MULTI-GPU PROGRAMMING MODELS

Jiri Kraus, Senior Devtech Compute
Sreeram Potluri, Senior CUDA Software Engineer
MOTIVATION
Why use multiple GPUs?

Need to compute larger, e.g. bigger networks, car models, ...

Need to compute faster, e.g. weather prediction

Better energy efficiency with dense nodes with multiple GPUs
Two fully connected quads, connected at corners

160GB/s per GPU bidirectional to Peers

Load/store access to Peer Memory

Full atomics to Peer GPUs

High speed copy engines for bulk data copy

PCIe to/from CPU
EXAMPLE: JACOBI SOLVER

Solves the 2D-Laplace Equation on a rectangle

\[ \Delta u(x, y) = 0 \ \forall \ (x, y) \in \Omega \setminus \delta \Omega \]

Dirichlet boundary conditions (constant values on boundaries) on left and right boundary

Periodic boundary conditions on top and bottom boundary
EXAMPLE: JACOBI SOLVER

Single GPU

While not converged

Do Jacobi step:

```c
for( int iy = 1; iy < ny-1; iy++ )
for( int ix = 1; ix < ny-1; ix++ )
a_new[iy*nx+ix] = -0.25 *
    -( a[ iy  *nx+(ix+1)] + a[ iy  *nx+ix-1]
        + a[(iy-1)*nx+ ix ] + a[(iy+1)*nx+ix ] );
```

Apply periodic boundary conditions

Swap `a_new` and `a`

Next iteration
DOMAIN DECOMPOSITION

Different Ways to split the work between processes:

Minimize number of neighbors:
Communicate to less neighbors
Optimal for latency bound communication

Minimize surface area/volume ratio:
Communicate less data
Optimal for bandwidth bound communication

Horizontal Stripes
Contiguous if data is row-major

Vertical Stripes
Contiguous if data is column-major

Tiles
While not converged

Do Jacobi step:

```c
for (int iy = iy_start; iy < iy_end; iy++)
    for (int ix = 1; ix < ny-1; ix++)
        a_new[iy*nx+ix] = -0.25 * 
            -( a[ iy  *nx+(ix+1)] + a[ iy  *nx+ix-1] 
               + a[(iy-1)*nx+ ix ] + a[(iy+1)*nx+ix ] );
```

Apply periodic boundary conditions

Exchange halo with 2 neighbors

Swap `a_new` and `a`

Next iteration

EXAMPLE: JACOBI SOLVER

Multi GPU
while ( l2_norm > tol && iter < iter_max ) {
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1);
        const int bottom = (dev_id+1)%num_devices;
        cudaSetDevice( dev_id );
        cudaMemcpyAsync( l2_norm_d[dev_id], 0 , sizeof(real) );
        jacobi_kernel<<<dim_grid,dim_block>>>( a_new[dev_id], a[dev_id], l2_norm_d[dev_id],
            iy_start[dev_id], iy_end[dev_id], nx );
        cudaMemcpyAsync( 12_norm_h[dev_id], l2_norm_d[dev_id], sizeof(real), cudaMemcpyDeviceToHost );
        cudaMemcpyAsync( a_new[top]+(iy_end[top]*nx), a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), cudaMemcpyHostToDevice );
        cudaMemcpyAsync( a_new[bottom], a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real), cudaMemcpyHostToDevice );
    }
    l2_norm = 0.0;
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        cudaSetDevice( dev_id );
        cudaMemcpyAsync( l2_norm_h[dev_id], 0 , sizeof(real) );
        cudaMemcpyAsync( l2_norm_h[dev_id], l2_norm_h[dev_id], sizeof(real), cudaMemcpyDeviceToHost );
        cudaMemcpyAsync( l2_norm_h[dev_id], 0 , sizeof(real) );
    }
    l2_norm = std::sqrt( 12_norm );
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) std::swap(a_new[dev_id],a[dev_id]);
    iter++;
}
cudaMemcpyAsync(
    a_new[top] + (iy_end[top] * nx),
    a_new[dev_id] + iy_start[dev_id] * nx, nx * sizeof(real), ...);
cudaMemcpyAsync(
  a_new[top]+(iy_end[top]*nx),
  a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
EXAMPLE JACOBI  
Top/Bottom Halo

cudaMemcpyAsync(  
    a_new[top] + (iy_end[top] * nx),  
    a_new[dev_id] + iy_start[dev_id] * nx, nx*sizeof(real), ...);

cudaMemcpyAsync(  
    a_new[bottom],  
    a_new[dev_id] + (iy_end[dev_id] - 1) * nx, nx*sizeof(real), ...);
SCALABILITY METRICS FOR SUCCESS

Serial Time: $T_s$: How long it takes to run the problem with a single process

Parallel Time: $T_p$: How long it takes to run the problem with multiple processes

Number of Processes: $P$: The number of Processes operating on the task at hand

Speedup: $S = \frac{T_s}{T_p}$: How much faster is the parallel version vs. serial. (optimal is $P$)

Efficiency: $E = \frac{S}{P}$: How efficient are the processors used (optimal is 1)
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

Chart Title

Runtime (s)

#GPUs

Single Threaded Copy
Parallel Efficiency
MULTI GPU JACOBI NVVP TIMELINE

Single Threaded Copy 4 P100 on DGX-1
MULTI GPU JACOBI NVVP TIMELINE

Single Threaded Copy 4 P100 on DGX-1
GPUDIRECT P2P

Maximizes intra node inter GPU Bandwidth

Avoids Host memory and system topology bottlenecks
for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
  cudaSetDevice( dev_id );
  const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1);
  int canAccessPeer = 0;
  cudaDeviceCanAccessPeer ( &canAccessPeer, dev_id, top );
  if ( canAccessPeer )
    cudaDeviceEnablePeerAccess ( top, 0 );
  const int bottom = (dev_id+1)%num_devices;
  if ( top != bottom ) {
    cudaDeviceCanAccessPeer ( &canAccessPeer, dev_id, bottom );
    if ( canAccessPeer )
      cudaDeviceEnablePeerAccess ( bottom, 0 );
  }
}
MULTI GPU JACOBI NVVP TIMELINE
Single Threaded Copy 4 P100 on DGX-1 with P2P
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

Parallel Efficiency

#GPUs

Single Threaded Copy

Single Threaded Copy P2P
1D RING EXCHANGE

Halo updates for 1D domain decomposition with periodic boundary conditions

Unidirectional rings are important building block for collective algorithms
MAPPING 1D RING EXCHANGE TO DGX-1
MAPPING 1D RING EXCHANGE TO DGX-1

```
export CUDA_VISIBLE_DEVICES="0,1,2,3,7,6,5,4"
```
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

Parallel Efficiency

Single Threaded Copy
Single Threaded Copy P2P (no opt)
Single Threaded Copy P2P
MULTI GPU JACOBI NVVP TIMELINE
Single Threaded Copy 4 P100 on DGX-1 with P2P
int num_devices = 0;
cudaGetDeviceCount(&num_devices);

#pragma omp parallel num_threads(num_devices)
{
    int dev_id = omp_get_thread_num();
    cudaSetDevice(dev_id);
}
MULTI GPU JACOBI NVVP TIMELINE

Multi Threaded Copy 4 P100 on DGX-1 with P2P
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

Parallel Efficiency

#GPUs

0,00%

100,00%

200,00%

300,00%

400,00%

500,00%

600,00%

700,00%

800,00%

900,00%

1000,00%

1100,00%

1200,00%

Single Threaded Copy P2P

Multi Threaded Copy (no thread pinning)
GPU/CPU AFFINITY

- GPU0
- GPU1
- GPU2
- GPU3
- GPU4
- GPU5
- GPU6
- GPU7

- CPU 0
  - 0 - 19

- CPU 1
  - 20 - 39

- thread 0
- thread 1
- thread 2
- thread 3
- thread 4
- thread 5
- thread 6
- thread 7
## GPU/CPU AFFINITY

**Querying system topology with** `nvidia-smi topo -m`

```bash
$ nvidia-smi topo -m
```

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>GPU0</th>
<th>GPU1</th>
<th>GPU2</th>
<th>GPU3</th>
<th>GPU4</th>
<th>GPU5</th>
<th>GPU6</th>
<th>GPU7</th>
<th>mlx5_0</th>
<th>mlx5_2</th>
<th>mlx5_1</th>
<th>mlx5_3</th>
<th>CPU Affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>0-19</td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>X</td>
<td>NV1</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 0</td>
<td>GPU0</td>
<td>GPU1</td>
<td>GPU2</td>
<td>GPU3</td>
<td>GPU4</td>
<td>GPU5</td>
<td>GPU6</td>
<td>GPU7</td>
<td>mlx5_0</td>
<td>mlx5_2</td>
<td>mlx5_1</td>
<td>mlx5_3</td>
<td>CPU Affinity</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>0-19</td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>X</td>
<td>NV1</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>NV1</td>
<td>X</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>NV1</td>
<td>PHB</td>
<td>SOC</td>
<td>PIX</td>
<td>0-19</td>
</tr>
<tr>
<td>CPU 0</td>
<td>GPU0</td>
<td>GPU1</td>
<td>GPU2</td>
<td>GPU3</td>
<td>GPU4</td>
<td>GPU5</td>
<td>GPU6</td>
<td>GPU7</td>
<td>mlx5_0</td>
<td>mlx5_2</td>
<td>mlx5_1</td>
<td>mlx5_3</td>
<td>CPU Affinity</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>0-19</td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>X</td>
<td>NV1</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>NV1</td>
<td>X</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>NV1</td>
<td>PHB</td>
<td>SOC</td>
<td>PIX</td>
<td>0-19</td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>20-39</td>
</tr>
<tr>
<td>CPU 0</td>
<td>GPU0</td>
<td>GPU1</td>
<td>GPU2</td>
<td>GPU3</td>
<td>GPU4</td>
<td>GPU5</td>
<td>GPU6</td>
<td>GPU7</td>
<td>mlx5_0</td>
<td>mlx5_2</td>
<td>mlx5_1</td>
<td>mlx5_3</td>
<td>CPU Affinity</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>20-39</td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>X</td>
<td>NV1</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>NV1</td>
<td>X</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>NV1</td>
<td>PHB</td>
<td>SOC</td>
<td>PIX</td>
<td>20-39</td>
</tr>
<tr>
<td></td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>PIX</td>
<td>20-39</td>
</tr>
<tr>
<td></td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>X</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>SOC</td>
<td>X</td>
<td>SOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>X</td>
<td>SOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>X</td>
<td>SOC</td>
<td></td>
</tr>
</tbody>
</table>

### Legend:
- `NV1`: Node 1
- `SOC`: System On Chip
- `PIX`: Peripheral Component Interconnect Express
- `PHB`: Peripheral Hub
出口 OMP_PROC_BIND=TRUE
出口 CUDA_VISIBLE_DEVICES="0,1,2,3,7,6,5,4"
出口 OMP_PLACES="\{0\},\{1\},\{2\},\{3\},\{20\},\{21\},\{22\},\{23\}"
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

Parallel Efficiency

#GPUs

0,00%

100,00%

120,00%

0,00%

20,00%

40,00%

60,00%

80,00%

100,00%

Single Threaded Copy P2P
Multi Threaded Copy (no thread pinning)
Multi Threaded Copy
MULTI GPU JACOBI NVVP TIMELINE

Multi Threaded Copy 4 P100 on DGX-1 with P2P
COMMUNICATION + COMPUTATION OVERLAP

No Overlap

Process Whole Domain

Overlap

Boundary and inner domain processing can overlap

Process inner domain

Process boundary domain

Dependency

COMM

Possible gain
COMMUNICATION + COMPUTATION OVERLAP

//Compute bulk
cudaStreamWaitEvent(compute_stream,push_top_done[[(iter%2)][dev_id],0]);
cudaStreamWaitEvent(compute_stream,push_bottom_done[[(iter%2)][dev_id],0]);
jacobi_kernel<<<dim_grid,dim_block,0,compute_stream>>>(a_new[dev_id],a,l2_norm_d,(iy_start+1),(iy_end[dev_id]-1),nx);

//Compute boundaries
cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0 );
cudaStreamWaitEvent(push_top_stream, push_bottom_done[[(iter%2)][top], 0 ]);  
jacobi_kernel<<<nx/128+1,128,0,push_top_stream>>>( a_new[dev_id],a,l2_norm_d,iy_start,(iy_start+1),nx);  
cudaStreamWaitEvent(push_bottom_stream,reset_l2norm_done,0);
cudaStreamWaitEvent(push_bottom_stream,push_top_done[[(iter%2)][bottom], 0 ]);  
jacobi_kernel<<<nx/128+1,128,0,push_bottom_stream>>>( a_new[dev_id],a,l2_norm_d,(iy_end[dev_id]-1),iy_end[dev_id],nx);

//Apply periodic boundary conditions and exchange halo
 cudaMemcpyAsync(a_new[top]+(iy_end[top]*nx),a_new[dev_id]+iy_start*nx,nx*sizeof(real),cudaMemcpyDeviceToDevice,push_top_stream); 
cudaEventRecord(push_top_done[[(iter+1)%2]][dev_id],push_top_stream);  
cudaMemcpyAsync(a_new[bottom],a_new[dev_id]+(iy_end[dev_id]-1)*nx,nx*sizeof(real),cudaMemcpyDeviceToDevice,push_bottom_stream); 
cudaEventRecord(push_bottom_done[[(iter+1)%2]][dev_id],push_bottom_stream);
int leastPriority = 0;

int greatestPriority = leastPriority;

cudaDeviceGetStreamPriorityRange (&leastPriority, &greatestPriority);

cudaStreamCreateWithPriority (&compute_stream, cudaStreamDefault, leastPriority);

cudaStreamCreateWithPriority (&push_top_stream, cudaStreamDefault, greatestPriority);

cudaStreamCreateWithPriority (&push_bottom_stream, cudaStreamDefault, greatestPriority);
MULTI GPU JACOBI NVVP TIMELINE
Multi Threaded Copy Overlap 4 P100 on DGX-1 with P2P
MULTI GPU JACOBI RUNTIME
DGX1 - 1024 x 1024, 1000 iterations

Parallel Efficiency vs. #GPUs

- Single Threaded Copy P2P
- Multi Threaded Copy
- Multi Threaded Copy Overlap
MULTI GPU JACOBI NVVP TIMELINE
Multi Threaded Copy Overlap 4 P100 on DGX-1 with P2P
while ( l2_norm > tol && iter < iter_max ) {
    cudaMemcpyAsync(l2_norm, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream);
    std::swap(a_new[dev_id], a);
    iter++;
}
__global__ void jacobi_kernel(...) {
    for (int iy = bIdx.y*bDim.y+tIdx.y + iy_start; iy < iy_end; iy += bDim.y* gDim.y) {
        for (int ix = bIdx.x*bDim.x+tIdx.x + 1; ix < (nx-1); ix += bDim.x*gDim.x) {
            const real new_val = 0.25 * ( a[ iy * nx + ix + 1 ] + a[ iy * nx + ix - 1 ]
                                         + a[ (iy+1) * nx + ix ] + a[ (iy-1) * nx + ix ] );
            a_new[ iy * nx + ix ] = new_val;
            if ( iy_start == iy ) { a_new_top[ top_iy *nx + ix ] = new_val; }
            if ( (iy_end - 1) == iy ) { a_new_bottom[ bottom_iy*nx + ix ] = new_val; }
            real residue = new_val - a[ iy * nx + ix ];
            atomicAdd( l2_norm, residue*residue );
        }
    }
}

Using OpenMP and P2P Mappings
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations
MULTI GPU JACOBI NVVP TIMELINE

Multi Threaded P2P 4 P100 on DGX-1 with P2P
MULTI THREADED MULTI GPU PROGRAMMING

L2 norm reduction

cudaMemcpyAsync( l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream );
#pragma omp barrier
#pragma omp single
{ l2_norm = 0.0; }
#pragma omp barrier
cudaStreamSynchronize( compute_stream );
#pragma omp critical
{ l2_norm += *(l2_norm_h); }
#pragma omp barrier
#pragma omp single
{ l2_norm = std::sqrt(l2_norm); }
#pragma omp barrier

Can be hidden if L2 norm check is delayed.
MULTI GPU JACOBI NVVP TIMELINE

Multi Threaded P2P 4 P100 on DGX-1 with P2P
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

Chart Title

Parallel Efficiency vs. #GPUs

- Single Threaded Copy P2P
- Multi Threaded Copy
- Multi Threaded Copy Overlap
- Multi Threaded P2P (norm opt)
MESSAGE PASSING INTERFACE - MPI

Standard to exchange data between processes via messages

Defines API to exchanges messages

Point to Point: e.g. MPI_Send, MPI_Recv

Collectives: e.g. MPI_Reduce

Multiple implementations (open source and commercial)

Bindings for C/C++, Fortran, Python, ...

E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, ...
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    /* Initialize the MPI library */
    MPI_Init(&argc, &argv);
    /* Determine the calling process rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
$ mpicc -o myapp myapp.c
$ mpirun -np 4 ./myapp <args>
EXAMPLE JACOBI

Top/Bottom Halo

MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
EXAMPLE JACOBI

Top/Bottom Halo

MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top, 0,
a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
EXAMPLE JACOBI
Top/Bottom Halo

MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top, 0,
a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(a_new+(iy_end-1)*nx, nx, MPI_FLOAT, bottom, 0,
a_new, nx, MPI_FLOAT, top, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
HANDLING MULTIPLE MULTI GPU NODES
HANDLING MULTIPLE MULTI GPU NODES

How to determine the local rank? - MPI-3

```c
MPI_Comm local_comm;
MPI_Info info;
MPI_Info_create(&info);
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank, info, &local_comm);
int local_rank = -1;
MPI_Comm_rank(local_comm,&local_rank);
MPI_Comm_free(&local_comm);
MPI_Info_free(&info);
```
HANDLING MULTIPLE MULTI GPU NODES
HANDLING MULTIPLE MULTI GPU NODES

GPU-affinity

Use local rank:

```c
int local_rank = -1;
MPI_Comm_rank(local_comm,&local_rank);
int num_devices = 0;
cudaGetDeviceCount(&num_devices);
cudaSetDevice(local_rank % num_devices);
```
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

![Chart showing parallel efficiency for Multi Threaded Copy Overlap and MPI with 1 to 8 GPUs.]
MULTI GPU JACOBI NVVP TIMELINE

MPI 4 P100 on DGX-1
COMMUNICATION + COMPUTATION OVERLAP

```c
launch_jacobi_kernel( a_new, a, l2_norm_d, iy_start, (iy_start+1), nx, push_top_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_end-1), iy_end, nx, push_bottom_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_start+1), (iy_end-1), nx, compute_stream );
const int top = rank > 0 ? rank - 1 : (size-1);
const int bottom = (rank+1)%size;
cudaStreamSynchronize( push_top_stream );
MPI_Sendrecv( a_new+iy_start*nx, nx, MPI_REAL_TYPE, top , 0,
               a_new+(iy_end*nx), nx, MPI_REAL_TYPE, bottom, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE );
cudaStreamSynchronize( push_bottom_stream );
MPI_Sendrecv( a_new+(iy_end-1)*nx, nx, MPI_REAL_TYPE, bottom, 0,
               a_new, nx, MPI_REAL_TYPE, top, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE );
```
MULTI GPU JACOBI NVVP TIMELINE

MPI Overlapping 4 P100 on DGX-1
MULTI GPU JACOBI RUNTIME

DGX1 - 1024 x 1024, 1000 iterations

Parallel Efficiency

Multi Threaded Copy Overlap  MPI  MPI Overlap
NOT COVERED IN THIS TALK I

MPI with GPUDirect Async support (under development)

Enables MPI communication to follow CUDA stream ordering

Avoids unwanted CPU/GPU synchronization

S7128 - HOW TO ENABLE NVIDIA CUDA STREAM SYNCHRONOUS COMMUNICATIONS USING GPUDIRECT - Monday, May 8, 2:00 PM - 2:50 AM (recorded)

Thursday, May 11, 2:00 PM - 2:50 PM: S7356 - MVAPICH2-GDR: PUSHING THE FRONTIER OF HPC AND DEEP LEARNING - Room 211B
NOT COVERED IN THIS TALK II
NCCL: Accelerating multi-GPU collective communications

GOAL:
- A library for collective communication using CUDA kernels for reduction and data movement.

APPROACH:
- Allreduce, Reduce, Broadcast, ReduceScatter and Allgather primitives, similar to MPI primitives.
- CUDA oriented: works on CUDA pointers only, enqueues operations to CUDA streams.
- Supports any mapping between processes, threads and GPUs per thread to integrate into any hybrid model.

S7155 - OPTIMIZED INTER-GPU COLLECTIVE OPERATIONS WITH NCCL - Tuesday, May 9, 9:00 AM - 9:50 AM (recorded)
CONCLUSION

<table>
<thead>
<tr>
<th>Programming Models</th>
<th>GPUDirect P2P</th>
<th>Multi Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Threaded</td>
<td>CUDA</td>
<td>Improves Perf.</td>
</tr>
<tr>
<td>Multi Threaded</td>
<td>CUDA + OpenMP/TBB/...</td>
<td>Improves Perf.</td>
</tr>
<tr>
<td>Multi Threaded P2P</td>
<td>CUDA + OpenMP/TBB/...</td>
<td>Required</td>
</tr>
<tr>
<td>MPI</td>
<td>CUDA + MPI</td>
<td>Improves Perf.</td>
</tr>
</tbody>
</table>

Wednesday, May 10, 2:00 PM - 3:00 PM: H7119 - CONNECT WITH THE EXPERTS: MULTI-GPU PROGRAMMING - LL Pod B

Thursday, May 11, 2:00 PM - 2:50 PM: S7356 - MVAPICH2-GDR: PUSHING THE FRONTIER OF HPC AND DEEP LEARNING - Room 211B