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, ...
#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>
A SIMPLE EXAMPLE
EXAMPLE: JACOBI SOLVER

- Solves the 2D-Laplace equation on a rectangle
  \[ \Delta u(x, y) = 0 \quad \forall \ (x, y) \in \Omega \setminus \delta \Omega \]
  - 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 SOLVER - SINGLE 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])
  ```

- Swap u_new and u

- Next iteration
EXAMPLE: JACOBI SOLVER - MULTI GPU

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
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);
EXAMPLE: JACOBI - TOP/BOTTOM HALO

```c
#include <mpi.h>

#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_bondary, m-2, MPI_DOUBLE, t_nb, 1,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

CUDA
//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

```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);
```
LAUNCH 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)

MVAPICH2-2.0b FDR IB - weak scaling 4k x 4k per process

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

```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_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);
#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_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
cudaMemcpy(u_new_d, u_new, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
```
UNIFIED VIRTUAL ADDRESSING

No UVA : Separate Address Spaces

UVA : Single Address Space
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
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, ...);
```

No UVA and regular MPI

```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, ...);
```
NVIDIA GPDIRECT™
ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES
NVIDIA GPUDIRECT™
ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES

GPU1
Memory

GPU2
Memory

System
Memory

CPU

Chip set

IB

PCI-e
NVIDIA GPUDIRECT™
PEER TO PEER TRANSFERS

GPU1 Memory

GPU2 Memory

System Memory

CPU

Chip set

IB
NVIDIA GPUDIRECT™
PEER TO PEER TRANSFERS

GPU1
Memory

GPU1

GPU2
Memory

System
Memory

CPU

Chip
set

IB

PCI-e
NVIDIA GPUDIRECT™
SUPPORT FOR RDMA
NVIDIA GPUDIRECT™
SUPPORT FOR RDMA

GPU1
Memory

GPU1

Memory

GPU2

System
Memory

GPU2

CPU

Chipset

PCI-e

IB
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

- Host Buffer
- Pinned CUDA Buffer
- Pinned fabric Buffer

memcpy

GPU Buffer
PCI-E DMA
RDMA
MPI_GPU_TO_REMOTE_GPU
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
SUPPORT FOR RDMA

MPI_Sendrecv

Time
REGULAR MPI GPU TO REMOTE GPU

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);
REGULAR MPI GPU TO REMOTE GPU

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

Time
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);
PERFORMANCE RESULTS TWO NODES

OpenMPI 1.8.4 MLNX FDR IB (4X) Tesla K40@875

Latency (1 byte)  
- GPU-aware MPI with GPUDirect RDMA: 19.79 us  
- GPU-aware MPI: 17.97 us  
- Regular MPI: 5.70 us
MULTI PROCESS SERVICE (MPS) FOR MPI APPLICATIONS
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
Multicore CPU only

GPU parallelizable part
CPU parallel part
Serial part

With Hyper-Q/MPS
Available in K20, K40, K80

GPU accelerated CPU
PROCESSES SHARING GPU WITHOUT MPS:
NO OVERLAP
GPU SHARING WITHOUT MPS

<table>
<thead>
<tr>
<th>Context</th>
<th>1-4117 (CUDA)</th>
<th>1-4123 (CUDA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
<tr>
<td>Compute</td>
<td>test_kernel(...)</td>
<td>test_kernel(...)</td>
</tr>
</tbody>
</table>

*Table showing GPU context switch and usage.*
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

![Screenshot of NVIDIA Visual Profiler](image.png)

<table>
<thead>
<tr>
<th>Process: 2387953440</th>
<th>Time: 3.55s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 2387953440</td>
<td>3.56s</td>
</tr>
<tr>
<td>Runtime API</td>
<td></td>
</tr>
<tr>
<td>Driver API</td>
<td></td>
</tr>
<tr>
<td>Profiling Overhead</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process: 3202574112</th>
<th>Time: 3.57s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 3202574112</td>
<td>3.58s</td>
</tr>
<tr>
<td>Runtime API</td>
<td></td>
</tr>
<tr>
<td>Driver API</td>
<td></td>
</tr>
<tr>
<td>Profiling Overhead</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default-3916</th>
<th>Time: 3.59s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default-3923</td>
<td>3.6s</td>
</tr>
</tbody>
</table>

**Notes:**
- "Profile_3916.out" file is being analyzed.
- The screenshot shows NVIDIA Visual Profiler's output for profiling.
- The timeline highlights various kernel execution times.
CASE STUDY: HYPER-Q/MPS FOR UMT

Enables overlap between copy and compute of different processes

Sharing the GPU between multi MPI ranks increases GPU utilization
HYPER-Q/MPS CASE STUDIES

CPU Scaling Speedup

- HACC
- MP2C
- VASP
- ENZO
- UMT

Speedup vs. 1 Rank/GPU

- CPU Scaling Speedup
Additional speedup with MPS
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 -c EXCLUSIVE_PROCESS
  nvidia-cuda-mps-control -d
  ```

- On Cray XK/XC systems
  
  ```
  export CRAY_CUDA_MPS=1
  ```

New with CUDA 7: MPS support on MULTI-GPU systems
MPS SUMMARY

- Easy path to get GPU acceleration for legacy applications
- Enables overlapping of memory copies and compute between different MPI ranks
- Remark: MPS adds some overhead so if your application does not benefit, you want to avoid starting MPS.
DEBUGGING AND PROFILING
TOOLS FOR MPI+CUDA APPLICATIONS

- Memory Checking `cuda-memcheck`
- Debugging `cuda-gdb`
- Profiling `nvprof` and NVIDIA Visual Profiler
Cuda-memcheck is a functional correctness checking suite similar to the valgrind memcheck tool.

It can be used in a MPI environment:

```
mpiexec -np 2 cuda-memcheck ./myapp <args>
```

Problem: output of different processes is interleaved

Use save or log-file command line options:

```
mpirun -np 2 cuda-memcheck
--log-file name.%q{OMPI_COMM_WORLD_RANK}.log
--save name.%q{OMPI_COMM_WORLD_RANK}.memcheck
./myapp <args>
```

- **OpenMPI**: `OMPI_COMM_WORLD_RANK`
- **MVAPICH2**: `MV2_COMM_WORLD_RANK`
MEMORY CHECKING WITH CUDA-MEMCHECK

```
jkraus@ivb114:~/workspace/qwiklabs/Multi-GPU-MPI/task3$ mpirun --pp 2 cuda-memcheck --log-file laplace2d.%g{OMPI_COMM_WORLD_RANK}.log --save laplace2d.%g{OMPI_COMM_WORLD_RANK}.memcheck ./laplace2d
Jacobi relaxation Calculation: 2048 x 2048 mesh
Calculate reference solution and time serial execution.
Call to cuMempyDtoHAsync returned error 719: Launch failed (often invalid pointer dereference)
call to cuMemcpyDtoHAsync returned error 719: Launch failed (often invalid pointer dereference)
Primary job terminated normally, but 1 process returned
  a non-zero exit code.. Per user-direction, the job has been aborted.

mpirun detected that one or more processes exited with non-zero status, thus causing
  the job to be terminated. The first process to do so was:

  Process name: [[42894,1],0]
  Exit code: 1

[jkraus@ivb114 task3]$ ls laplace2d.*.log laplace2d.*.memcheck
laplace2d.0.log  laplace2d.0.memcheck  laplace2d.1.log  laplace2d.1.memcheck
[jkraus@ivb114 task3]$
```
MEMORY CHECKING WITH CUDA-MEMCHECK

Read outputfiles with **cuda-memcheck --read**
DEBUGGING MPI+CUDA APPLICATIONS

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

```bash
> mpiexec -x -np 2 xterm -e cuda-gdb ./myapp <args>
```
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);
    }
}
```

```bash
> 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

The application encountered an exception: Device Illegal Address.

The exception was triggered in device 3.

Program received signal CUDA_EXCEPTION_10, Device Illegal Address.

[Switching focus to CUDA kernel 0, grid 0, block (6,36,0), thread (0,6,0), device 3, cm 0, warp 15, lane 0]

0x000000000018e1ce in JacobiComputeKernel<<<(64,64,1),(16,16,1)>> (size=..., startmod=..., endmod=..., oldBlock=0x2300200000, newBlock=0x2301340000, stride=1024) at Device:0150

150 AtomicMaxTrace> (devResidue, rabs(newVal - oldBlock[memIdx]));
(cuda-gdb) bc

0x000000000018e1ce in JacobiComputeKernel<<<(64,64,1),(16,16,1)>> (size=..., startmod=..., endmod=..., oldBlock=0x2300200000, newBlock=0x2301340000, devResidue=0x2301340000, stride=1024) at Device:0150

(cuda-gdb)
With CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1 core dumps are generated in case of an exception:
- Can be used for offline debugging
- Helpful if live debugging is not possible, e.g. too many nodes needed to reproduce

CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION
- Enable/Disable CPU part of core dump (enabled by default)

CUDA_COREDUMP_FILE
- Specify name of core dump file

Open GPU:  (cuda-gdb) target cudacore core.cuda
Open CPU+GPU: (cuda-gdb) target core core.cpu core.cuda
DEBUGGING MPI+CUDA APPLICATIONS
CUDA_ENABLE_COREDUMP_ON_EXCEPTION

```
[jkraus@ivb114 solutions]$ CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1 mpirun -x CUDA_ENABLE_COREDUMP_ON_EXCEPTION -np 2 ./jacobi_mpi+cuda
Jacobi relaxation Calculation: 4096 x 4096 mesh with 2 processes and one Tesla K40m for each process (2049 rows per process).
---------------------------------------------------------------------
mpirun noticed that process rank 1 with PID 28723 on node ivb114 exited on signal 11 (Segmentation fault).
---------------------------------------------------------------------
[jkraus@ivb114 solutions]$ ls core.*
core.cuda.ivb114.28722  core.cuda.ivb114.28723
[jkraus@ivb114 solutions]$`
```
DEBUGGING MPI+CUDA APPLICATIONS
CUDA_ENABLE_COREDUMP_ON_EXCEPTION
DEBUGGING MPI+CUDA APPLICATIONS

THIRD PARTY TOOLS

- Allinea DDT debugger
- Totalview
- S5417 - Three Ways to Debug Parallel CUDA Applications: Interactive, Batch, and Corefile - (Thursday 03/19, 17:00 - 17:25, Room 212A)
PROFILING MPI+CUDA APPLICATIONS
USING NVPROF+NVVP

3 Usage modes:

- Embed MPI rank in output filename
  `mpirun -np 2 nvprof --output-profile profile.out.%q{OMPI_COMM_WORLD_RANK}`

- Only save the textual output
  `mpirun -np 2 nvprof --log-file profile.out.%q{OMPI_COMM_WORLD_RANK}`

- Collect profile data on all processes that run on a node
  `nvprof --profile-all-processes -o profile.out.%h.%p`

  - **OpenMPI:** `OMPI_COMM_WORLD_RANK`
  - **MVAPICH2:** `MV2_COMM_WORLD_RANK`
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_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);
}

MPI_Request t_b_req[4];
#pragma acc host_data use_device ( u_new ) {
    MPI_Irecv(u_new+offset_top_boundary, m-2, MPI_DOUBLE, t_nb, 0,
               t_b_req);
    MPI_Irecv(u_new+offset_bottom_boundary, m-2, MPI_DOUBLE, b_nb, 0,
               t_b_req+1);
    MPI_Isend(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
               t_b_req+3);
    MPI_Isend(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 1,
               t_b_req+4);
}
MPI_Waitall(4, t_b_req, MPI_STATUSES_IGNORE);
```

Gives MPI more opportunities to build efficient pipelines.
COMMUNICATION + COMPUTATION OVERLAP

MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

- Runtime (seconds)
- Local problem size

- No overlap
- Ideal
COMMUNICATION + COMPUTATION OVERLAP

No Overlap
- Process whole domain
  - MPI

Overlap
- Process inner domain
  - Possible Speedup
- Process boundary domain
  - Dependency
  - MPI

Boundary and inner domain processing can overlap
#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

COMMUNICATION + COMPUTATION OVERLAP

OpenACC
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
**COMMUNICATION + COMPUTATION OVERLAP**

**Graph**: MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

- **Y-axis**: Runtime (seconds)
- **X-axis**: Local problem size

- **Bars**:
  - 4096x4096: Yellow (Nooverlap), Turquoise (Overlap)
  - 2048x2048: Yellow (Nooverlap), Turquoise (Overlap)
  - 1024x1024: Yellow (Nooverlap), Turquoise (Overlap)

- **Legend**:
  - Yellow: Nooverlap
  - Turquoise: Overlap
  - Black: Speedup (Overlap vs. Nooverlap)

**Speedup**:
- **4096x4096**: Speedup 3.5
- **2048x2048**: Speedup 1.5
- **1024x1024**: Speedup 1.0
HIGH PRIORITY STREAMS

- Improve scalability with high priority streams
  - cudaStreamCreateWithPriority
- Use-case: MD Simulations

Stream 1: Comp. Local Forces
Stream 1 (LP): Comp. Local Forces

Possible Speedup
MPI AND UNIFIED MEMORY

- Unified Memory support for CUDA-aware MPI needs explicit support from the MPI implementation:
  - Check with your MPI implementation of choice for their support
  - OpenMPI 1.8.5 supports unified memory

- Unified Memory and regular MPI
  - Require unmanaged staging buffer
    - Regular MPI has no knowledge of managed memory
    - CUDA 6 managed memory does not play well with RDMA protocols
CUDA-aware MPI might use the default stream
- Allocate stream with the non-blocking flag (cudaStreamNonBlocking)

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
HANDLING MULTI GPU NODES

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

How to determine local rank:

- Rely on process placement (with one rank per GPU)

```c
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
  ```c
  int local_rank = atoi(getenv("OMPI_COMM_WORLD_LOCAL_RANK"));
  ```
  
  - e.g. For MVPAICH2
  ```c
  int local_rank = atoi(getenv("MV2_COMM_WORLD_LOCAL_RANK"));
  ```
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
CONCLUSIONS

- S5461 - Latest Advances in MVAPICH2 MPI Library for NVIDIA GPU Clusters with InfiniBand - (Thursday 03/19, 17:00 - 17:50, Room 212B)
- S5146 - Data Movement Options for Scalable GPU Cluster Communication - (Thursday 03/19, 14:30 - 14:55, Room 210D)
- S5417 - Three Ways to Debug Parallel CUDA Applications: Interactive, Batch, and Corefile - (Thursday 03/19, 17:00 - 17:25, Room 212A)
- S5470 - Enabling Efficient Use of UPC and OpenSHMEM PGAS Models on GPU Clusters - (Thursday 03/19, 10:00 - 10:25, Room 212B)
- S5426 - Lesson Learned Using GPU Direct over RDMA on Production Heterogeneous Clusters - (Friday 03/20, 09:00 - 09:25, Room 212B)
- S5169 - Maximizing Scalability Performance in HOOMD-blue by Exploiting GPUDirect® RDMA on Green500 Supercomputer - (Friday 03/20, 10:30 - 10:55, Room 212B)