高性能计算II(B)

基于图形处理器的并行计算及CUDA编程

Ying Liu, Associate Prof., Ph.D

School of Computer and Control, University of Chinese Academy of Sciences

Key Lab of Big Data Mining & Knowledge Management, Chinese Academy of Sciences

Outline

- Brief review of the scenarios
- Single CPU process, multiple GPUs
- Multiple processes

Several Scenarios

- We assume CUDA 4.0 or later
  - Simplifies multi-GPU programming
- Working set is decomposed across GPUs
  - Reasons:
    - To speedup computation
    - Working set exceeds single GPU's memory
  - Inter-GPU communication is needed
- Two cases:
  - GPUs within a single network node
  - GPUs across network nodes

Outline

- Brief review of the scenarios
- Single CPU process, multiple GPUs
- Multiple processes

Multiple GPUs within a Node

- GPUs can be controlled by:
  - A single CPU thread
  - Multiple CPU threads belonging to the same process
  - Multiple CPU processes
- Definitions used:
  - CPU process has its own address space
  - A process may spawn several threads, which can share address space

Single CPU Thread – Multiple GPUs

- All CUDA calls are issued to the current GPU
  - One exception: asynchronous peer-to-peer memcopies
- cudaSetDevice() sets the current GPU
- Asynchronous calls (kernels, memcopies) don't block switching the GPU
  - The following code will have both GPUs executing concurrently:

```
cudaSetDevice( 0 );
kernel<<<...>>>(...);
cudaSetDevice( 1 );
kernel<<<...>>>(...);
```

Devices, Streams, and Events

- CUDA streams and events are per device (GPU)
  - Determined by the GPU that is current at the time of their creation
  - Each device has its own default stream (0 or NULL stream)
- Using streams and events
  - Calls to a stream can be issued only when its device is current
  - Event can be recorded only to a stream of the same device
- Synchronization/query:
  - It is OK to synchronize with or query any event/stream
    - Even if stream/event belong to one device and a different device is current

Single CPU thread – Multiple GPUs

- Example 1

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

cudaSetDevice( 0 );
cudaStreamCreate( &streamA );      // streamA and eventA belong to device-0
cudaEventCreate( &eventA );

cudaSetDevice( 1 );
cudaStreamCreate( &streamB );      // streamB and eventB belong to device-1
cudaEventCreate( &eventB );

kernel<<<..., streamB>>>(...);
cudaEventRecord( eventB, streamB );
cudaEventSynchronize( eventB );
```

OK:

- *device-1 is current*
- *eventB and streamB belong to device-1*

Single CPU thread – Multiple GPUs

- Example 2

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

cudaSetDevice( 0 );
cudaStreamCreate( &streamA );      // streamA and eventA belong to device-0
cudaEventCreate( &eventA );

cudaSetDevice( 1 );
cudaStreamCreate( &streamB );      // streamB and eventB belong to device-1
cudaEventCreate( &eventB );

kernel<<<..., streamA>>>(...);
cudaEventRecord( eventB, streamB );
cudaEventSynchronize( eventB );
```

ERROR:
- *device-1 is current*
- *streamA belongs to device-0*

Single CPU thread – Multiple GPUs

- Example 3

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

cudaSetDevice( 0 );
cudaStreamCreate( &streamA );      // streamA and eventA belong to device-0
cudaEventCreate( &eventA );

cudaSetDevice( 1 );
cudaStreamCreate( &streamB );      // streamB and eventB belong to device-1
cudaEventCreate( &eventB );

kernel<<<..., streamB>>>(...);
cudaEventRecord( eventA, streamB );
```

ERROR:
- *eventA belongs to device-0*
- *streamB belongs to device-1*

Single CPU thread – Multiple GPUs

- Example 4

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

cudaSetDevice( 0 );
cudaStreamCreate( &streamA );      // streamA and eventA belong to device-0
cudaEventCreate( &eventA );

cudaSetDevice( 1 );
cudaStreamCreate( &streamB );      // streamB and eventB belong to device-1
cudaEventCreate( &eventB );

kernel<<<..., streamB>>>(...);
cudaEventRecord( eventB, streamB );

cudaSetDevice( 0 );
cudaEventSynchronize( eventB );
kernel<<<..., streamA>>>(...);
```

OK:

- *device-0 is current*
- *synchronizing/querying events/streams of other devices is allowed*
- *here, device-0 won't start executing the kernel until device-1 finishes its kernel*

Unified Addressing (CUDA 4.0+)

- CPU and GPU allocations use unified virtual address space
  - Think of each one (CPU, GPU) getting its own range of virtual addresses
    - Thus, driver/device can determine from the address where data resides
    - Allocation still resides on a single device (can't allocate one array across several GPUs)
  - Requires:
    - 64-bit Linux or 64-bit Windows with TCC driver
    - Fermi or later architecture GPUs (compute capability 2.0 or higher)
    - CUDA 4.0 or later
- A GPU can dereference a pointer that is:
  - an address on another GPU
  - an address on the host (CPU)

UVA and Multi-GPU Programming

- Two interesting aspects:
  - Accessing another GPU's addresses
  - Peer-to-peer (P2P) memcopies
- Both require and peer-access to memory be enabled:
  - cudaDeviceEnablePeerAccess(peer_device, 0)
    - Enables current GPU to access addresses on peer_device GPU
  - cudaDeviceCanAccessPeer(&accessible, dev_X, dev_Y)
    - Checks whether dev_X can access memory of dev_Y
    - Returns 0/1 via the first argument
    - Peer-access is not available if:
      - One of the GPUs is pre-Fermi
      - GPUs are connected to different Intel IOH chips on the motherboard
      - QPI and PCIe protocols disagree on P2P

UVA and Multi-GPU Programming

- Example 5

```
int gpu1 = 0;
int gpu2 = 1;

cudaSetDevice( gpu1 );
cudaMalloc( &d_A, num_bytes );

int accessible = 0;
cudaDeviceCanAccessPeer( &accessible, gpu2, gpu1 );
if( accessible )
{
    cudaSetDevice( gpu2 );
    cudaDeviceEnablePeerAccess( gpu1, 0 );
    kernel<<<...>>>( d_A);
}
```

Even though kernel executes on gpu2, it will access (via PCIe) memory allocated on gpu1

UVA and Multi-GPU Programming

- Peer-to-peer memcopy
  - cudaMemcpyPeerAsync(void* dst_addr, int dst_dev, void* src_addr, int src_dev, size_t num_bytes, cudaStream_t stream)
    - Copies the bytes between two devices
    - Stream must belong to the source GPU
    - There is also a blocking (as opposed to Async) version
  - If peer-access is enabled:
    - Bytes are transferred along the shortest PCIe path
    - No staging through CPU memory
  - If peer-access is not available
    - CUDA driver stages the transfer via CPU memory

How Does P2P Memcopy Help Multi-GPU?

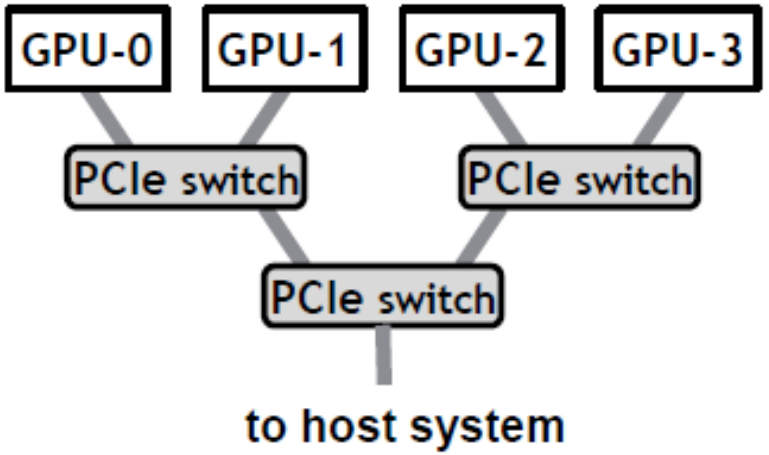
- Ease of programming
  - No need to manually maintain memory buffers on the host for inter-GPU exchanges
- Performance
  - Especially when GPUs connected to a PCIe switch:
    - Single-directional transfers achieve up to ~6.6 GB/s
    - Duplex transfers achieve ~12.2 GB/s
    - 4-5 GB/s if going through the host
  - Disjoint GPU-pairs can communicate without competing for bandwidth

Example: 1D Domain Decomposition and P2P

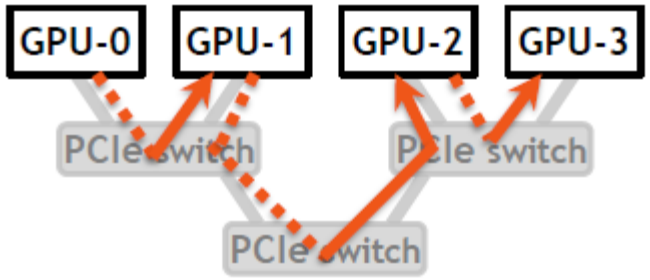
- Each subdomain has at most two neighbors
  - “left”/”right”
  - Communication graph = path
- GPUs are physically arranged into a tree(s)
  - GPUs can be connected to a PCIe switch
  - PCIe switches can be connected to another switch
- A path can be efficiently mapped onto a tree
  - Multiple exchanges can happen without contending for the same PCIe links
  - Aggregate exchange throughput:
    - Approaches (PCIe bandwdith) * (number of GPU pairs)
    - Typical achieved PCIe gen2 simplex bandwidth on a single link: 6 GB/s

Example: 4-GPU Topology

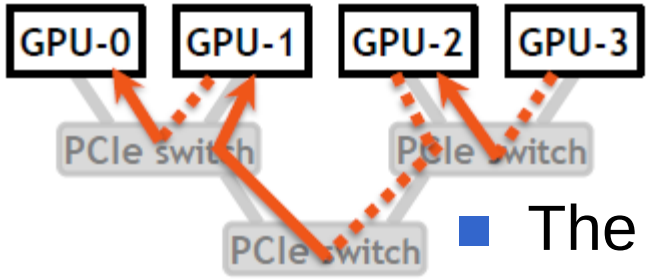
- Two ways to implement 1D exchange
  - Left-right approach
  - Pairwise approach
  - Both require two stages



Example: Left-Right Approach for 4 GPUs



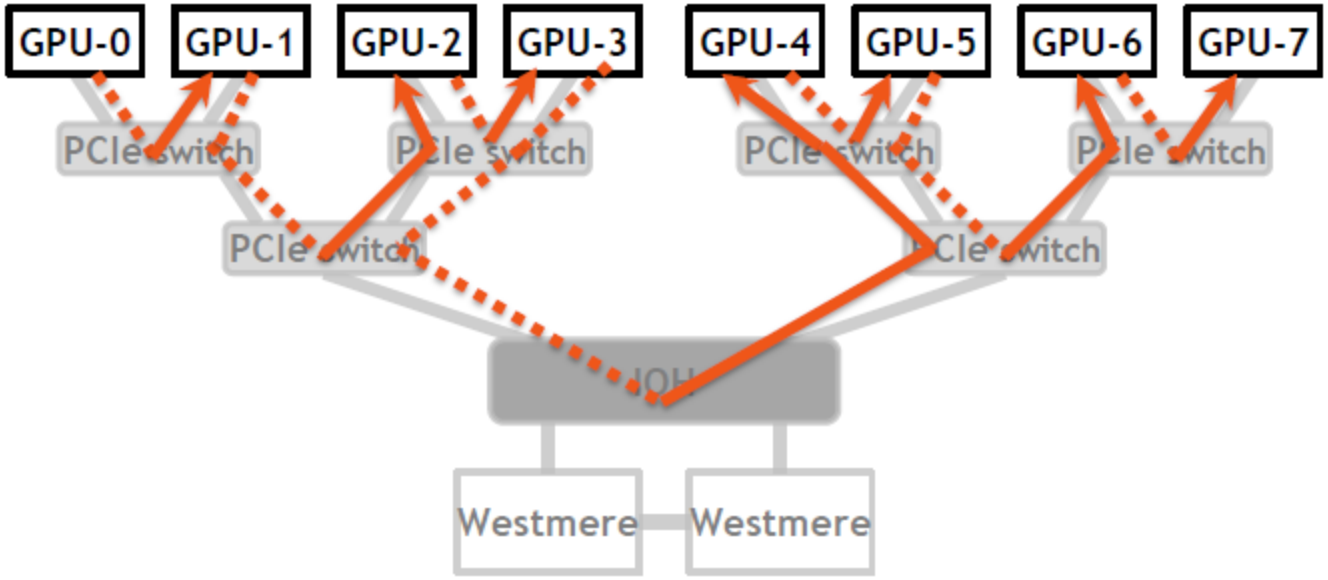
Stage 1: send “right” / receive from “left”



Stage 2: send “left” / receive from “right”

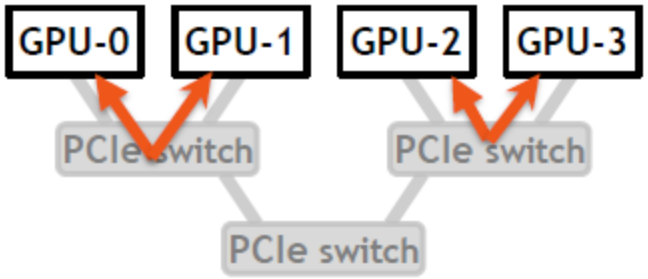
- The 3 transfers in a stage happen concurrently
  - Achieved throughput: ~15 GB/s (4-MB messages)
- No contention for PCIe links
  - PCIe links are duplex
  - Note that no link has 2 communications in the same “direction”

Example: Left-Right Approach for 8 GPUs

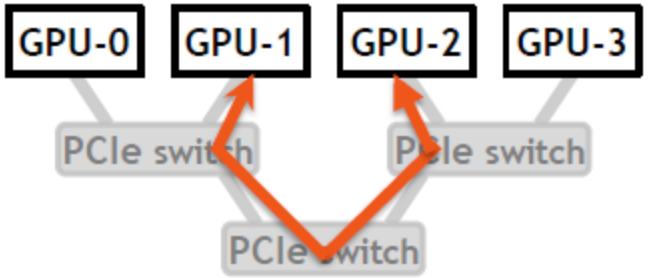


- Stage 1 shown above (Stage 2 is basically the same)
- Achieved aggregate throughput: ~34 GB/s

Example: Pairwise Approach for 4 GPUs



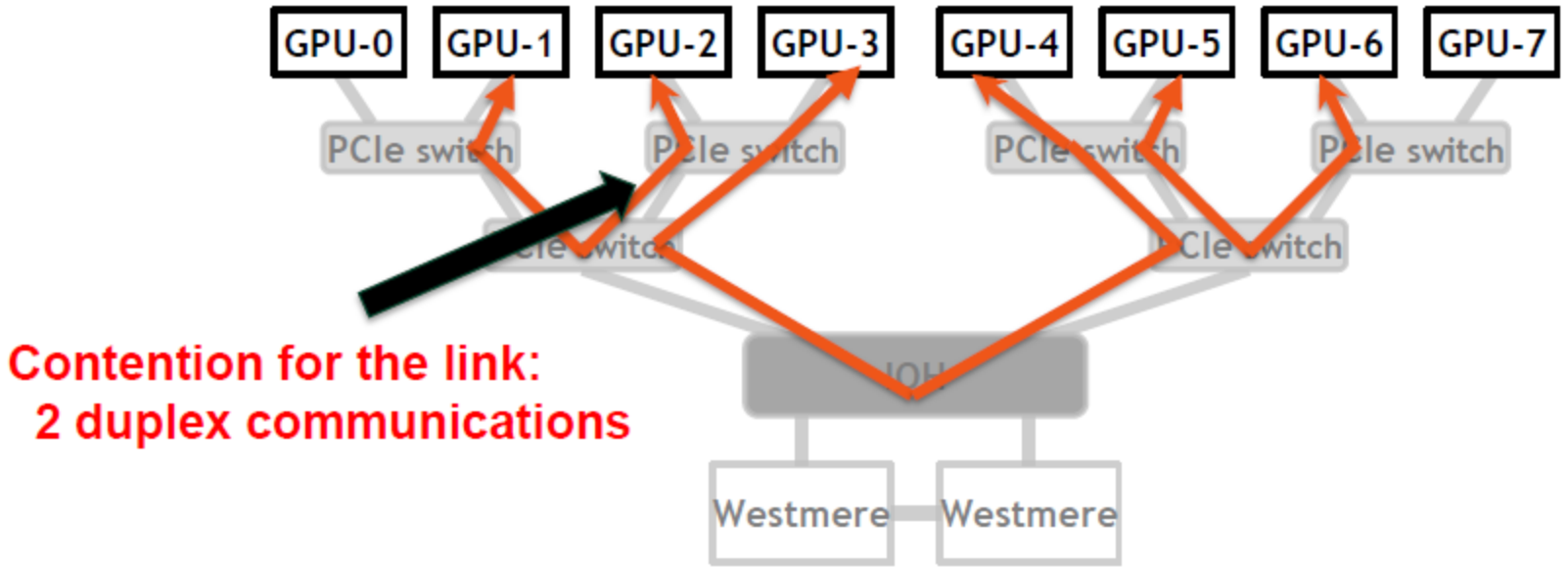
Stage 1: even-odd pairs



Stage 2: odd-even pairs

- No contention for PCIe links
  - All transfers are duplex, PCIe links are duplex
  - Note that no link has more than 1 exchange
    - Not true for 8 or more GPUs

Example: Even-Odd Stage of Pairwise Approach for 8 GPUs



- Odd-even stage:
  - Will always have contention for 8 or more GPUs
- Even-odd stage:
  - Will not have contention

1D Communication

- Pairwise approach slightly better for 2-GPU case
- Left-Right approach better for the other cases

Code for the Left-Right Approach

```
for( int i=0; i<num_gpus-1; i++ ) // “right” phase
    cudaMemcpyPeerAsync( d_a[i+1], device[i+1], d_a[i], device[i],
    num_bytes, stream[i] );
for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream[i] );
for( int i=1; i<num_gpus; i++ ) // “left” phase
    cudaMemcpyPeerAsync( d_b[i-1], device[i-1], d_b[i], device[i],
    num_bytes, stream[i] );
```

- Code assumes that addresses and GPU IDs are stored in arrays
- The middle loop isn’t necessary for correctness
  - Improves performance by preventing the two stages from interfering with each other (15 vs 11 GB/s for the 4-GPU example)

Summary for Single CPU-Thread/Multiple-GPUs

- CUDA calls are issued to the current GPU
  - Pay attention to which GPUs streams and events belong
- GPUs can access each other's memory
  - Keep in mind that still at PCIe latency/bandwidth
- P2P memcopies between GPUs enable high aggregate throughputs
  - P2P not possible for GPUs connected to different IOH chips
- Try to overlap communication and computation
  - Issue to different streams

Outline

- Brief review of the scenarios
- Single CPU process, multiple GPUs
- Multiple processes

Multiple Threads/Processes

- Multiple threads of the same process
  - Communication is same as single-thread/multiple-GPUs
- Multiple processes
  - Processes have their own address spaces
    - No matter if they’re on the same or different nodes
  - Some type of CPU-side message passing (MPI, ...) will be needed
    - Exactly the same as you would use on non-GPU code

Example: cudaOpenMP

```
__global__ void kernelAddConstant(int *g_a, const int b) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    g_a[idx] += b;
}
```

Example: cudaOpenMP (Cont.)

```
int num_gpus = 0;
printf("%s Starting...\n\n", argv[0]);
cudaGetDeviceCount(&num_gpus);
printf("number of host CPUs:\t%d\n", omp_get_num_procs());
printf("number of CUDA devices:\t%d\n", num_gpus);
unsigned int n = num_gpus * 8192;
unsigned int nbytes = n * sizeof(int);
int *a = 0;
int b = 3;
a = (int *)malloc(nbytes);
for (unsigned int i = 0; i < n; ++i)
    a[i] = i;
```

Example: cudaOpenMP (Cont.)

```
omp_set_num_threads(2 * num_gpus);
#pragma omp parallel
{
    unsigned int cpu_thread_id = omp_get_thread_num();
    unsigned int num_cpu_threads = omp_get_num_threads();

    int gpu_id = -1;
    checkCudaErrors(cudaSetDevice(cpu_thread_id % num_gpus));
    checkCudaErrors(cudaGetDevice(&gpu_id));

    int *d_a = 0;
    int *sub_a = a + cpu_thread_id * n / num_cpu_threads;
    unsigned int nbytes_per_kernel = nbytes / num_cpu_threads;
    dim3 gpu_threads(128);
    dim3 gpu_blocks(n / (gpu_threads.x * num_cpu_threads));
```

Example: cudaOpenMP (Cont.)

```
    checkCudaErrors(cudaMalloc((void **)&d_a, nbytes_per_kernel));
    checkCudaErrors(cudaMemset(d_a, 0, nbytes_per_kernel));
    checkCudaErrors(cudaMemcpy(d_a, sub_a, nbytes_per_kernel, cudaM
emcpyHostToDevice));
     kernelAddConstant<<<gpu_blocks, gpu_threads>>>(d_a, b);


     checkCudaErrors(cudaMemcpy(sub_a, d_a, nbytes_per_kernel, cudaM
emcpyDeviceToHost));
     checkCudaErrors(cudaFree(d_a));
}
```

Example: cudaOpenMP (Cont.)

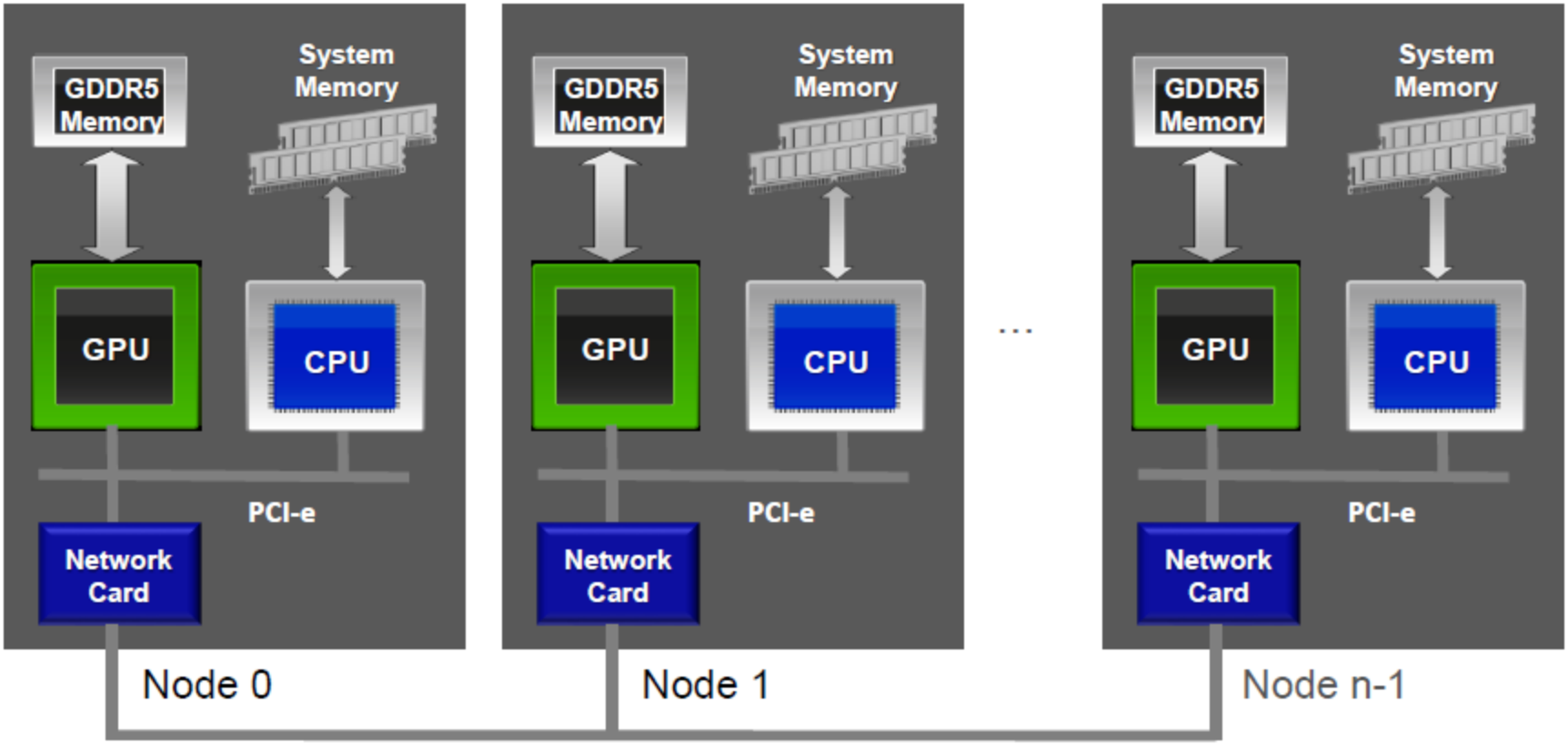
```
if (cudaSuccess != cudaGetLastError())
    printf("%s\n", cudaGetErrorString(cudaGetLastError()));

bool bResult = correctResult(a, n, b);

if (a) free(a);

cudaDeviceReset();

exit(bResult ? EXIT_SUCCESS : EXIT_FAILURE);
```

Multiple Threads/Processes

- MPI + CUDA
  - CUDA and MPI can be considered separate entities
    - CUDA handles parallelization on GPU
    - MPI handles parallelization over nodes
  - Use one MPI process per GPU and accelerate the computational kernels with CUDA
  - To transfer data between to devices
    - Sender: Copy data from device to temporary host buffer
    - Sender: Send host buffer data
    - Receiver: Receive data to host buffer
    - Receiver: Copy data to device

Multiple Threads/Processes

- MPI + CUDA



Multiple Threads/Processes

- MPI + CUDA
  - One MPI process per GPU
    - GPU handling is straight forward
    - Wastes the other cores of the processor
  - Many MPI processes share a GPU
    - Two processes cannot share the same GPU context, per process memory on GPU
    - Sharing may not always be possible
      - Limited memory on GPU
      - If GPUs are in exclusive mode

Multiple Threads/Processes

- Selecting GPUs
  - The number of active GPUs visible to the rank is
    - cudaGetDeviceCount(&deviceCount);
  - Divide GPUs to processes
    - id= rank%deviceCount;
    - cudaSetDevice(id);
  - Transfer

```
if (rank==0){
    cudaMemcpy(hBuffer,dBuffer,size,cudaMemcpyDeviceToHost);
    MPI_Send(hBuffer,size,MPI_BYTE,1,100,MPI_COMM_WORLD);
}
else if (rank==1){
    MPI_Recv(hBuffer,size,MPI_BYTE,
    0,100,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    cudaMemcpy(dBuffer,hBuffer,size,cudaMemcpyHostToDevice);
}
```

GPUs across Network Nodes

- Requires network communication
  - Currently requires data to first be transferred to host
- Steps for an exchange:
  - GPU->CPU transfer
  - CPU exchanges via network
    - For example, MPI_Sendrecv
    - Just like you would do for non-GPU code
  - CPU->GPU transfer
- If each node also has multiple GPUs:
  - Can continue using P2P within the node
  - Can overlap some PCIe transfers with network communication
    - In addition to kernel execution

GPUs across Network Nodes

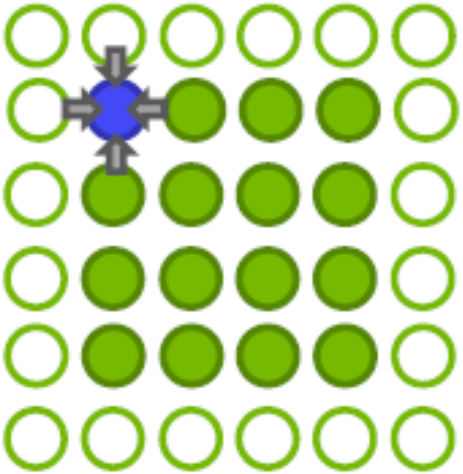
- Code Pattern

```
cudaMemcpyAsync( ..., stream[i] );
cudaStreamSynchronize( stream[i] );
MPI_Sendrecv( ... );
cudaMemcpyAsync( ..., stream[i] );
```

EXAMPLE: JACOBI SOLVER –SINGLE GPU

- While not converged
  - Do Jacobistep:

```
for (int i=1; i < n-1; i++)
    for (int j=1; j < m-1; j++)
        u_new[i][j] = 0.0f -0.25f*(u[i-1][j] +u[i+1][j]
                                   +u[i][j-1] + u[i][j+1])
```



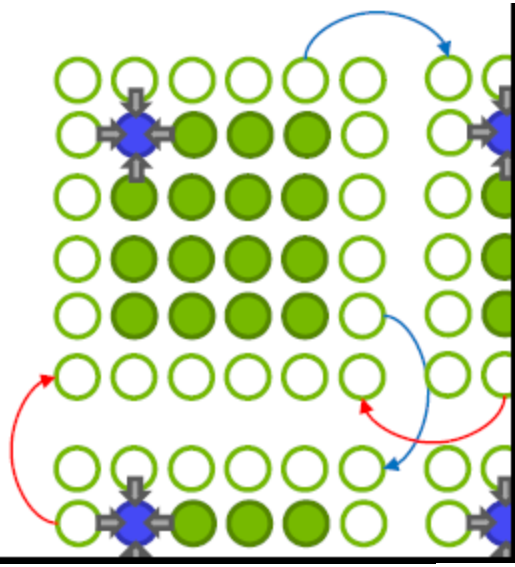
  - Swap u_new and u
  - Next iteration

EXAMPLE: JACOBI SOLVER – MULTI GPUs

- While not converged
  - Do Jacobi step:

```
for (int i=1; i < n-1; i++)
    for (int j=1; j < m-1; j++)
        u_new[i][j] = 0.0f -0.25f*(u[i-1][j] +u[i+1][j]
                                   +u[i][j-1] + u[i][j+1])
```

  - Exchange halo with 2 4 neighbor
  - Swap u_new and u
  - Next iteration



EXAMPLE: JACOBI SOLVER

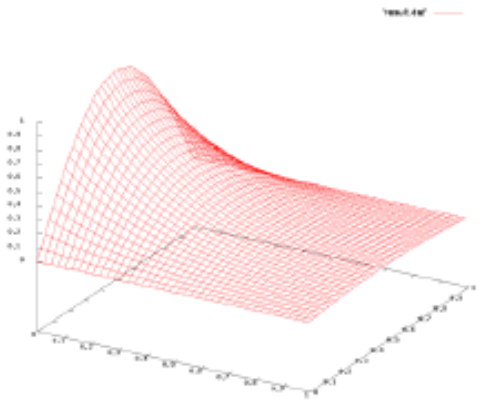
- Solves the 2D-Laplace equation on a rectangle

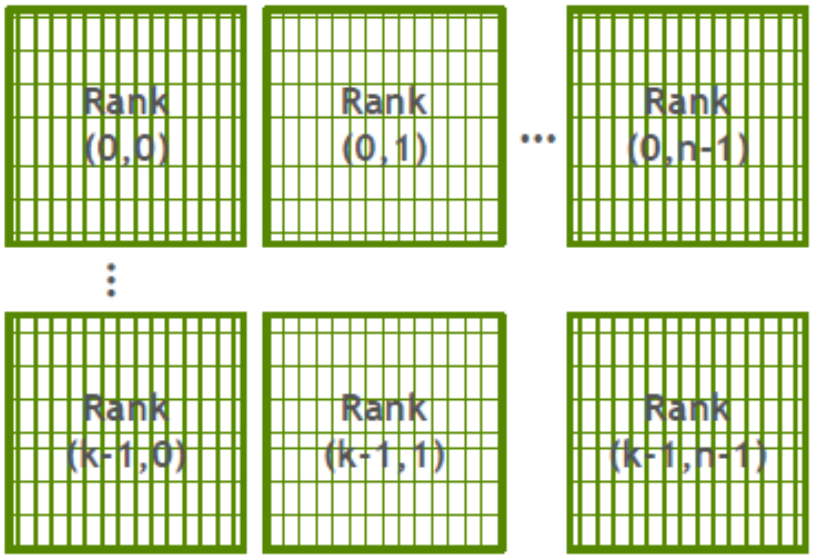
$$\Delta u(x, y) = 0 \; \forall \; (x, y) \in \Omega \backslash \delta\Omega$$

  - Dirichlet boundary conditions (constant values on boundaries)

$$u(x, y) = f(x, y) \in \delta\Omega$$

- 2D domain decomposition with n x k domains

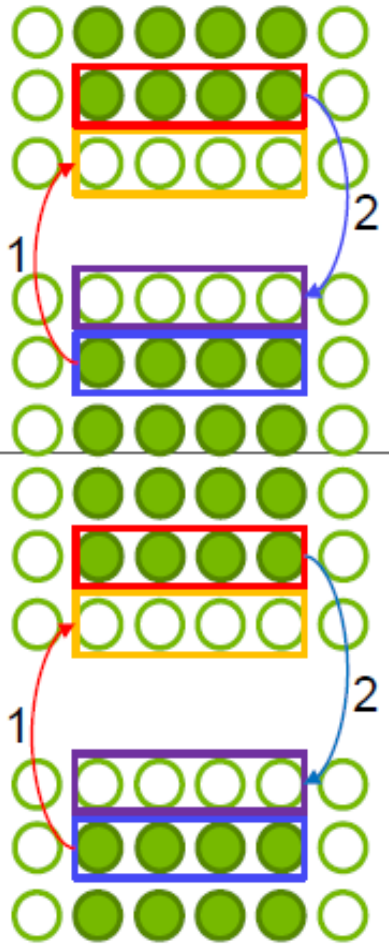




EXAMPLE: JACOBI –TOP/BOTTOM HALO UPDATE

```
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



EXAMPLE: JACOBI –TOP/BOTTOM HALO UPDATE

```
//right neighbor omitted
pack<<<gs,bs,0,s>>>(to_left_d, u_new_d, n, m);
cudaStreamSynchronize(s);




MPI_Sendrecv( to_left_d, n-2, MPI_DOUBLE, l_nb, 0,
              from_left_d, n-2, MPI_DOUBLE, l_nb, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );




unpack<<<gs,bs,0,s>>>(u_new_d, from_left_d, n, m);
```