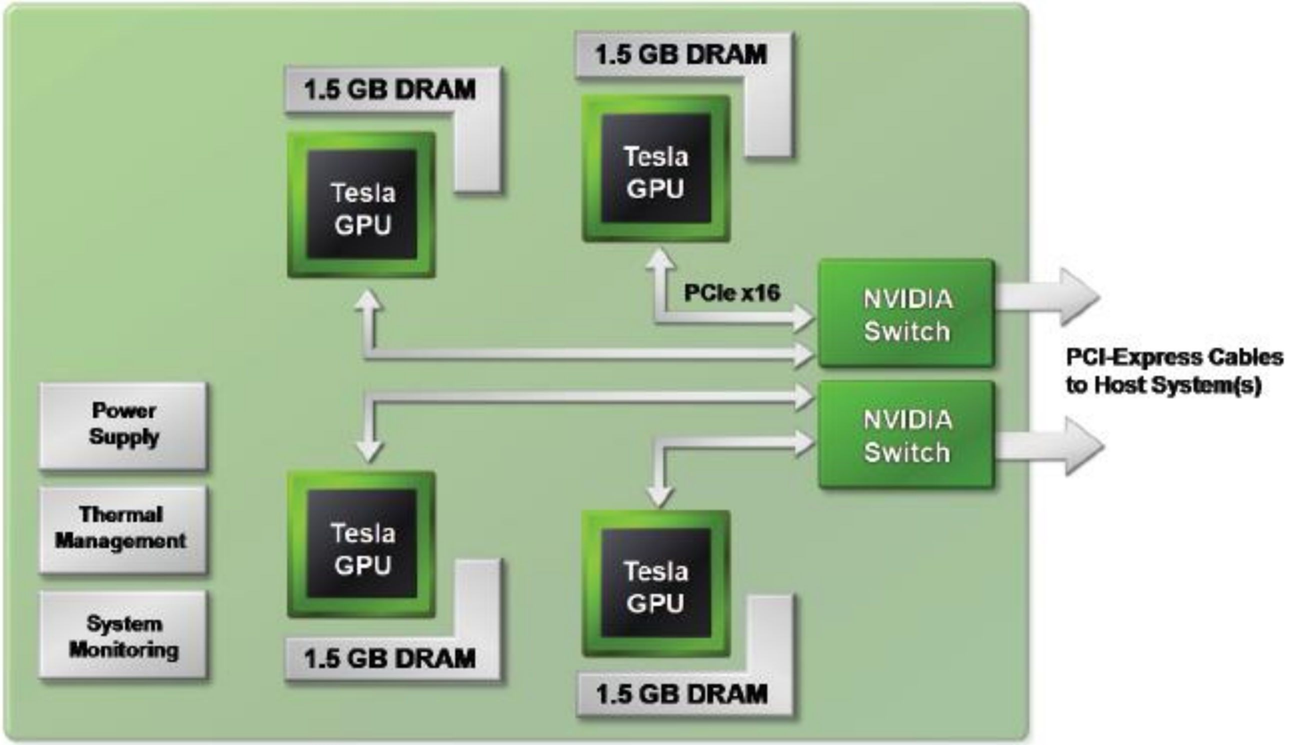
- From the top 5: 3 are using GPUs
- A total of 39 systems on the list are using GPU technology (up from 17 in the previous list)
- 35 of these use NVIDIA chips, two use Cell processors, and two use ATI Radeon

Flavors

- Multiple GPUs in the same node (e.g. PC)
- Multi-node system (e.g. MPI).

Multi-GPU configuration is here to stay!

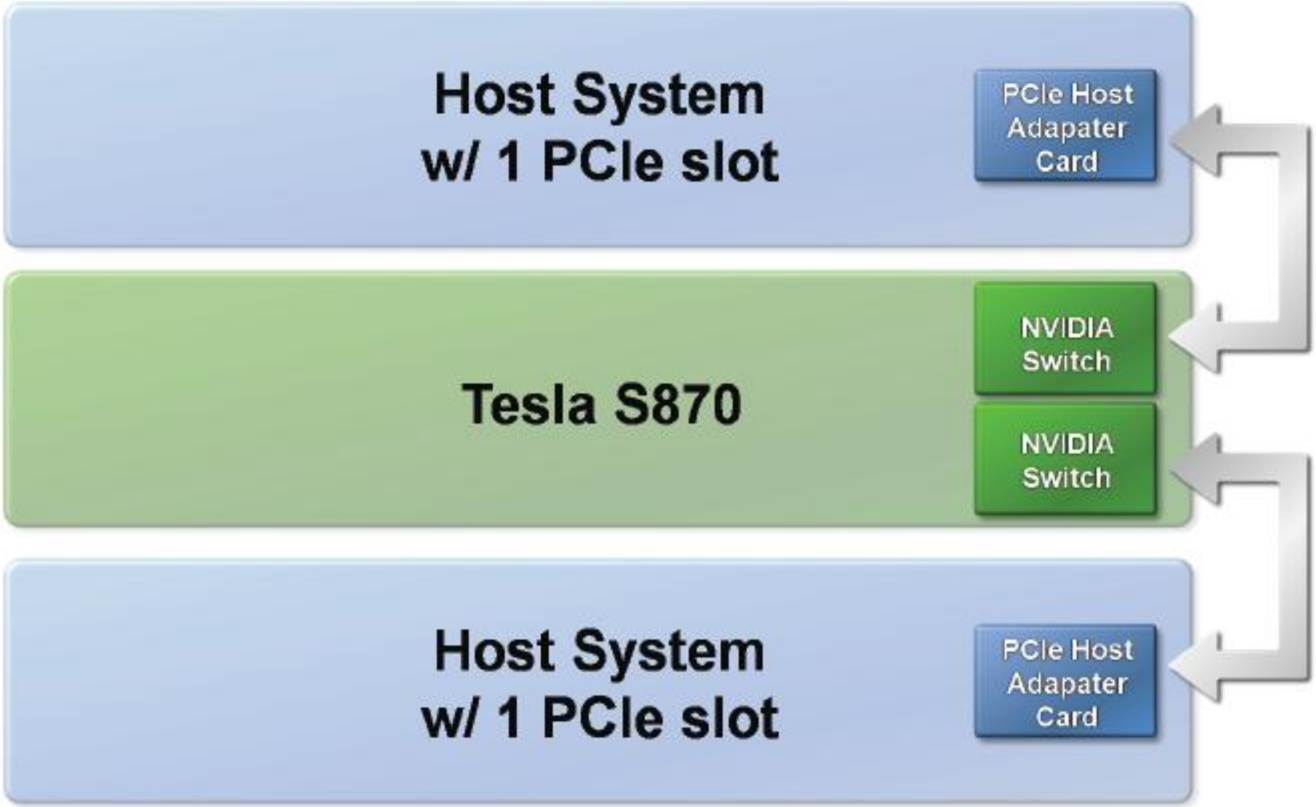
Hardware Example: Tesla S870 Server



Hardware Example: Tesla S870 Server

Connected to a single-host

Hardware Example: Tesla S870 Server



Connected to a two host systems

Why Multi-GPU Solutions

- Scaling-up performance
- Another level of parallelism
- Power
- Reliability

```
// Run independent kernel on each CUDA device
intnumDevs= 0;
cuda GetNumDevices(&numDevs);
…
for (intd = 0; d < numDevs; d++) {
        cudaSetDevice(d);
        kernel<<<blocks, threads>>>(args);
}
```

CUDA Support

- cudaGetDeviceCount(int * count)
  - Returns in *count the number of devices
- cudaGetDevice (int * *device*)
  - Returns in *device the device on which the active host thread executes the device code.

CUDA Support

- cudaSetDevice(devID)
  - Devise selection within the code by specifying the identifier and making CUDA kernels run on the selected GPU.

```
size_t size = 1024 * sizeof(float);
cudaSetDevice(0);            // Set device 0 as current
float* p0;
cudaMalloc(&p0, size);       // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1);            // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);       // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```

CUDA Support: Peer to peer memory Access

- Peer-to-Peer Memory Access
  - Only on Tesla or above
  - cudaDeviceEnablePeerAccess() to check peer access

```
cudaSetDevice(0);                      // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);                 // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);           // Launch kernel on device 0
cudaSetDevice(1);                      // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0);      // Enable peer-to-peer access
                                       // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```

peerDevice
ID

cudaError_t **cudaDeviceEnablePeerAccess** (
int peerDevice,
unsigned int flags)

Always set
to zero

Access granted by this call is unidirectional (i.e. current device can access peer device)

CUDA Support
Peer to peer memory Copy

- Using cudaMemcpyPeer()
  - works for Geforce 480 and other GPUs.

```
cudaSetDevice(0);                    // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);               // Allocate memory on device 0
cudaSetDevice(1);                    // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);               // Allocate memory on device 1
cudaSetDevice(0);                    // Set device 0 as current
MyKernel<<<1000, 128>>>(p0);         // Launch kernel on device 0
cudaSetDevice(1);                    // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1);         // Launch kernel on device 1
```

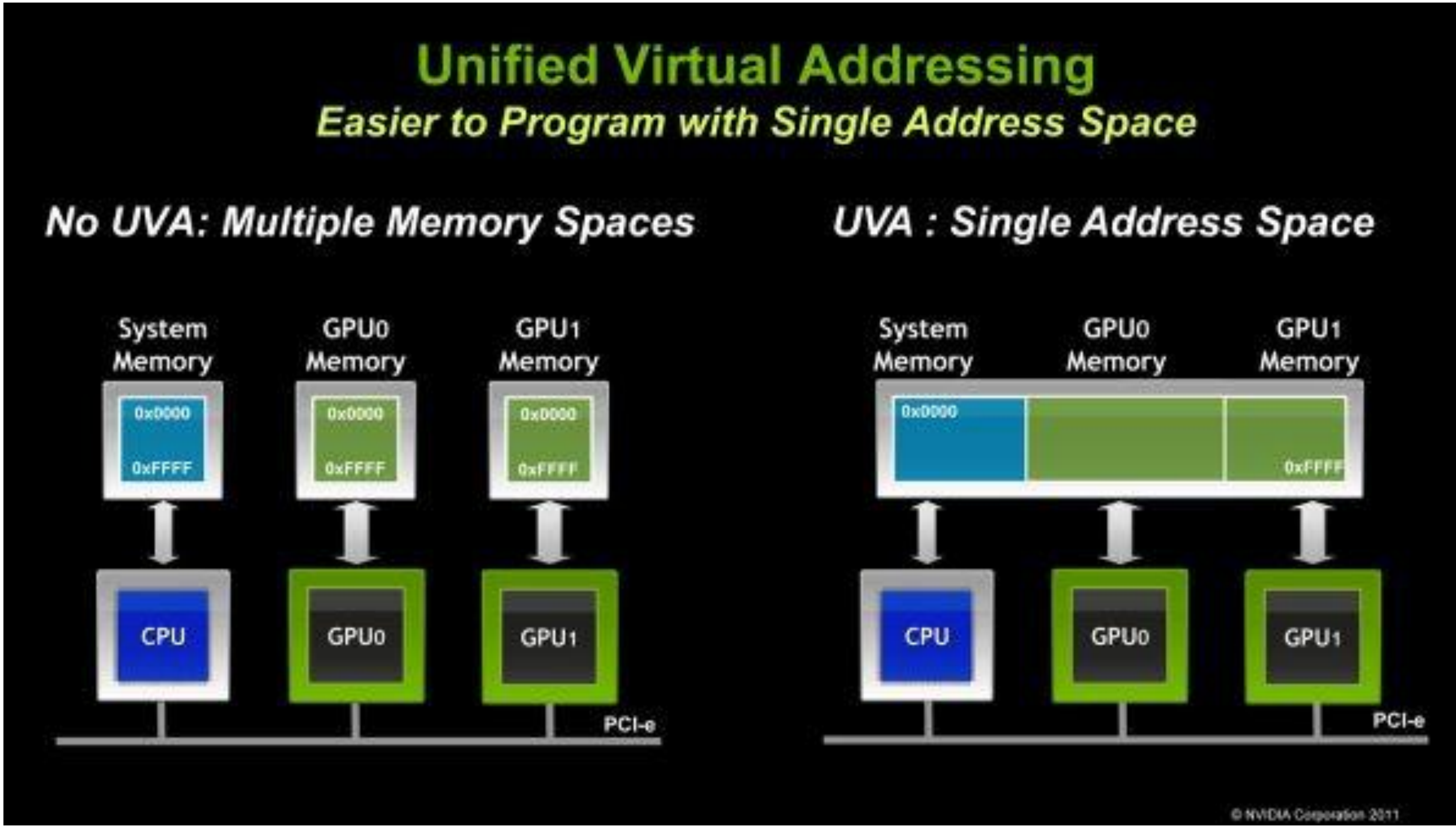
cudaMemcpyPeer (void * *dst*,
int *dstDevice*,
const void * *src*,
int *srcDevice*,
size_t *count*)

Size of memory copy in bytes

- This function is asynchronous with respect to the host.
- This function is serialized with respect all pending and future asynchronous work in to the current device.

Unified Virtual Address Space (UVA)

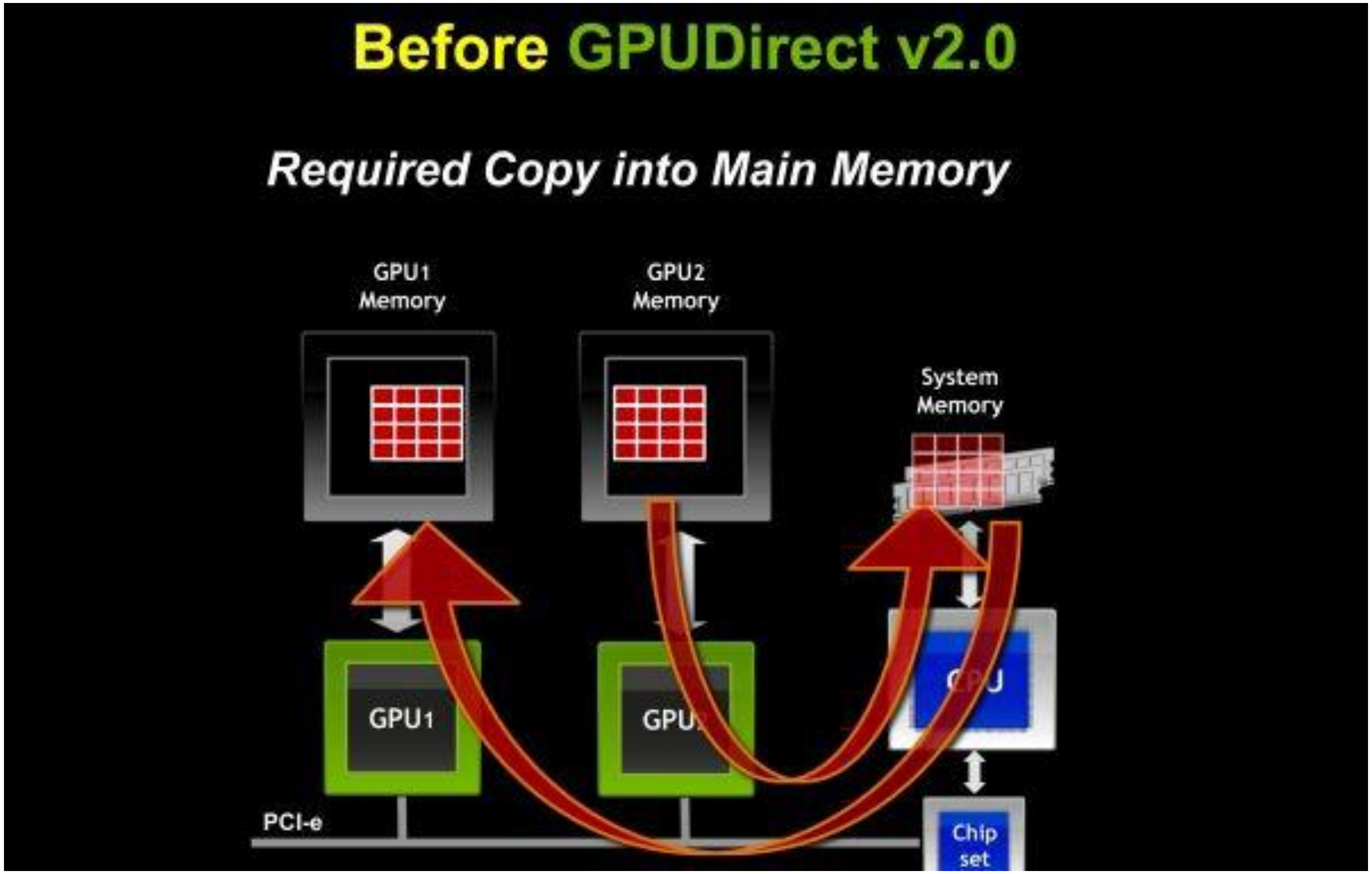
- From CUDA 4.0
  - CUDA < 4.0 mandates one host thread per CUDA device
- puts all CUDA execution, CPU and GPU, in the same address space
- Requires Fermi-class GPU
- Requires 64-bit application
- Call cudaGetDeviceProperties() for all participating devices and check cudaDeviceProp::unifiedAddressing flag



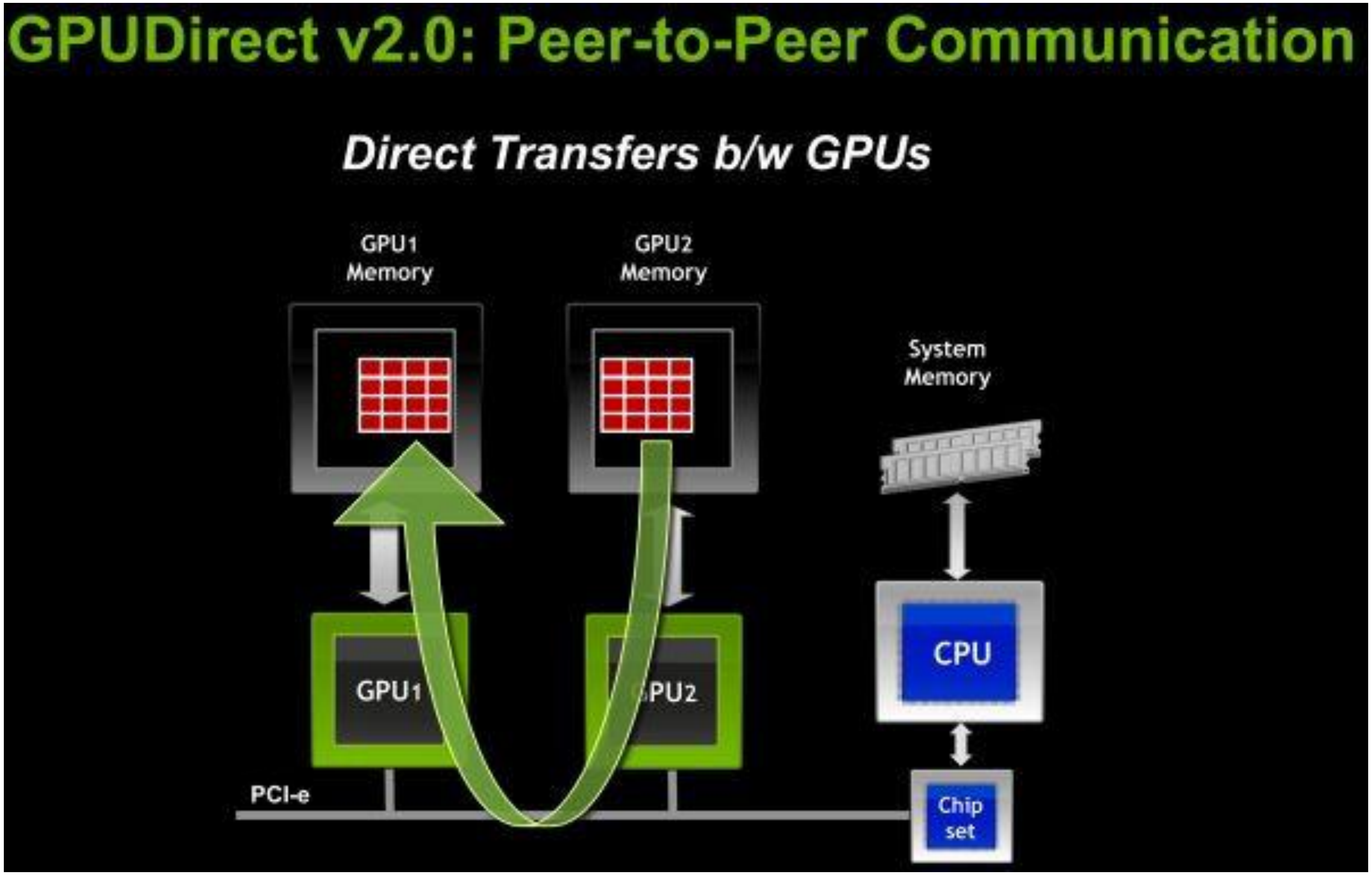
Unified Virtual Addressing
Easier to Program with Single Address Space
No UVA: Multiple Memory Spaces
UVA : Single Address Space
System Memory
GPU0 Memory
GPU1 Memory
0x0000
0xFFFF
CPU
GPU0
GPU1
PCI-e
© NVIDIA Corporation 2011

GPUDirect

- Build on UVA for Tesla (fermi) products.



GPUDirect



GPUDirect v2.0: Peer-to-Peer Communication
Direct Transfers b/w GPUs
GPU1
Memory
GPU2
Memory
System
Memory
CPU
GPU1
PU2
PCI-e
Chip
set

Easier Memory Copy

- **Between host and multiple devices:**

cudaMemcpy(gpu0_buf, host_buf, buf_size, cudaMemcpyDefault)
cudaMemcpy(gpu1_buf, host_buf, buf_size, cudaMemcpyDefault)
cudaMemcpy(host_buf, gpu0_buf, buf_size, cudaMemcpyDefault)
cudaMemcpy(host_buf, gpu1_buf, buf_size, cudaMemcpyDefault)

- **Between two devices:**

cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault)

- **cudaMemcpy() knows that our buffers are on different devices**
- **(UVA), will do a P2P copy**
- **Note that this will transparently fall back to a normal copy through the host if P2P is not available**

Example: Direct N-Body

- Simulation of dynamical system of N-bodies
- $O(N^2)$
- Compute-Bound application
- Assume we have K GPUs
  - So each GPU is responsible for N/K bodies
- For each iteration:
  - Get all N up-to-date positions onto each GPU
  - Compute accelerations –N/k per GPU
  - Integrate position, velocity –N/k per GPU

Example: Direct N-Body

- Sharing data among GPUs: options
  - Explicit copies via host
  - Zero-copy shared host array (cudaMallocHost())
  - Per-device arrays with peer-to-peer exchange transfers
  - Peer-to-peer memory access

Example: Direct N-Body

- Sharing data among GPUs: options
  - Explicit copies via host
  - Zero-copy shared host array (cudaMallocHost()): use it when:
    - You copy data to the device and access it there only once AND/OR
    - You generate data on the device and copy back to host without reuse AND/OR
    - Your kernel(s) that access the memory are compute bound (see n-body)
  - Per-device peer-to-peer exchange transfers (UVA)
  - Peer-to-peer memory access

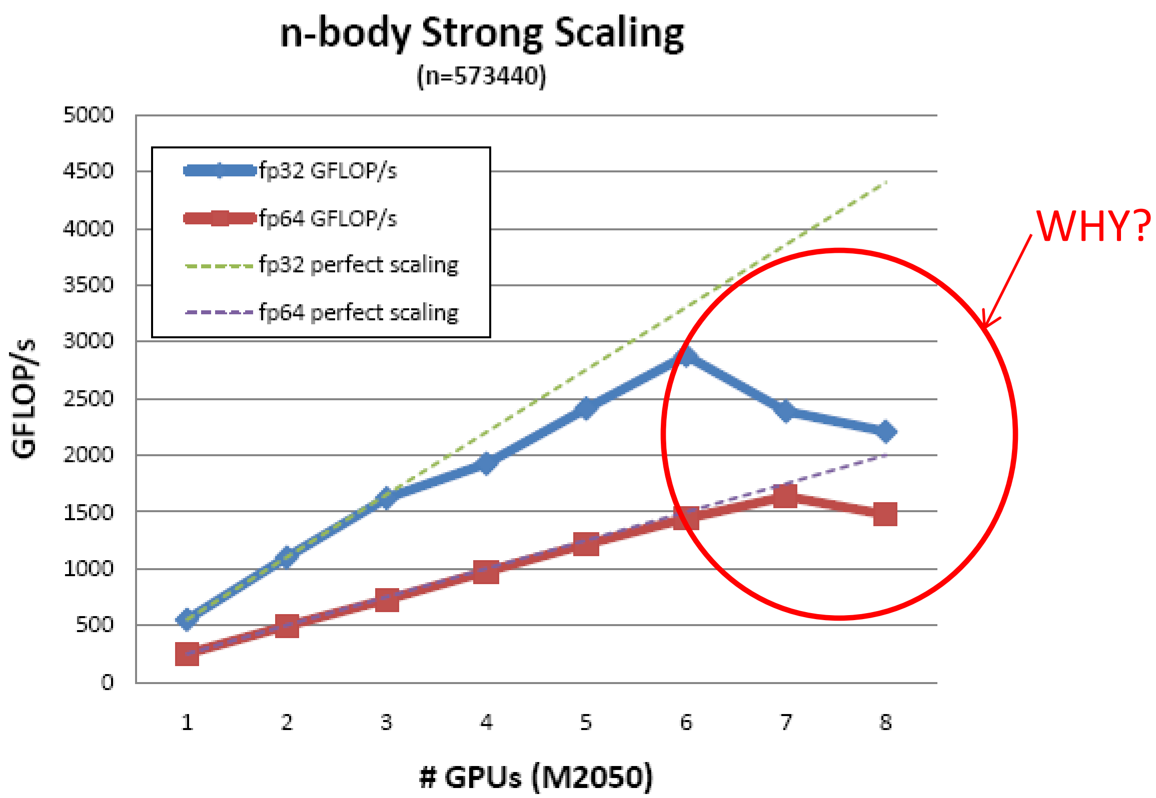
Example: Direct N-Body

- Sharing data among GPUs: options
  - Explicit copies via host
  - Zero-copy shared host array (cudaMallocHost())
  - Per-device peer-to-peer exchange transfers
    - UVA as we have seen
    - Non-UVA:
      - cudaMemcpyPeer()
      - Copies memory from one device to memory on another device
  - Peer-to-peer memory access

Example: Direct N-Body

- Sharing data among GPUs: options
  - Explicit copies via host
  - Zero-copy shared host array (cudaMallocHost())
  - Per-device peer-to-peer exchange transfers
  - Peer-to-peer memory access
    - Pass pointer to memory on device A to kernel running on device B
    - Requires UVA
    - Must first enable peer access for every pair:
    - cudaDeviceEnablePeerAccess

Example: Direct N-Body



Using zero-copy from host memory

Issues

- How to decompose your problem?
- Several copies versus data movement
- Dealing with multithreaded applications on CPU
- Coherence
- Homogeneous versus heterogeneous GPUs

Conclusions

- Multi-GPU system is an efficient way to reach higher performance
- GPUs have several ways of exchanging information among themselves
- Performance gain is application-dependent and programmer-dependent!