MULTI GPU PROGRAMMING WITH MPI

Jiri Kraus and Peter Messmer, NVIDIA
MPI+CUDA

Node 0

Node 1

Node n-1
MPI+CUDA
MPI+CUDA

//MPI rank 0
MPI_Send(s_buf_d, size, MPI_CHAR, n-1, tag, MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
WHAT YOU WILL LEARN

- What MPI is
- How to use MPI for inter GPU communication with CUDA and OpenACC
- What CUDA-aware MPI is
- What Multi Process Service is and how to use it
- How to use NVIDIA Tools in an MPI environment
- How to hide MPI communication times
MESSAGE PASSING INTERFACE - MPI

- Standard to exchange data between processes via messages
  - Defines API to exchanges messages
    - Pt. 2 Pt.: e.g. MPI_Send, MPI_Recv
    - Collectives, e.g. MPI_Reduce

- Multiple implementations (open source and commercial)
  - Binding for C/C++, Fortran, Python, ...
  - E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, ...
MPI - A MINIMAL PROGRAM

#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_Recev, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
MPI - COMPILING AND LAUNCHING

$ mpicc -o myapp myapp.c
$ mpirun -np 4 ./myapp <args>
A SIMPLE EXAMPLE
EXAMPLE: JACOBI SOLVER - SINGLE GPU

While not converged

- Do Jacobi step:
  
  for (int i=1; i < n-1; i++)
    for (int j=1; j < m-1; j++)
      \( u_{\text{new}}[i][j] = 0.0f - 0.25f*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]) \)

- Swap \( u_{\text{new}} \) and \( u \)

- Next iteration
EXAMPLE: JACOBI SOLVER - MULTI GPU

While not converged

- Do Jacobi step:
  
  ```c
  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
  \[ \Delta u(x, y) = 0 \ \forall \ (x, y) \in \Omega \setminus \delta \Omega \]
  - 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

```c
MPI_Sendrecv(u_new+\text{offset\_first\_row}, m-2, MPI\_DOUBLE, t\_nb, 0,
    u_new+\text{offset\_bottom\_boundary}, m-2, MPI\_DOUBLE, b\_nb, 0,
    MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI_Sendrecv(u_new+\text{offset\_last\_row}, m-2, MPI\_DOUBLE, b\_nb, 1,
    u_new+\text{offset\_top\_boundary}, m-2, MPI\_DOUBLE, t\_nb, 1,
    MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);
```
EXAMPLE: JACOBI - TOP/BOTTOM HALO UPDATE

```c
#pragma acc host_data use_device ( u_new ) {
    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_boundary, m-2, MPI_DOUBLE, t_nb, 1,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```c
MPI_Sendrecv (u_new_d+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
                u_new_d+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv (u_new_d+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
                u_new_d+offset_top_boundary, m-2, MPI_DOUBLE, t_nb, 1,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```
EXAMPLE: JACOBI - LEFT/RIGHT HALO UPDATE

```c
//right neighbor omitted
#pragma acc parallel loop present ( u_new, to_left )
for ( int i=0; i<n-2; ++i )
   to_left[i] = u_new[(i+1)*m+1];

#pragma acc host_data use_device ( from_left, to_left ) {
   MPI_Sendrecv( to_left, n-2, MPI_DOUBLE, l_nb, 0,
                  from_left, n-2, MPI_DOUBLE, l_nb, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}

#pragma acc parallel loop present ( u_new, from_left )
for ( int i=0; i<n-2; ++i )
   u_new[(i+1)*m] = from_left[i];
```
EXAMPLE: JACOBI – LEFT/RIGHT HALO UPDATE

```c
// 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);
```

CUDA
STARTING MPI+CUDA/OPENACC PROGRAMS

- **Launch one process per GPU**
  - **MVAPICH**: MV2_USE_CUDA
    
    $ MV2_USE_CUDA=1 mpirun -np ${np} ./myapp <args>
  - **Open MPI**: CUDA-aware features are enabled per default
  - **Cray**: MPICH_RDMA_ENABLED_CUDA
  - **IBM Platform MPI**: PMPI_GPU_AWARE
JACOBI RESULTS (1000 STEPS)
WEAK SCALING 4K X 4K PER PROCESS

MVAPICH2-2.0b FDR IB

Runtime (s)

#MPI Ranks - 1 CPU Socket with 10 OMP Threads or 1 GPU per Rank

- Tesla K20X
- Xeon E5-2690 v2 @ 3.0Ghz
**EXAMPLE: JACOBI – TOP/BOTTOM HALO UPDATE – WITHOUT CUDA-AWARE MPI**

```c
#pragma acc update host( u_new[1:m-2], u_new[(n-2)*m+1:m-2] )
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_boundary, 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_boundary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
#pragma acc update device( u_new[0:m-2], u_new[(n-2)*m:m-2] )
//send to bottom and receive from top – top bottom omitted

cudaMemcpy(u_new+1, u_new_d+1, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
cudaMemcpy(u_new_d, u_new, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
```
THE DETAILS
UNIFIED VIRTUAL ADDRESSING

No UVA : Separate Address Spaces  UVA : Single Address Space

System Memory  GPU Memory  System Memory  GPU Memory

0x0000  0xFFFF  0x0000  0xFFFF

CPU  GPU  CPU  GPU

PCI-e  PCI-e
UNIFIED VIRTUAL ADDRESSING

- One address space for all CPU and GPU memory
  - Determine physical memory location from a pointer value
  - Enable libraries to simplify their interfaces (e.g. MPI and cudaMemcpy)

- Supported on devices with compute capability 2.0 for
  - 64-bit applications on Linux and on Windows also TCC mode
MPI+CUDA

With UVA and CUDA-aware MPI

```c
//MPI rank 0
MPI_Send(s_buf_d, size, ...);

//MPI rank n-1
MPI_Recv(r_buf_d, size, ...);
```

```c
//MPI rank 0
cudaMemcpy(s_buf_h, s_buf_d, size, ...);
MPI_Send(s_buf_h, size, ...);

//MPI rank n-1
MPI_Recv(r_buf_h, size, ...);
cudaMemcpy(r_buf_d, r_buf_h, size, ...);
```

No UVA and regular MPI
NVIDIA GPUDIRECT™
ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES
NVIDIA GPUDIRECT™
ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES
NVIDIA GPUDIRECT™
PEER TO PEER TRANSFERS
NVIDIA GPDIRECT™
PEER TO PEER TRANSFERS
NVIDIA GPDIRECT™
SUPPORT FOR RDMA
NVIDIA GPUDIRECT™
SUPPORT FOR RDMA
CUDA-AWARE MPI

Example:

MPI Rank 0 MPI_Send from GPU Buffer
MPI Rank 1 MPI_Recv to GPU Buffer

- Show how CUDA+MPI works in principle
  - Depending on the MPI implementation, message size, system setup, ... situation might be different

- Two GPUs in two nodes
CUDA-AWARE MPI

GPU Buffer
PCI-E DMA

Host Buffer
memcpy
Pinned CUDA Buffer

Pinned fabric Buffer
RDMA
MPI_GPU_TOREMOTE_GPU
GPUDIRECT_SUPPORT_FOR_RDMA

MPI_Send(s_buf_d, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

MPI_Recv(r_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
MPI GPU TO REMOTE GPU
GPUDIRECT SUPPORT FOR RDMA
cudaMemcpy(s_buf_h, s_buf_d, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

MPI_Recv(r_buf_h, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
cudaMemcpy(r_buf_d, r_buf_h, size, cudaMemcpyHostToDevice);

cudaMemcpy(s_buf_h, s_buf_d, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
REGULAR MPI GPU TO REMOTE GPU

memcpy D->H  |  MPI_Sendrecv  |  memcpy H->D

Time
MPI GPU TO REMOTE GPU
WITHOUT GPUDIRECT

MPI_Send(s_buf_h, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

MPI_Recv(r_buf_h, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
MPI_GPU_TO_REMOTE_GPU
WITHOUT_GPUDIRECT

MPI_Sendrecv

More Details on pipelines:
CUDA_Streams: Best Practices and Common Pitfalls (Tuesday 03/27 210A)
PERFORMANCE RESULTS TWO NODES

OpenMPI 1.7.4 MLNX FDR IB (4X) Tesla K40

Latency (1 byte)  19.04 us  16.91 us  5.52 us
MULTI PROCESS SERVICE (MPS) FOR MPI APPLICATIONS
GPU ACCELERATION OF LEGACY MPI APPLICATION

- Typical legacy application
  - MPI parallel
  - Single or few threads per MPI rank (e.g. OpenMP)

- Running with multiple MPI ranks per node

- GPU acceleration in phases
  - Proof of concept prototype, ..
  - Great speedup at kernel level

- Application performance misses expectations
GPU parallelizable part
CPU parallel part
Serial part

N=1

Multicore CPU only
GPU parallelizable part
CPU parallel part
Serial part

Multicore CPU only

N=1
N=2
GPU parallelizable part
CPU parallel part
Serial part

N=1
N=2
N=4

Multicore CPU only
GPU parallelizable part
CPU parallel part
Serial part

Multicore CPU only

N=1  N=2  N=4  N=8
GPU parallelizable part
CPU parallel part
Serial part

N=4
N=2
N=1
N=8

Multicore CPU only
GPU accelerated CPU

With Hyper-Q/MPS
Available in K20, K40
PROCESSES SHARING GPU WITHOUT MPS: NO OVERLAP
## GPU SHARING WITHOUT MPS

### Time-Sliced Use of GPU

<table>
<thead>
<tr>
<th>Process &quot;a.out&quot; (4117)</th>
<th>Thread 203947168</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtime API</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Profiling Overhead</td>
</tr>
<tr>
<td>Process &quot;a.out&quot; (4125)</td>
<td>Thread 1718705384</td>
</tr>
<tr>
<td></td>
<td>Runtime API</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Profiling Overhead</td>
</tr>
<tr>
<td>[0] Tesla K20Xn</td>
<td></td>
</tr>
<tr>
<td>Context 1-4117 (CUDA)</td>
<td>MmCpy (HtoD)</td>
</tr>
<tr>
<td></td>
<td>Compute</td>
</tr>
<tr>
<td></td>
<td>50.1% test_kernel(c)</td>
</tr>
<tr>
<td></td>
<td>Streams</td>
</tr>
<tr>
<td></td>
<td>Default</td>
</tr>
<tr>
<td>Context 1-4123 (CUDA)</td>
<td>MmCpy (HtoD)</td>
</tr>
<tr>
<td></td>
<td>Compute</td>
</tr>
<tr>
<td></td>
<td>49.9% test_kernel(c)</td>
</tr>
<tr>
<td></td>
<td>Streams</td>
</tr>
<tr>
<td></td>
<td>Default</td>
</tr>
</tbody>
</table>

### Context Switch

![Context Switch Diagram](image)
PROCESSES SHARING GPU WITH MPS: MAXIMUM OVERLAP

Process A
Context A

Process B
Context B

MPS Process

GPU

Kernels from Process A
Kernels from Process B
GPU SHARING WITH MPS

<table>
<thead>
<tr>
<th>Process &quot;a.out&quot; (3916)</th>
<th>Thread 2387953440</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtime API</td>
</tr>
<tr>
<td></td>
<td>Driver API</td>
</tr>
<tr>
<td></td>
<td>Profiling Overhead</td>
</tr>
<tr>
<td>Process &quot;a.out&quot; (3923)</td>
<td>Thread 3202574112</td>
</tr>
<tr>
<td></td>
<td>Runtime API</td>
</tr>
<tr>
<td></td>
<td>Driver API</td>
</tr>
<tr>
<td></td>
<td>Profiling Overhead</td>
</tr>
<tr>
<td>[0] Tesla K20Xm</td>
<td></td>
</tr>
<tr>
<td>Contact MPS (CUDA)</td>
<td>MemCpy (HtoD)</td>
</tr>
<tr>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td></td>
</tr>
<tr>
<td>Default-3916</td>
<td></td>
</tr>
<tr>
<td>Default-3923</td>
<td></td>
</tr>
</tbody>
</table>

![Image of NVIDIA Visual Profiler interface showing GPU performance metrics and kernel execution times](image.png)
CASE STUDY: HYPER-Q/MPS FOR UMT

Sharing the GPU between multi MPI ranks increases GPU utilization

Enables overlap between copy and compute of different processes
HYPER-Q/MPS CASE STUDIES

- Speedup vs. 1 Rank/GPU
- CPU Scaling Speedup
- Overlap/MPS Speedup
USING MPS

- No application modifications necessary
- Not limited to MPI applications
- MPS control daemon
  - Spawn MPS server upon CUDA application startup

- Typical setup
  
  ```
  export CUDA_VISIBLE_DEVICES=0
  nvidia-smi -i 0 -c EXCLUSIVE_PROCESS
  nvidia-cuda-mps-control -d
  ```

- On Cray XK/XC systems
  
  ```
  export CRAY_CUDA_MPS=1
  ```
USING MPS ON MULTI-GPU SYSTEMS

- MPS server only supports a single GPU
  - Use one MPS server per GPU
- Target specific GPU by setting CUDA_VISIBLE_DEVICES
- Adjust pipe/log directory

  export DEVICE=0
  export CUDA_VISIBLE_DEVICES=${DEVICE}
  export CUDA_MPS_PIPE_DIRECTORY=${HOME}/mps${DEVICE}/pipe
  export CUDA_MPS_LOG_DIRECTORY=${HOME}/mps${DEVICE}/log
  cuda_mps_server_control -d
  export DEVICE=1 ... 

MPS SUMMARY

- Easy path to get GPU acceleration for legacy applications
- Enables overlapping of memory copies and compute between different MPI ranks
DEBUGGING AND PROFILING
TOOLS FOR MPI+CUDA APPLICATIONS

- Memory Checking `cuda-memcheck`
- Debugging `cuda-gdb`
- Profiling `nvprof` and NVIDIA Visual Profiler
MEMORY CHECKING WITH CUDA-MEMCHECK

- Cuda-memcheck is a functional correctness checking suite similar to the valgrind memcheck tool
- Can be used in a MPI environment
  ```bash
  mpiexec -np 2 cuda-memcheck ./myapp <args>
  ```
- Problem: output of different processes is interleaved
  - Use save, log-file command line options and launcher script
    ```bash
   #!/bin/bash
    LOG=$1.$OMPI_COMM_WORLD_RANK
    #LOG=$1.$MV2_COMM_WORLD_RANK
    cuda-memcheck --log-file $LOG.log --save $LOG.memcheck $*
    
    mpiexec -np 2 cuda-memcheck-script.sh ./myapp <args>
    ```
MEMORY CHECKING WITH CUDA-MEMCHECK

[jkraus@sbo77 bin]$ MV2_USE_CUDA=1 mpiexec -np 4 ./cuda-memcheck-script.sh ./jacobi_cuda_aware_mpi_async -t 2 2 -d 1024 1024 -fs
Topology size: 2 x 2
Local domain size (current node): 1024 x 1024
Global domain size (all nodes): 2048 x 2048
Starting Jacobi run with 4 processes:
Error: CUDA result "unspecified launch failure" for call "cudaDeviceSynchronize()" in file "Host.c" at line 453. Terminating...

=================================================================
= BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES          
= EXIT CODE: 255                                                  
= CLEANING UP REMAINING PROCESSES                                
= YOU CAN IGNORE THE BELOW CLEANUP MESSAGES                      
=================================================================

[jkraus@sbo77 bin]$ ls *.memcheck
jacobi_cuda_aware_mpi_async.0.memcheck  jacobi_cuda_aware_mpi_async.2.memcheck
jacobi_cuda_aware_mpi_async.1.memcheck  jacobi_cuda_aware_mpi_async.3.memcheck
[jkraus@sbo77 bin]$ cuda-memcheck --read jacobi_cuda_aware_mpi_async.0.memcheck
MEMORY CHECKING WITH CUDA-MEMCHECK

Read output files with `cuda-memcheck --read`
DEBUGGING MPI+CUDA APPLICATIONS USING CUDA-GDB WITH MPI APPLICATIONS

- You can use cuda-gdb just like gdb with the same tricks
- For smaller applications, just launch xterms and cuda-gdb

```
> mpiexec -x -np 2 xterm -e cuda-gdb ./myapp <args>
```
DEBUGGING MPI+CUDA APPLICATIONS
CUDA-GDB ATTACH

- CUDA 5.0 and forward have the ability to attach to a running process

```c
if ( rank == 0 ) {
    int i=0;
    printf("rank %d: pid %d on %s ready for attach\n.", rank, getpid(),name);
    while (0 == i) {
        sleep(5);
    }
}
```

> mpiexec -np 2 ./jacobi_mpi+cuda

Jacobi relaxation Calculation: 4096 x 4096 mesh with 2 processes and one Tesla M2070 for each process (2049 rows per process).

rank 0: pid 30034 on judge107 ready for attach

> ssh judge107

jkraus@judge107:~> cuda-gdb --pid 30034
DEBUGGING MPI+CUDA APPLICATIONS

CUDA_DEVICE_WAITS_ON_EXCEPTION
DEBUGGING MPI+CUDA APPLICATIONS
THIRD PARTY TOOLS
- Allinea DDT debugger
- Totalview
- S4284 - Debugging PGI CUDA Fortran and OpenACC on GPUs with Allinea DDT - Tuesday 4pm LL20D
DDT: THREAD LEVEL DEBUGGING

Focus

Breakpoint inside kernel

Launch configuration

Per thread variables
PROFILING MPI+CUDA APPLICATIONS USING NVPROF+NVVP

3 Usage modes:

- Embed pid in output filename
  mpirun -np 2 nvprof --output-profile profile.out.%p

- Only save the textual output
  mpirun -np 2 nvprof --log-file profile.out.%p

- Collect profile data on all processes that run on a node
  nvprof --profile-all-processes -o profile.out.%p
PROFILING MPI+CUDA APPLICATIONS USING NVPROF+NVVP
PROFILING MPI+CUDA APPLICATIONS USING NVPROF+NVVP
PROFILING MPI+CUDA APPLICATIONS USING NVPROF+NVVP
Multiple parallel profiling tools are CUDA aware
- Score-P
- Vampir
- Tau

These tools are good for discovering MPI issues as well as basic CUDA performance inhibitors
ADVANCED MPI ON GPUS
BEST PRACTICE: USE NONE-BLOCKING MPI

```c
#pragma acc host_data use_device ( u_new ) {
    MPI_Irecv( 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_Irecv( 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);
}

MPI_Request t_b_req[4];
#pragma acc host_data use_device ( u_new ) {
    MPI_Isend( u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 0,
               u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Isend( u_new+offset_first_row, m-2, MPI_DOUBLE, b_nb, 0,
               u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
               MPI_COMM_WORLD, MPI_STATUSIGNORE);
}

MPI_Waitall(4, t_b_req, MPI_STATUSES_IGNORE);
```

Gives MPI more opportunities to build efficient pipelines
OVELAPPING COMMUNICATION AND COMPUTATION

MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

Runtime (seconds)

Local problem size

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Nooverlap</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096x4096</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>2048x2048</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>1024x1024</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>
OVERLAPPING COMMUNICATION AND COMPUTATION

No Overlap

- Process whole domain

Overlap

- Boundary and inner domain processing can overlap
- Process inner domain
- Process boundary domain

Dependency

Possible Speedup

MPI
OVERLAPPING COMMUNICATION AND COMPUTATION

```c
#pragma acc parallel loop present ( u_new, u, to_left, to_right ) async(1)
for ( ... )
    //Process boundary and pack to_left and to_right
#pragma acc parallel loop present ( u_new, u ) async(2)
for ( ... )
    //Process inner domain
#pragma acc wait(1)  //wait for boundary
MPI_Request req[8];
#pragma acc host_data use_device ( from_left, to_left, form_right, to_right, u_new ) {
    //Exchange halo with left, right, top and bottom neighbor
}
MPI_Waitall(8, req, MPI_STATUSES_IGNORE);
#pragma acc parallel loop present ( u_new, from_left, from_right )
for ( ... )
    //unpack from_left and from_right
#pragma acc wait  //wait for iteration to finish
```
OVERLAPPING COMMUNICATION AND COMPUTATION

process_boundary_and_pack<<<gs_b, bs_b, 0, s1>>>(u_new_d, u_d, to_left_d, to_right_d, n, m);

process_inner_domain<<<gs_id, bs_id, 0, s2>>>(u_new_d, u_d, to_left_d, to_right_d, n, m);

cudaStreamSynchronize(s1); //wait for boundary

MPI_Request req[8];

//Exchange halo with left, right, top and bottom neighbor

MPI_Waitall(8, req, MPI_STATUSES_IGNORE);

unpack<<<gs_s, bs_s>>>(u_new_d, from_left_d, from_right_d, n, m);

cudaDeviceSynchronize(); //wait for iteration to finish
OVERLAPPING COMMUNICATION AND COMPUTATION

MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

Speedup (Overlap vs. Nooverlap)

Runtime (seconds)

Local problem size

4096x4096 2048x2048 1024x1024

MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

Nooverlap

Overlap

Speedup

4096x4096 2048x2048 1024x1024

0 0.5 1 1.5 2 2.5 3 3.5

0 0.2 0.4 0.6 0.8 1 1.2 1.4

GPU TECHNOLOGY CONFERENCE
## MPI AND UNIFIED MEMORY

- Unified Memory support for CUDA-aware MPI needs changes to the MPI implementations
  - Check with your MPI implementation of choice for their plans
  - It might work in some situations but it is not supported

- Unified Memory and regular MPI
  - Require unmanaged staging buffers
    - Regular MPI has no knowledge of managed memory
    - CUDA 6 managed memory does not play well with RDMA protocols
HANDLING MULTI GPU NODES

- Multi GPU nodes and GPU-affinity:
  - Use local rank:
    ```
    int local_rank = //determine local rank
    int num_devices = 0;
    cudaGetDeviceCount(&num_devices);
    cudaSetDevice(local_rank % num_devices);
    ```
  - Use exclusive process mode + `cudaSetDevice(0)`
HANDLING MULTI GPU NODES

- How to determine local rank:
  - Rely on process placement (with one rank per GPU)
    ```
    int rank = 0;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    int num_devices = 0;
    cudaGetDeviceCount(&num_devices); // num_devices == ranks per node
    int local_rank = rank % num_devices;
    ```
  - Use environment variables provided by MPI launcher
    - e.g. for OpenMPI
      ```
      int local_rank = atoi(getenv("OMPI_COMM_WORLD_LOCAL_RANK"));
      ```
    - e.g. For MVPAICH2
      ```
      int local_rank = atoi(getenv("MV2_COMM_WORLD_LOCAL_RANK"));
      ```
CASE STUDY: B-CALM

- CUDA-aware MPI enabled a easy transition from single node multi GPU to multi node multi GPU
- Multi node multi GPU version allows to solve problems that could not be tackled before
CASE STUDY: LBM D2Q37

- CUDA-aware MPI improves strong scalability and simplifies programming

More info at: S4186 - Optimizing a LBM code for Compute Clusters with Kepler GPUs (Wed. 03/26)
CONCLUSIONS

- Using MPI as abstraction layer for Multi GPU programming allows multi GPU programs to scale beyond a single node
  - CUDA-aware MPI delivers ease of use, reduced network latency and increased bandwidth
- All NVIDIA tools are usable and third party tools are available
- Multiple CUDA-aware MPI implementations available
  - OpenMPI, MVAPICH2, Cray, IBM Platform MPI
- Other interesting sessions:
  - S4517 - Latest Advances in MVAPICH2 MPI Library for NVIDIA GPU Clusters with InfiniBand - Tuesday 3pm LL21A
  - S4589 - OpenMPI with RDMA Support and CUDA - Thursday 2pm 211B
OVERLAPPING COMMUNICATION AND COMPUTATION - TIPS AND TRICKS

- CUDA-aware MPI might use the default stream
  - Allocate stream with the non-blocking flag (cudaStreamNonBlocking)
  - More info: S4158 - CUDA Streams: Best Practices and Common Pitfalls Tuesday 03/27)

- In case of multiple kernels for boundary handling the kernel processing the inner domain might sneak in
  - Use single stream or events for inter stream dependencies via cudaStreamWaitEvent (#pragma acc wait async) - disables overlapping of boundary and inner domain kernels
  - Use high priority streams for boundary handling kernels - allows overlapping of boundary and inner domain kernels

- As of CUDA 6.0 GPUDirect P2P in multi process can overlap disable it for older releases
HIGH PRIORITY STREAMS

- Improve scalability with high priority streams
  (cudaStreamCreateWithPriority)
  - S4158 - CUDA Streams: Best Practices and Common Pitfalls (Thu. 03/27)

- Use-case MD Simulations:
  - S4465 - Optimizing CoMD: A Molecular Dynamics Proxy Application Study (Wed. 03/26)