KOÇ UNIVERSITY

Programming on Multi-GPUs

Didem Unat

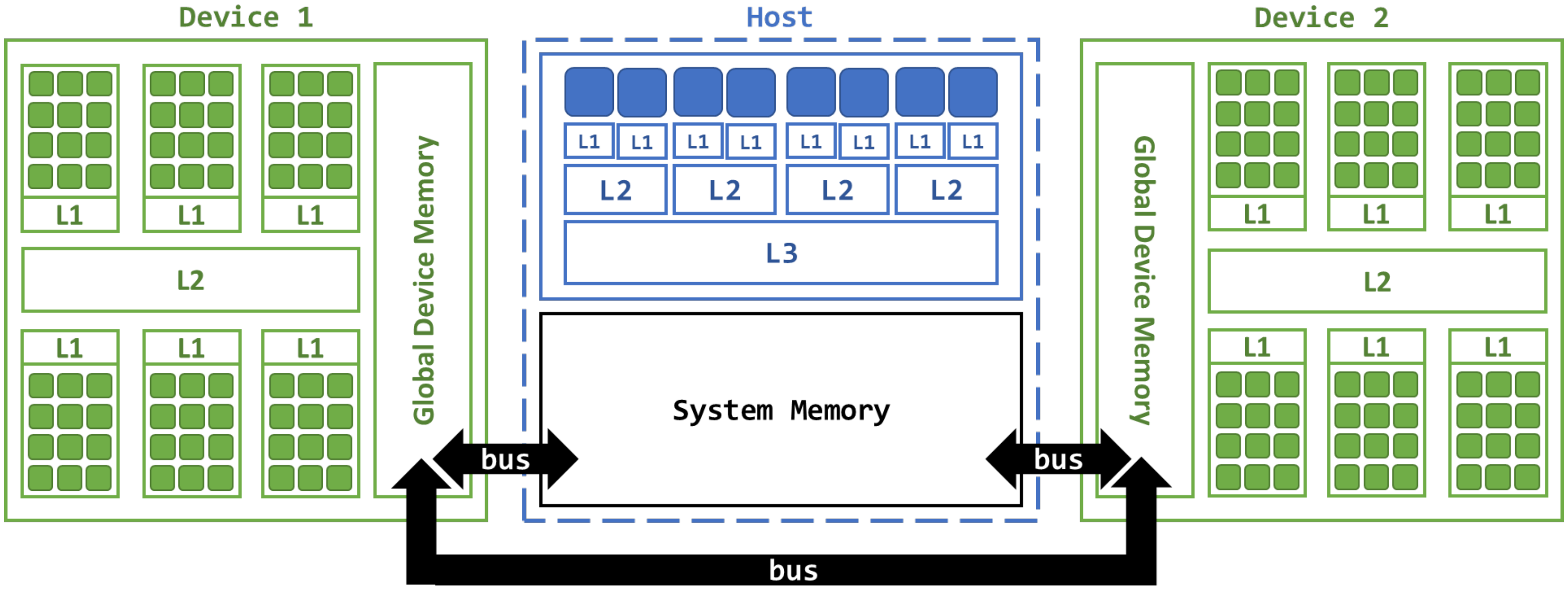
dunat@ku.edu.tr

http://parcorelab.ku.edu.tr

Programming Multiple GPUs

- Programming Multiple GPUs on a Host
  - Using a Single Thread
  - Using Multiple Threads
  - Using Multiple Processes
- Programming GPUs on Multiple Hosts
  - Only possible using multiple processes (with MPI)

Programming Multiple GPUs on a Host



- Split/distribute the application data
- Submit a kernel for each device
- Set a device before submitting operations
  - Check number of GPUs with: **`cudaGetDeviceCount`** ( )
  - Set a device with: **`cudaSetDevice()`**

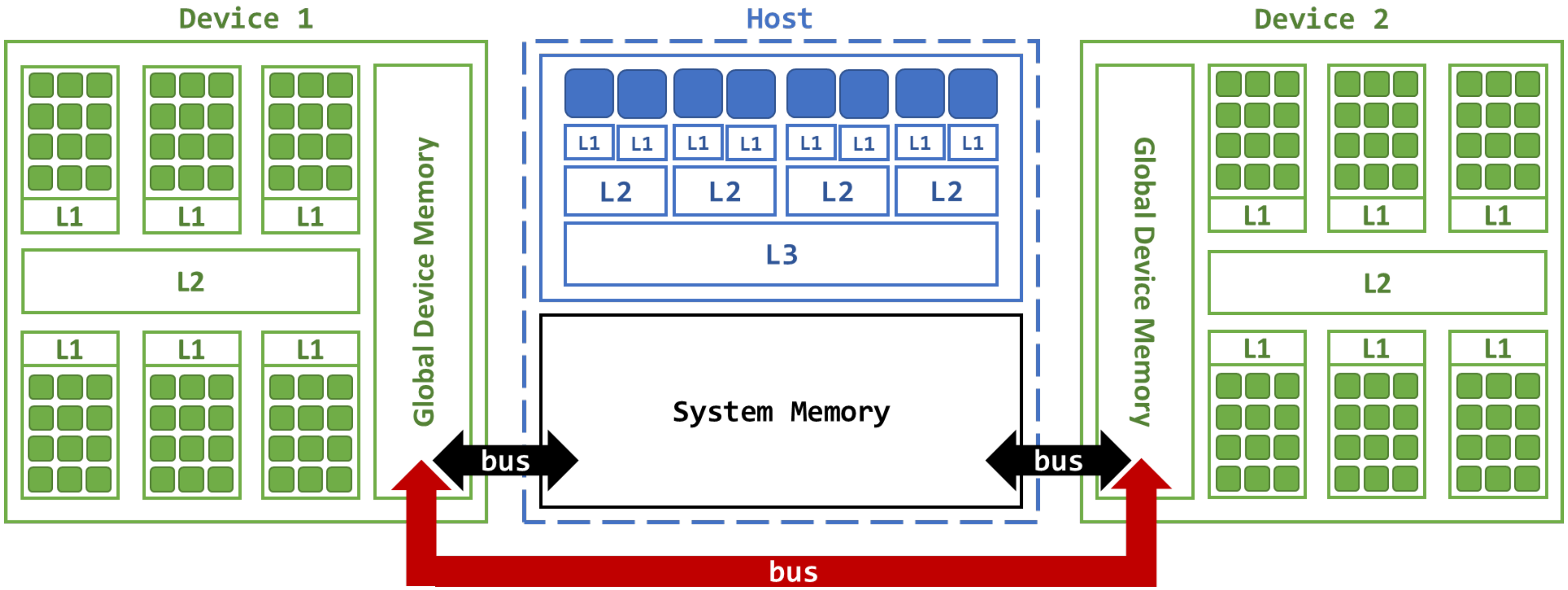
Using a Single Thread

```
...
size_t size = sizeof(double) * 8; // array size
...
double* h_arr; // host array
h_arr = (double*) malloc(size); // host allocation

cudaSetDevice(0);
double* d0_arr;
cudaMalloc((void **)&d0_arr, size/2); // allocation for device 0
cudaMemcpy(d0_arr, h_arr, size/2, cudaMemcpyHostToDevice);

cudaSetDevice(1);
double* d1_arr;
cudaMalloc((void **)&d1_arr, size/2); // allocation for device 1
cudaMemcpy(d1_arr, &h_arr[4], size/2, cudaMemcpyHostToDevice);
...
cudaSetDevice(0);
simpleAdd<<<blocksPerGrid, threadsPerBlock>>>(d0_arr, 4);

cudaSetDevice(1);
simpleAdd<<<blocksPerGrid, threadsPerBlock>>>(d1_arr, 4);
...
```

distribute data to each device

launch a kernel for each device

What if GPUs need to share data?



- Thread(s) can move data from one GPU to another through host
  - Not ideal, degrades performance with unnecessary host transfers
- If the system employs an interconnect between GPUs and supports GPUDirect, data can be shared directly
  - **GPUDirect Peer-to-Peer Direct Access:** directly access another GPUs memory
  - **GPUDirect P2P Direct Transfers:** directly transfer data from another GPU

GPUDirect Peer-to-Peer

- Check if P2P access is available with:
  - **`cudaDeviceCanAccessPeer()`**
- Enable P2P access with:
  - **`cudaDeviceEnablePeerAccess()`**
- P2P transfer data with:
  - `cudaMemcpy(..., `**`cudaMemcpyDeviceToDevice`**`)`
    - No need to specify a device number
    - Uses virtual addressing to identify where the data is located
  - `cudaMemcpyPeer()`
    - Need to indicate which device to transfer the data

Types of Communications

- Device to Device
  - Host-initiated Explicit Transfer(CUDA memcpy PtoP)
    - Case 1.1: P2P cudamemcpy with UVA
    - Case 1.2: P2P cudamemcpy with No-UVA
  - P2P Implicit Transfer
    - Case 2: Unified Virtual Addressing
      - Direct access to other GPU’s memory from the kernel with GPUDirect P2P
    - Case 3: Unified Virtual Memory (Memcpy DtoD)
  - Case 4: Through Host
- Host To Device and Device to Host (No other device)
  - Case 5: Memcpy (CUDA memcpy DtoH and CUDA memcpy HtoD)
  - Case 6: Unified Virtual Addressing
  - Case 7: Unified Memory (Unified Memory Memcpy HtoD and Unified Memory

Types of Communication

	Device-Device (Peer-to-Peer) Communication		Host-Device Communication
	Peer Access Enabled	**Peer Access Disabled**	
Explicit	**Case 1.1:** `cudaMemcpy` with UVA **Case 1.2:** `cudaMemcpyPeer` without UVA	**Case 1.3:** `cudaMemcpyPeer` & `cudaMemcpy` (implicit copies through host) **Case 1.4:** `cudaMemcpy` with H2D and D2H kinds	**Case 4:** `cudaMemcpy` with H2D, D2H or `cudaMemcpyDefault` kinds
Implicit	**Case 2:** Zero-copy Memory **Case 3:** Unified Memory		**Case 5:** Zero-copy Memory **Case 6:** Unified Memory
	(a)		(b)

UVA Example-1

```
/* Initial Setup */
// For UVA it is imp to enable peer access
 int can_access_peer_0_1, can_access_peer_1_0;
 cudaSetDevice(0);
 cudaDeviceEnablePeerAccess(&can_access_peer_0_1, 0, 1); //Device 0 can access Device 1
 cudaSetDevice(1);
 cudaDeviceEnablePeerAccess(&can_access_peer_1_0, 1, 0); //Device 1 can access Device 0

 // Allocate buffers
 const size_t buf_size = 1024 * 1024 * 16 * sizeof(float);
 cudaSetDevice(0);
 float *g0; //g0 allocated on Device 0
 cudaMalloc(&g0, buf_size);
 cudaSetDevice(1);
 float *g1; //g1 allocated on Device 1
 cudaMalloc(&g1, buf_size);
 //h0 allocated and initialized on Host
 float *h0;
 cudaMallocHost(&h0, buf_size); // Automatically portable with UVA
 for (int i=0; i<buf_size / sizeof(float); i++)
     h0[i] = float(i % 4096);
```

UVA Example -1 – cont.

```
    cudaSetDevice(0); //Set to Device 0
    cudaMemcpy(g0, h0, buf_size, cudaMemcpyDefault);

    // Kernel launch configuration
    const dim3 threads(512, 1);
    const dim3 blocks((buf_size / sizeof(float)) / threads.x, 1);

    // Run kernel on GPU 1, reading input from the GPU 0 buffer, writing
    // output to the GPU 1 buffer
    cudaSetDevice(1); //Set to Device 1
    SimpleKernel<<<blocks, threads>>>(g0, g1);
    cudaDeviceSynchronize();
```

HtoD memcpy using UVA (Case 6)

Data transferred from Host to GPU0

```
//Kernel Example
__global__ void SimpleKernel(float *src, float *dst)
{
    // Just a dummy kernel, doing enough for us to verify that everything
    // worked
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    //Here src is g0 and dst is g1
    dst[idx] = src[idx] * 2.0f;
}
```

Implicit PtoP memcpy (Case 2)

Data is transferred from GPU0 to GPU1

UVA Example -1 cont.

```
// Run kernel on GPU 0, reading input from the GPU 1 buffer, writing
// output to the GPU 0 buffer
cudaSetDevice(0); //Set to Device 0
SimpleKernel<<<blocks, threads>>>(g1, g0);
cudaDeviceSynchronize();
```

```
__global__ void SimpleKernel(float *src, float *dst)
{
    // Just a dummy kernel, doing enough for us to verify that everything
    // worked
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    //Here src is g1 and dst is g0
    dst[idx] = src[idx] * 2.0f;
}
```

Implicit PtoP memcpy (Case 2)

Data transferred from GPU1 to GPU0

```
// Copy data back to the host and verify
cudaMemcpy(h0, g0, buf_size, cudaMemcpyDefault);
for (int i=0; i<buf_size / sizeof(float); i++){
// Re-generate input data and apply 2x '* 2.0f' computation of both kernel runs
    if (h0[i] != float(i % 4096) * 2.0f * 2.0f) break;
}
```

DtoH memcpy using UVA (Case 6)

Data transferred from GPU0 to Host

P2P Memcpy with UVA Example

```
// P2P memcpy() benchmark
for (int i=0; i<100; i++){
// With UVA we don't need to specify source and target
devices, the runtime figures this out by itself from the
pointers
// Ping-pong copy between GPUs
    if (i % 2 == 0){
        cudaMemcpy(g1, g0, buf_size, cudaMemcpyDefault);
    }
    else
    {
        cudaMemcpy(g0, g1, buf_size, cudaMemcpyDefault);
    }
}
```

Explicit PtoP memcpy
(Case 1.1)

Data transferred from
GPU0 to GPU1

Explicit PtoP memcpy
(Case 1.1)

Data transferred from
GPU1 to GPU0

No UVA Example

```
// Allocate buffers
const size_t buf_size = 1024 * 1024 * 16 * sizeof(float);
// g0_src and g0_dst allocated on GPU 0
cudaSetDevice(0);
float *g0_src, *g0_dst;
cudaMalloc(&g0_src, buf_size);
cudaMalloc(&g0_dst, buf_size);

// g1_src and g1_dst allocated on GPU 1
cudaSetDevice(1);
float *g1_src, *g1_dst;
cudaMalloc(&g1_src, buf_size);
cudaMalloc(&g1_dst, buf_size);

//h0 allocated and initialized on Host
float *h0;
h0 = (float *)malloc(buf_size);
for (int i=0; i<buf_size / sizeof(float); i++)
    h0[i] = float(i % 4096);
```

Case 1.2 & 5: P2P Memcpy (No UVA) Example

```
cudaSetDevice(0); //Set to Device 0
//HtoD copy using cudaMemcpyHostToDevice instead of cudaMemcpyDefault
cudaMemcpy(g0_src, h0, buf_size, cudaMemcpyHostToDevice);

// Kernel launch configuration
const dim3 threads(512, 1);
const dim3 blocks((buf_size / sizeof(float)) / threads.x, 1);

cudaSetDevice(1); //Set to Device 1
//Perform P2P memcpy from Device 0 to Device 1 before kernel launch
cudaMemcpyPeer(g1_src, 1, g0_src, 0, buf_size);
SimpleKernel<<<blocks, threads>>>(g1_src, g1_dst);
checkCudaErrors(cudaDeviceSynchronize());

cudaSetDevice(0); //Set to Device 0
//Perform P2P memcpy from Device 1 to Device 0 before kernel launch
cudaMemcpyPeer(g0_src, 0, g1_dst, 1, buf_size);
SimpleKernel<<<blocks, threads>>>(g0_src, g0_dst);
cudaDeviceSynchronize();
```

HtoD memcpy (Case 5)
Data transferred from Host to GPU0

Explicit PtoP memcpy (Case 1.2)
Data transferred from GPU0 to GPU1

Explicit PtoP memcpy (Case 1.2)
Data transferred from GPU1 to GPU0

No UVA Example

```
// Copy data back to the host and verify
//DtoH copy using cudaMemcpyDeviceToHost instead of
cudaMemcpyDefault
cudaMemcpy(h0, g0_dst, buf_size, cudaMemcpyDeviceToHost);

//Verification on Host
int error_count = 0;
for (int i=0; i<buf_size / sizeof(float); i++)
{
  // Re-generate input data and apply 2x '* 2.0f' computation of
both
  // kernel runs
  if (h0[i] != float(i % 4096) * 2.0f * 2.0f)
  {
    if (error_count++ > 10)
    {
        break;
    }
  }
}
```

DtoH memcpy
(Case 5)

Data transferred
from GPU0 to Host

UVM Example

```
const size_t buf_size = 1024 * 1024 * 16 * sizeof(float); //Allocate Buffers
float *g0;
cudaMallocManaged(&g0, buf_size);
float *g1;
cudaMallocManaged(&g1, buf_size);

for (int i=0; i<buf_size / sizeof(float); i++)
{ g0[i] = float(i % 4096); }

// Kernel launch configuration
const dim3 threads(512, 1);
const dim3 blocks((buf_size / sizeof(float)) / threads.x, 1);

// Run kernel on GPU 1
cudaSetDevice(1)); // Set to Device 1
SimpleKernel<<<blocks, threads>>>(g0, g1);
cudaDeviceSynchronize();
```

Buffer allocated and initialized at host in unified memory

```
__global__ void SimpleKernel(float *src, float *dst){
    // Just a dummy kernel, doing enough for us to verify that everything
    // worked
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    //Both src and dst are transferred to Device 1 on CPU page fault
    dst[idx] = src[idx] * 2.0f;
}
```

Implicit HtoD Unified Memory Memcpy (Case 7)
g0 and g1 transferred from Host to Device 1

UVM Example – cont.

```
// Run kernel on GPU 0
cudaSetDevice(0); //Set to Device 0
SimpleKernel<<<blocks, threads>>>(g1, g0);
cudaDeviceSynchronize();
```

```
__global__ void SimpleKernel(float *src, float *dst){
 // Just a dummy kernel, doing enough for us to verify that everything worked
 const int idx = blockIdx.x * blockDim.x + threadIdx.x;
 //Both src and dst are brought from Device 1 to Device 0 on GPU page fault
    dst[idx] = src[idx] * 2.0f;
}
```

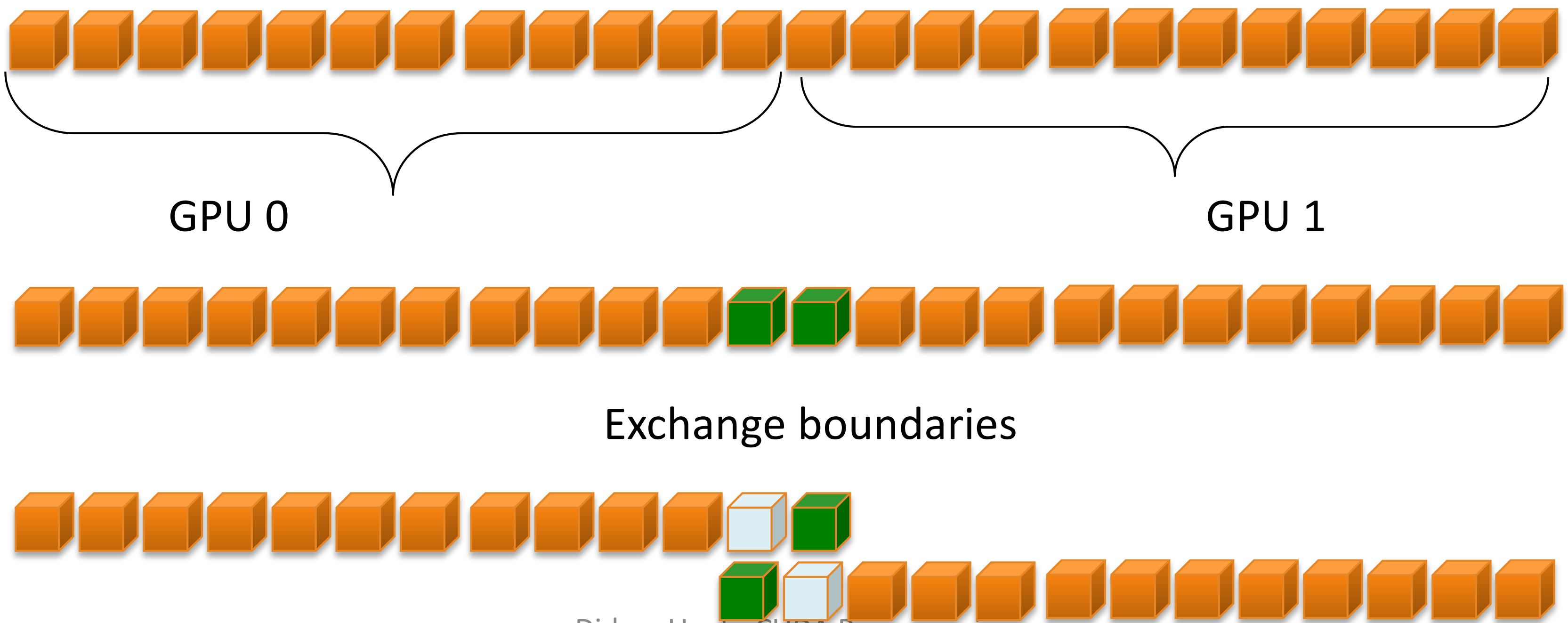
Implicit PtoP Unified Memory memcpy (Case 3)

```
int error_count = 0;
for (int i=0; i<buf_size / sizeof(float); i++)
{// Re-generate input data and apply 2x '* 2.0f' computation of both kernel runs
 if (g0[i] != float(i % 4096) * 2.0f * 2.0f)
 { if (error_count++ > 10) break;}
}
```

Implicit DtoH Unified Memory Memcpy (Case 7)
g0 transferred from Device 0 to Host

Lab 10 – Multi-GPU 1D Jacobi

- Simple 1D Jacobi to illustrate the peer to peer communication between two GPU devices
- Input data is divided between two GPUs
- Every iteration, GPUs send single data point to each other



Lab 10 – Multi-GPU 1D Jacobi

- The code is missing some sections in ***runJacobi1DCUDA***
  - Need to add necessary **malloc** and **cudaMemcpy** for Device #1
    - For Device 0, these are already set for you as an example
  - Need to add kernel launch for Device 1
    - See the example for Device 0
  - Need to synchronize before you perform communication
  - Next, you need to perform halo updates between two devices
    - See next slide for details

Lab 10 – Multi-GPU 1D Jacobi

- The code is missing the necessary data transfer between the GPUs.
- Fix the code so that
  - First it performs transfers with peer transfer using **cudaMemcpyPeer**
  - Second it performs transfers with memory copy using **cudaMemcpy**
  - Finally it performs the transfers through the host
- Usage for cudaMemcpyPeer
  - http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_g046702971bc5a66d9bc6000682a6d844.html
- Usage for cudaMemcpy
  - http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html#g48efa06b81cc031b2aa6fdc2e9930741

Using Multiple Threads

```
// thread 0
cudaSetDevice(0);
double* d_arr;
cudaMalloc((void **)&d_arr, size/2);
cudaMemcpy(d_arr, h_arr, size/2, cudaMemcpyHostToDevice);
...
simpleAdd<<<blocksPerGrid, threadsPerBlock>>>(d_arr, 4);
...
```

thread 0 gets its own GPU

```
// thread 1
cudaSetDevice(1);
double* d_arr;
cudaMalloc((void **)&d_arr, size/2);
cudaMemcpy(d_arr, &h_arr[4], size/2,
cudaMemcpyHostToDevice);
...
simpleAdd<<<blocksPerGrid, threadsPerBlock>>>(d_arr, 4);
...
```

thread 1 gets its own GPU

Multi-GPUs with Multi-threads

- We can access multiple GPUs with multiple threads using OpenMP

```
#pragma omp parallel num_threads(2)
 {
   int tid = omp_get_thread_num();

   if (tid == 0)
      // do something
   else
      // do something
 }
```

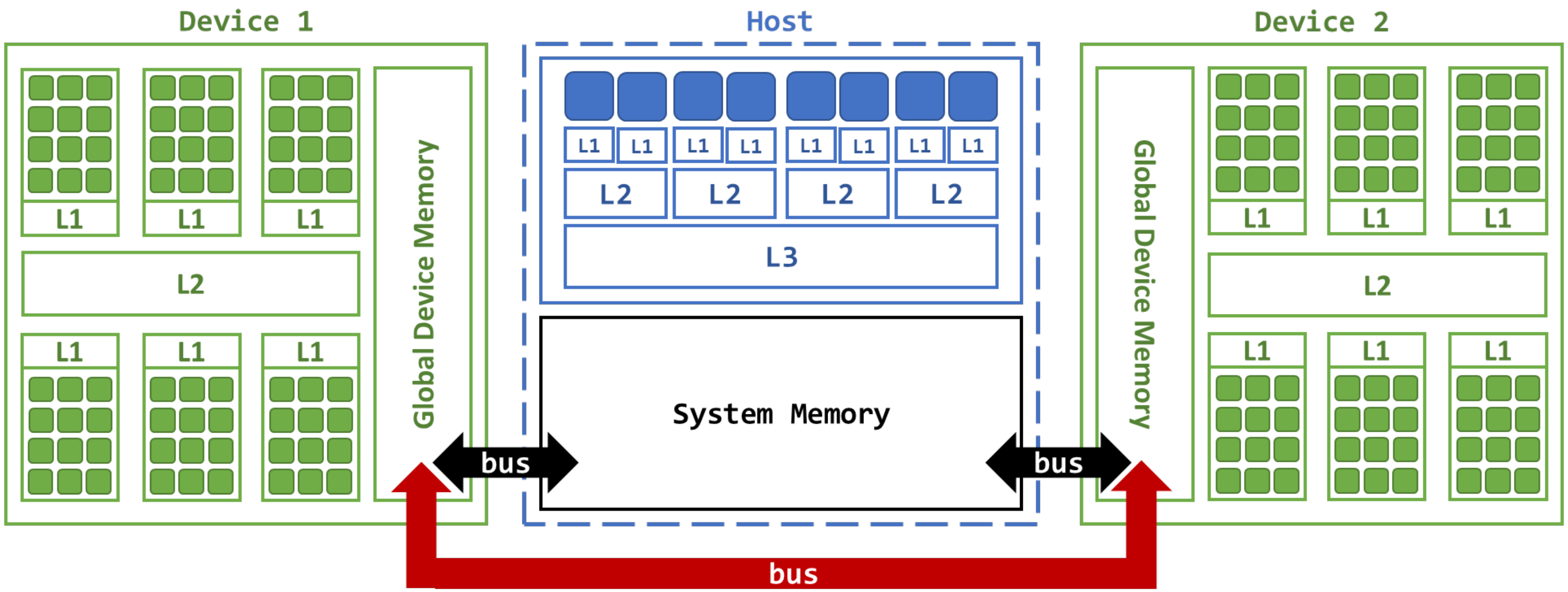
Outline

- ✓ Programming Multiple GPUs on a Host
  - ✓ Using a Single Thread
  - ✓ Using Multiple Threads
  - ✓ **Using Multiple Processes**
- ➢ Programming GPUs on Multiple Hosts

Using Multiple Processes on a Host

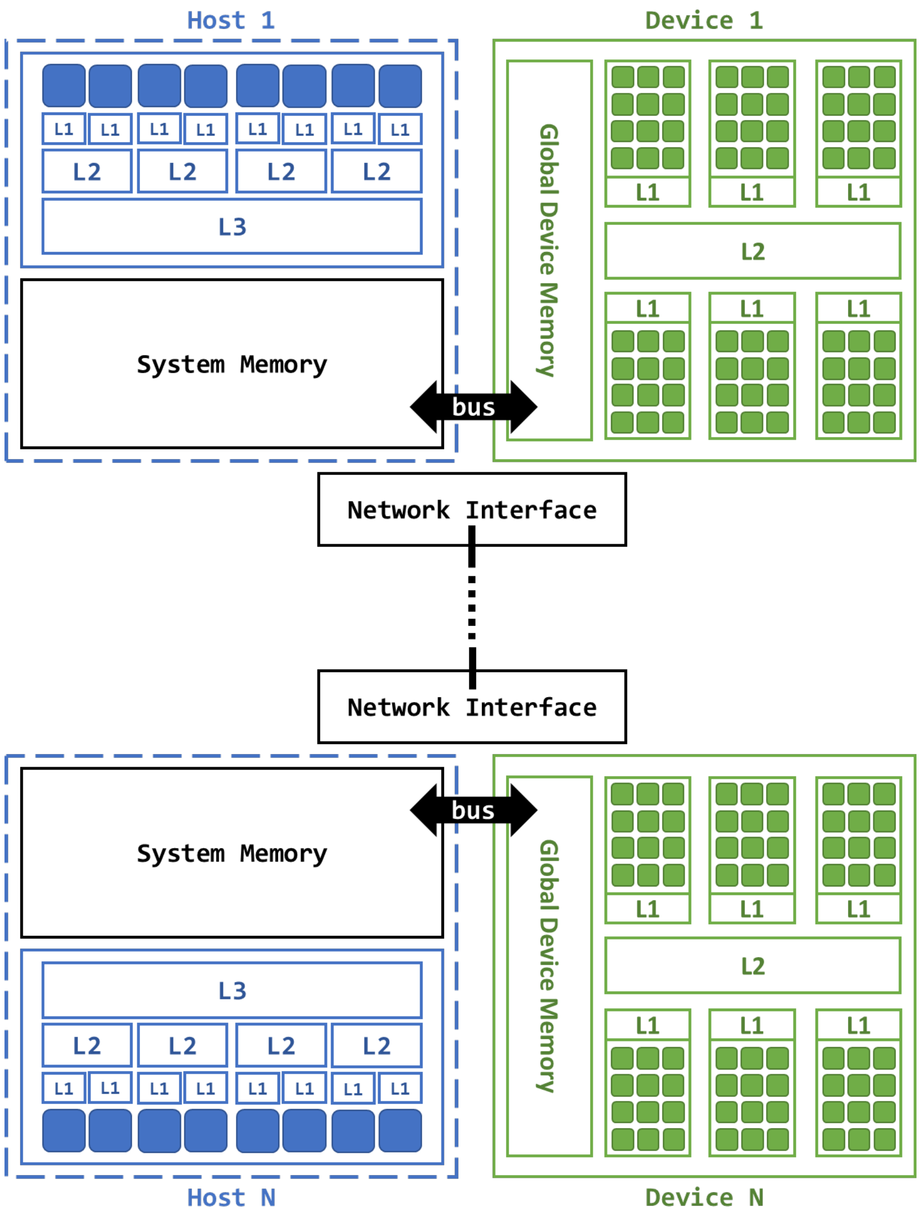
- Similar to programming with multiple threads but
  - Each process either gets its own GPU
  - Or shares a GPU
  - Use message passing (MPI) for communication among processes

What if GPUs need to share data?



- Processes have their own memory address space
  - They cannot directly share pointers as threads
- CUDA Inter-Process Communication (IPC) helps share device memory pointers across processes
  - Get an IPC handle with: **`cudaIpcGetMemHandle()`**
  - Open IPC handle with: **`cudaIpcOpenMemHandle()`**

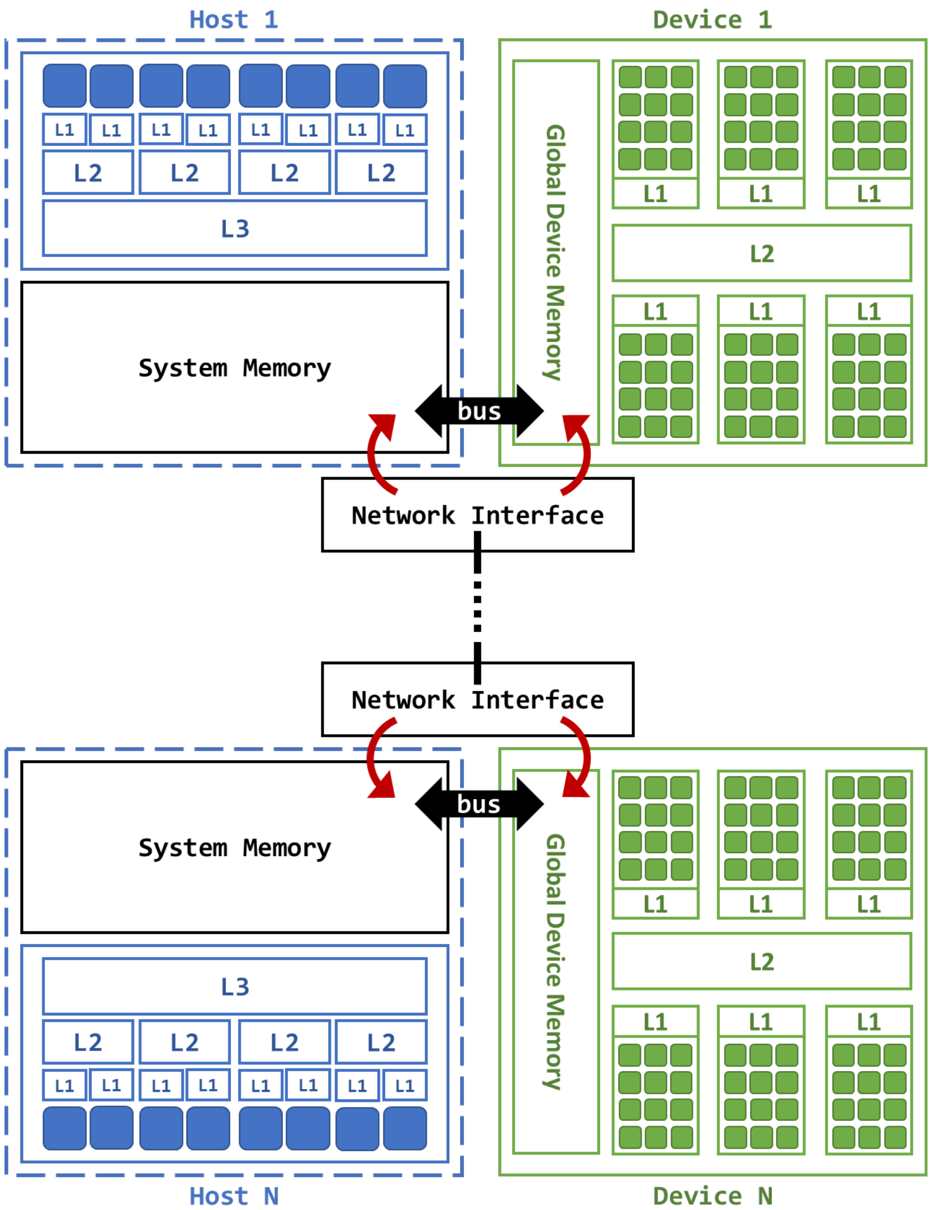
Programming GPUs on Multiple Hosts



- Processes are the only option for programming GPUs on multiple hosts
  - Combine CUDA with MPI
- Communication happens through network interfaces

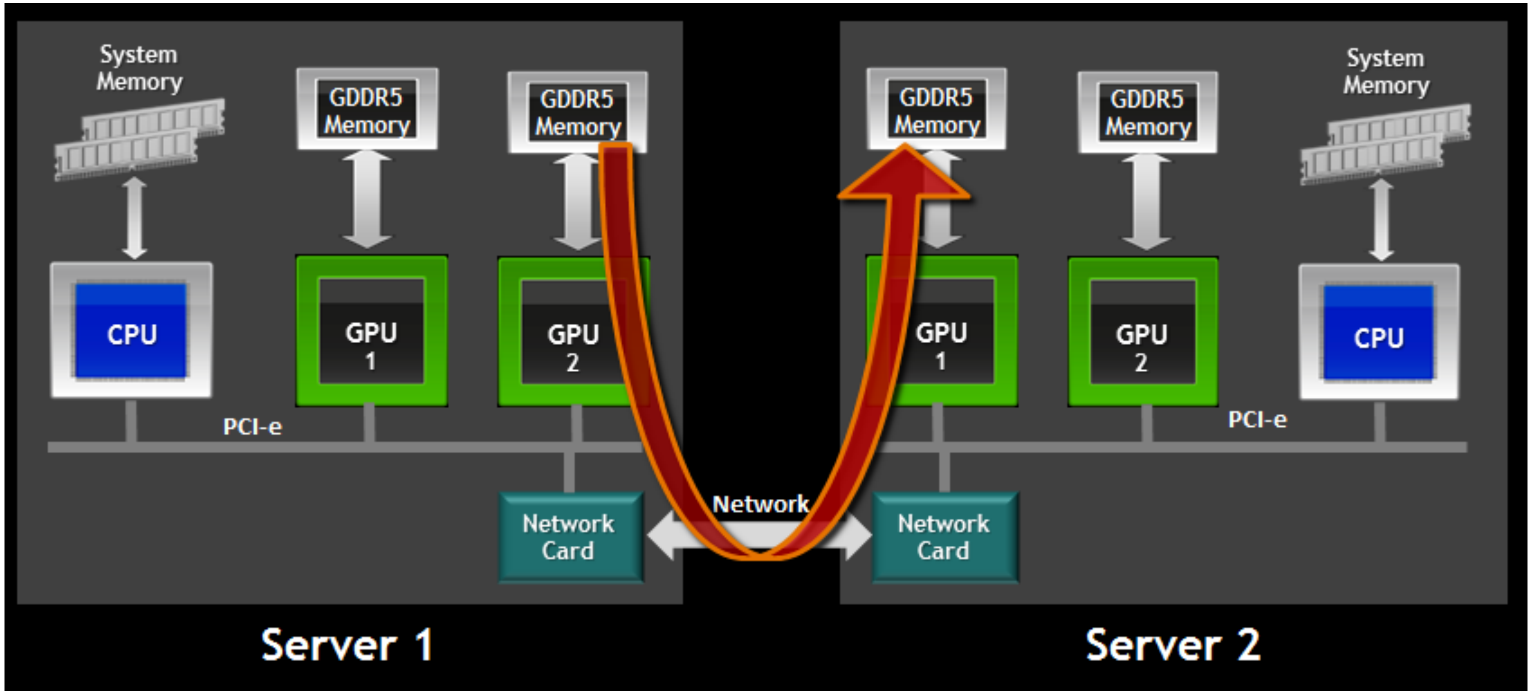
GPUDirect Remote Direct Memory Access (RDMA)

- Specific network interfaces with GPUDirect RDMA support helps avoid unnecessary host transfers
  - It is low-level!
  - There are CUDA-aware MPI implementations
  - Programmer doesn't deal with IPC, etc.



GPUDirect RDMA

- Prevents unnecessary transfers to hosts with
  - Remote Direct Memory Access (RDMA)
  - GPUDirect (pinned to pageable copy on host)



src: NVIDIA

GPU-Aware MPI Library

- Transfer data directly to/from CUDA device memory via MPI calls
- Standard MPI interfaces used for unified data movement
- Takes advantage of Unified Virtual Addressing (>= CUDA 4.0)
- Overlaps data movement from GPU with RDMA transfers
- Avoids copies through the host

At Sender:

```
MPI_Send(s_devbuf, size, …);
```

At Receiver:

```
MPI_Recv(r_devbuf, size, …);
```

Conclusion

- ✓ Programming Multiple GPUs on a Host
  - ✓ Using a Single Thread
    - ✓ Overlap is possible but code size increases
  - ✓ Using Multiple Threads
    - ✓ Use OpenMP
  - ✓ Using Multiple Processes
    - ✓ Can use Multi-process service (MPS) if processes need to share a GPU
  - ✓ **GPUDirect Peer-to-Peer Direct Access:** directly access another GPUs memory
  - ✓ **GPUDirect P2P Direct Transfers:** directly transfer data from another GPU
- ➢ Programming GPUs on Multiple Hosts
  - ➢ Need to use MPI
  - ➢ Transfers to GPU -> host -> host -> GPU
  - ➢ GPUDirect RDMA enables GPU <-> GPU transfers