GPUs on Distributed Systems

CSE 5194.01
Spring ‘20

Quentin Anthony
The Ohio State University
E-mail: Anthony.301@osu.edu
Aside: CUDA

- CUDA is a parallel computing platform and programming model (NOT a language nor API)
- Programming in CUDA is typically done in Fortran, C, or C++
  - Support for other low-level languages is available (e.g. OpenCL, OpenACC)
  - PyCUDA wrapper allows CUDA functionality from Python code (written in C++, so very little performance degradation)
- CUDA programming typically follows the following general workflow
Aside: CUDA

- Example CUDA code: Saxpy
  - Single-precision A*X Plus Y
- Split into Host and Device code
- General idea:

```c
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    for (int i = 0; i < N; i++)
    {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
    saxpy<<<(N/256, 256)>>>(N, 2.0f, d_x, d_y);
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = max(maxError, abs(y[i]-4.0f));
    printf("Max error: %f\n", maxError);

    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
}``
Aside: CUDA

- Example CUDA code: Saxpy
  - Single-precision A*X Plus Y
- Split into Host and Device code
- General idea:
  - Allocate memory on both Host and Device

```c
#include <stdio.h>
__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
float *x, *y, *d_x, *d_y;
x = (float*)malloc(N*sizeof(float));
y = (float*)malloc(N*sizeof(float));
cudaMalloc(&d_x, N*sizeof(float));
cudaMalloc(&d_y, N*sizeof(float));

for (int i = 0; i < N; i++)
    x[i] = 1.0f;
y[i] = 2.0f;

cudamemcpy(d_x, x, N*sizeof(float), cudamemcpyHostToDevice);
cudamemcpy(d_y, y, N*sizeof(float), cudamemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<(N/256, 256)>>>(N, 2.0f, d_x, d_y);
cudamemcpy(y, d_y, N*sizeof(float), cudamemcpyDeviceToHost);

float maxError = 0.0f;
for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
printf("Max error: %f\n", maxError);
cudafree(d_x);
cudafree(d_y);
free(x);
free(y);
```
Aside: CUDA

- Example CUDA code: Saxpy
  - Single-precision A*X Plus Y
- Split into Host and Device code

General idea:
- Allocate memory on both Host and Device
- Initialize variables on Host

```c
#include <stdio.h>

__global__
void saxpy(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = alpha*x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    cudaMalloc((void**)&d_x, N*sizeof(float));
    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
    saxpy<<<(N/256), 256>>>(N, 2.0f, d_x, d_y);
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = max(maxError, abs(y[i]-4.0f));
    printf("Max error: %f\n", maxError);
    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
```
**Aside: CUDA**

- Example CUDA code: Saxpy
  - Single-precision A*X Plus Y
- Split into **Host** and **Device** code

**General idea:**
- Allocate memory on both **Host** and **Device**
- Initialize variables on **Host**
- Copy data from **Host** to **Device**

```c
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    for (int i = 0; i < N; i++)
        x[i] = 1.0f;
    y[i] = 2.0f;

    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
    saxpy<<<<<(N/255), 256>>>(N, 2.0f, d_x, d_y);
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = max(maxError, abs(y[i]-4.0f));
    printf("Max error: %f\n", maxError);
    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
}
Aside: CUDA

- Example CUDA code: Saxpy
  - Single-precision A*X Plus Y
- Split into Host and Device code

- General idea:
  - Allocate memory on both Host and Device
  - Initialize variables on Host
  - Copy data from Host to Device
  - Perform parallel computation on Device

```c
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    for (int i = 0; i < N; i++)
    {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1H elements
    saxpy<<<(N/256)/256, 256>>>(N, 2.0f, d_x, d_y);

    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = max(maxError, abs(y[i]-4.0f));
    printf("Max error: %f\n", maxError);
    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
}
Aside: CUDA

- Example CUDA code: Saxpy
  - Single-precision A\*X Plus Y
- Split into Host and Device code

General idea:
- Allocate memory on both Host and Device
- Initialize variables on Host
- Copy data from Host to Device
- Perform parallel computation on Device
- Copy Data back from Device to Host
Aside: CUDA

- Example CUDA code: Saxpy
  - Single-precision A*X Plus Y
- Split into Host and Device code

General idea:
- Allocate memory on both Host and Device
- Initialize variables on Host
- Copy data from Host to Device
- Perform parallel computation on Device
- Copy Data back from Device to Host
- Free memory on both Host and Device

```c
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));
    cudaMemcpy(d_x, N*sizeof(float));
    cudaMemcpy(d_y, N*sizeof(float));

    for (int i = 0; i < N; i++)
    {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(d_x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
    saxpy<<<(N/255)/256, 256>>>(N, 2.0f, d_x, d_y);
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
    {
        maxError = max(maxError, abs(y[i]-4.0f));
        printf("Max error: \%f\n", maxError);
    }

    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
}```
MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters

H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur and D. K. Panda
Motivation

- GPUs have become increasingly important for HPC applications
- MPI and CUDA were separate libraries
  - Buffers must be staged explicitly (higher latency)
  - No advanced MPI features (e.g. one-sided communication)
- Without MPI CUDA support, MPI collectives would need to be reimplemented for each application
  - MPI collectives are complex, tuned, and therefore internal to the MPI library
- Application-level MPI-CUDA support is prohibitive
  - Complex code
  - Time-consuming
  - Platform-specific (don’t tune the application, tune the collective)
1. Source process (Node 0) copies data from GPU memory to main memory
2. Source process sends data to the remote process (Node 1) over the IB network
3. Target process copies data from its main memory to its GPU memory
MVAPICH2-GPU

if(rank == 0) {
    cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
    MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
    ...
} else if(rank == 1) {
    MPI_Recv(r_buf, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &reqstat);
    cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
    ...
}

(a) Naïve MPI and CUDA without pipelining (good productivity, bad performance)

if(rank == 0) {
    MPI_Send(s_device, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
    ...
} else if(rank == 1) {
    MPI_Recv(r_device, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &reqstat);
    ...
}

(c) MVAPICH2-GPU (highest performance and productivity)

if(rank == 0) {
    for(j=0; j<pipeline_length; j++) {
        cudaMemcpyAsync(s_buf + j*block_size, s_device + j*block_size,
        block_size, cudaMemcpyDeviceToHost, cuda_stream[j]);
    }
    for(j=0; j<pipeline_length; j++) {
        while (result != cudaSuccess) {
            result = cudaStreamQuery(cuda_stream[j]);
            if(j>0) MPI_Test(&req[j-1], &flag, &status[j-1]);
        }
        MPI_Isend(s_buf + j*block_size, block_size, MPI_CHAR, 1, 1,
        MPI_COMM_WORLD, &req[j]);
    }
    MPI_WaitAll(pipeline_length, req, sstat);
    ...
} else if (rank == 1) {
    for(j=0; j<pipeline_length; j++) {
        MPI_Irecv(r_buf + j*block_size, block_size, MPI_CHAR, 0, 1,
        MPI_COMM_WORLD, &req[j]);
    }
    for(j=0; j<pipeline_length; j++) {
        MPI_Wait(&req[j], &rstat[j]);
        cudaMemcpyAsync(r_device + j*block_size, r_buf + j*block_size,
        block_size, cudaMemcpyHostToDevice, cuda_stream[j]);
    }
    for(j=0; j<pipeline_length; j++) {
        cudaMemcpyAsync(cuda_stream[j]);
    }
    ...
}

(b) Advanced MPI and CUDA with pipelining (low productivity, good performance)
MVAPICH2-GPU Pt-to-Pt Pipelining with GPU-Direct

1. Sender sends Request-To-Send (RTS) message, and receiver receives it
2. Receiver replies with Clear-To-Send (CTS) message
3. Data is internally buffered at sender/receiver host memory in vbufs (address in CTS message is a list of vbufs)
4. Sender receives CTS message, and starts a CUDA memory copy from device to host for each block (Asynchronously)
5. Sender calls IB verbs to perform RDMA write after each memory copy finishes
6. Sender sends RDMA write finish message after each RMDA write finishes
7. Receiver receives RDMA write finish and copies data from vbuf to GPU memory
MVAPICH2-GPU Pipelining without GPU-Direct

- Without GPU-Direct enabled, an additional `shadow_vbuf` performs an additional memory copy.

- `vbufs` are allocated and freed in `MPI_Init()` and `MPI_Finalize()`, respectively.

- Support is included for one-sided communication (`MPI_Put` and `MPI_Get`).

- Since collectives are implemented with Pt-to-Pt transfers, collectives also benefit.
MV2-GPU: Results

- Transfer latency of data (size $N$) split into pipeline blocks (size $n$): $T_{\text{cuda}} \left( \frac{N}{n} \right) + n \times T_{\text{mpi}} \left( \frac{N}{n} \right) + T_{\text{cuda}} \left( \frac{N}{n} \right)$
  - Where $T_{\text{cuda}} = \text{(time spent in CUDA memory copies)}$, $T_{\text{mpi}} = \text{(time spent in MPI library for network transfer)}$
- Best performance when overlapping latencies of network transfer and CUDA memory copies
- At 128 KB, network (RDMA write) latency is equal to CUDA memory copy latency (on this system)
  - MV2-GPU gets better performance above 128 KB because of overlap with network latency
MV2-GPU: Results

- Compare MV2-GPU to combination of CUDA and MPI (cudaMemcpy, MPI_Send, and MPI_Recv)
- MV2-GPU gets better performance than application-level optimization
  - MV2-GPU avoids rendezvous handshakes, message matching and message processing for each of the pipeline messages (better than best-achievable MPI-level optimization)
MV2-GPU: Results

- Again, MV2-GPU performs better than CUDA+MPI optimizations because it can overlap network transfers and memory copies.

![Graphs showing performance comparison between different methods](image-url)
MV2-GPU: Results

- Improvements over MPI+CUDA of up to 32% (Scatter), 37% (Gather), and 30% (Alltoall) with GPU Direct
- 23% (Scatter), 33% (Gather), 26% (Alltoall) without GPU-Direct

Fig. 7 Collectives latency performance
Efficient Inter-node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs

Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda Department of Computer Science and Engineering, The Ohio State University
Motivation

• CUDA was extended with GPUDirect RDMA (GDR), which allows network adapters to directly read/write to device memory (*bypassing the host*)

• NVIDIA partnered with Mellanox to enable GDR + IB

• GDR has low latency compared to Host, but bandwidth is limited when reading from GPU memory

• First paper to solve GPU-to-GPU MPI with GDR
GDR Design Background

- Existing GPU-Direct MV2 design with host-based pipeline has considerable overhead that can be avoided with RDMA support
- For small messages, we seek to avoid synchronization overhead
- There’s significant overhead for both semantics:
  - IB Send/Recv [send/receive processes post matching requests]
  - RDMA [buffer address and registration information is exchanged between processes]
- To avoid these overheads, previous MPI libraries use the eager protocol
  - Data is staged through pre-determined intermediate buffers at receiving process
  - Data is copied to destination buffer from intermediate buffers
GDR Design

- Most MPI libraries use RDMA for small messages, but GPU buffers cause issues:
  - CPU process polling on GPU memory is expensive
  - An extra copy from either a staging host or GPU buffer to the destination GPU buffer reduces GDR benefits
- Therefore, MV2-GDR uses the rendezvous protocol for all message transfers
GDR Design [Hybrid]

- While GDR with rendezvous protocol gives a low-latency path for small messages from GPU memory, some Intel architectures (Xeon E5-2600) provide limited bandwidth.
- Solution: hybrid between MV2-GPU (large messages) and MV2-GDR (small messages).
- Better solution: advanced hybrid design that overcomes Intel’s limited P2P read bandwidth by copying data onto sender’s host, then directly writing it to GPU at receiver.

Figure 5. Proposed GDR-H-Advanced based Inter-node GPU-to-GPU Pipelined Data Movement in MVAPICH2
Experiments [Pt-to-Pt Latency]

- All GDR designs are significantly improved for small messages
- GDR performs worse than MV2-GPU after 32KB due to P2P read bottleneck, so we need to use the Advanced Hybrid design (which switches to MV2-GPU with the added bypass of the Host on the receiver process)

Figure 6. Comparison of latency performance using the existing host-based pipelining and the proposed GDR-based designs on Sandy Bridge
Experiments [Pt-to-Pt Bandwidth]

- Again, all GDR designs are improved for small messages
- After 8KB, we again need the Advanced Hybrid design
- After 512KB, Advanced Hybrid peaks early due to P2P write limitation on Sandy Bridge system
- Similar insights with bi-bandwidth
Experiments [Pt-to-Pt WestmereEP]

- Only PCIe 2.0 on WestmereEP
- Despite higher latency, GDR still outperforms MV2-GPU
- Similar insights to those on Sandy Bridge system, but bandwidth doesn’t bottom off for Advanced Hybrid
Results [Collectives]

- Again, Pt-to-Pt improvements translate to collective improvements because collectives are implemented with Pt-to-Pt operations.
- Again, Advanced Hybrid design delivers best performance by switching between GDR and MV2-GPU designs while accounting for P2P bandwidth limitation on Sandy Bridge.

![Graphs showing latency under different message sizes for small, medium, and large messages.](image-url)
Results [Applications]

- GPULBM (CFD) improves slightly over different lattice sizes (communication will increase for more nodes)
- AWP-ODC (Seismology) improves greatly (again, more communication for more nodes)
Designing Efficient Small Message Transfer Mechanism for Inter-node MPI Communication on InfiniBand GPU Clusters

Rong Shi, Sreeram Potluri, Khaled Hamidouche, Jonathan Perkins, Mingzhe Li, Davide Rossetti, and Dhabaleswar K. (DK) Panda
Motivation

• GPUDirect is a set of CUDA features for efficient GPU communication
• GPUDirect RDMA (GDR) allows network adapters to directly read/write on GPU device memory (bypassing the host)
  – 3rd party PCIe devices may directly read/write data to/from GPU memory
• MV2-GDR already exists, but there’s room for improvement:
  – Reliance on rendezvous protocol
  – RTS/CTS header exchange overhead
  – Bandwidth limitations when the adapter and GPU are on different sockets
  – Existing eager protocol uses host-based copies
GDR Eager Design (Existing)

- Meta-data (header) is usually on **Host** buffers, message data is usually on **Device** buffers.

- Eager protocol is traditionally the most efficient for small message sizes:
  1. Sender copies from **Device** buffer to temporary **Host** buffer.
  2. Sender writes data to target buffer in receiver **Host** memory.
  3. Receiver copies data from **Host** temporary buffer to **Device** memory.

- The two *cudaMemcpy* calls (one at sender one at receiver) make this design inefficient.
GDR Rendezvous Design (Existing)

- When GDR is available for GPU-to-GPU communications, it’s better to use rendezvous
  1. Sender (Node A) sends RTS
  2. Receiver (Node B) sends CTS
  3. Sender writes data to receiver using RDMA
- Initial synchronization phase required (adds large overhead for small messages)
- Avoids memory copies via RDMA

Figure 1: Overview of GDR Rendezvous Protocol Design
GDR Eager Design (Proposed)

- **Sender Side Design**
  - Fastcopy (discussed later) [1A] and sglist (scatter-gather list) [1B]
  - Sglist overcomes the meta-data / data split between CPU and GPU with IB verbs
GDR Eager Design (Proposed) [NaiveCopy]

- Receiver Side Design
  1. Naïve Design [2A]
     - Uses `cudaMemcpy` to move data from host to device buffer
     - Receiver gets data from sender, checks if destination is on device (if so, use `cudaMemcpy` to copy data directly to device memory)
     - Figure 3 compares existing GDR rendezvous protocol and NaïveCopy (rendezvous is better because of `cudaMemcpy`’s expense)
GDR Eager Design (Proposed) [Loopback]

- Receiver Side Design
  1. Loopback Design [2B]
     - Avoid the expensive \textit{cudaMemcpy} using IB verbs to transfer from host to device memory using GDR
     - After tag matching using header data, the IB adapter reads data from temporary Host receive buffer to Device memory
     - Avoids calling \textit{cudaMemcpy}, but limited by PCIe limitations and consumes bandwidth (data travels three times across PCIe link)
GDR Eager Design (Proposed) [Fastcopy]

• Receiver Side Design
  1. Fastcopy Design [2C]
     • Low-latency host-device copy data-path by NVIDIA with three sets of primitives:
       - Pin/Unpin: used to setup/tear down HW mappings of device buffers
       - Map/Unmap: Which pages are memory-mapped into contiguous user-space CPU address range
       - Copy to/from PCIe BAR: tuned functions implemented in terms of Intel SSE instructions
     • Combination of these are used in 1A and 2C above
Performance [Pt-to-Pt]

- New OMB for GPUs was developed
- Compared rendezvous (MV2-GDR) with new proposed designs
- Fastcopy provides the best latency and bandwidth on both clusters and for each communication scheme
- Loopback is better for smaller message sizes, but rendezvous is comparable or better for larger messages
- Note that H-D always performs better than D-D, as expected
Performance [Message Rate and Collectives]

- Loopback and rendezvous (MV2-GDR) have comparable message rates, but Fastcopy has an average 1.6x improvement.
- Since collectives are implemented with pt-to-pt operations, we again see significant improvements for Bcast (67%), Gather (64%), and Allgather (43%).
Performance [End-Applications]

- **GPULBM**
  - Lattice Boltzmann Method (LBM) is a CFD method that can simulate the interaction/dynamics of fluids
  - Both Fastcopy and Loopback provide better performance for varying lattice sizes

- **HOOMD-blue**
  - Highly Optimized Object-oriented Many-particle Dynamics (HOOMD) is a general-purpose particle simulation toolkit for molecular dynamics
  - Both Fastcopy and Loopback provide better scaling
Performance Evaluation of MPI Libraries on GPU-enabled OpenPOWER Architectures: Early Experiences

Kawthar Shafie Khorassani, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K (DK) Panda
Motivation

- Even in 2019, GPU clusters are on the rise. New Hardware/Software/Interconnects have arisen in the TOP500
  - OpenPOWER CPUs (POWER8, POWER9)
  - CUDA10, CUDA-Aware MPI (SpectrumMPI, OpenMPI, MVAPICH2)
  - NVLINK, IB

- What MPI performance may be achieved on OpenPOWER architectures?
Motivation: System

While the four available interconnects improve performance, they pose the challenge of optimizing MPI libraries to take advantage of each data path.
Motivation: Theoretical Peak

- Theoretical peak bandwidths are between 85.43% and 94.56% due to the overhead of hardware, firmware, and software protocols (as well as other factors like cache effect)
- In addition to these peak bandwidths, it’s critical to understand the overhead of CUDA-aware MPI libraries
Aside: MPI Libraries

• **SpectrumMPI**
  – Provided by IBM (deployed by default on many OpenPOWER systems)
  – CUDA-Aware (GPUDirect RDMA and CUDA IPC)

• **OpenMPI**
  – Open-source
  – May be built with Unified Communication X (UCX) for CUDA-aware pt-to-pt communication

• **MVAPICH2**
  – Open-source
  – MVAPICH2-GDR is optimized for NVIDIA GPU clusters with GPUDirect RDMA and NVLink support
Results [Comm. Through NVLink Between GPUs]

- Most MPI+CUDA applications use a single MPI process and a single GPU
- Two GPUs on the same socket are compared over NVLink (on Sierra system)
- Bandwidth numbers are close to theoretical peak

**Fig. 2.** Latency comparison of MPI libraries on moving data between GPUs on the same socket (i.e., via NVLink interconnect) on a Sierra-like system

**Fig. 3.** Bandwidth comparison of MPI libraries on moving data between GPUs on the same socket (i.e., via NVLink interconnect) on a Sierra-like system
Results [Comm. Through NVLink Between GPUs]

- Most MPI+CUDA applications use a single MPI process and a single GPU
- Two GPUs on the same socket are compared over NVLink (on Summit system)
- Bandwidth numbers are close to theoretical peak, but farther than on Sierra
Results [Comm. Through GPU HBM2]

- To evaluate performance within GPU memory, two MPI processes are mapped to the same GPU (on Sierra system)
- OpenMPI bandwidth numbers aren’t comparable because it doesn’t use CUDA IPC for intra-node, intra-GPU communication
- Bandwidth numbers are close to theoretical peak
Results [Comm. Through GPU HBM2]

- To evaluate performance within GPU memory, two MPI processes are mapped to the same GPU (on Sierra system)
- OpenMPI bandwidth numbers aren’t comparable because it doesn’t use CUDA IPC for intra-node, intra-GPU communication
- Bandwidth numbers are close to theoretical peak

Fig. 6. Latency comparison of MPI libraries on moving data within one single GPU on a Sierra-like system

Fig. 7. Bandwidth comparison of MPI libraries on moving data within one single GPU on a Sierra-like system
Results [Comm. Through NVLink Between CPU and GPU]

- To evaluate hybrid (CPU-GPU) performance, two MPI processes (one on host buffer, one on GPU buffer) are tested (on Sierra system).
- SpectrumMPI latency numbers aren’t comparable.
- Bandwidth numbers are NOT close to theoretical peak (Current CUDA-aware MPI libraries have poor locality support).

Fig. 8. Latency comparison of MPI libraries on moving data from the Host to the Device on a Sierra-like system.

Fig. 9. Bandwidth comparison of MPI libraries on moving data from the Host to the Device on a Sierra-like system. The peak achievable uni-directional bandwidth in this case is approximately 68.78 GB/s.
Results [Comm. Through NVLink and X-Bus]

- Communication through X-bus between NUMA nodes [different sockets] (on Sierra system)
- Large-message latency is large compared to NVLink (indicating that X-bus becomes a performance bottleneck across sockets)
- Bandwidth numbers are close to theoretical peak

Fig. 10. Latency comparison of MPI libraries on moving data between GPUs on different sockets on a Sierra-like system

Fig. 11. Bandwidth comparison of MPI libraries on moving data between GPUs on different sockets on a Sierra-like system
Results [Comm. Through IB Network]

- Communication across two nodes (on Sierra system)
- Bandwidth numbers are close to theoretical peak [excluding OpenMPI degradation] (multi-rail support is being used)
Results [Comm. Through IB Network]

- Communication across two nodes (on Summit system)
- Bandwidth numbers are close to theoretical peak [excluding OpenMPI degradation] (multi-rail support is being used)
Conclusions [Experimental Peaks]

Table 2. Summary of achievable peak bandwidth of MPI libraries and fraction of peak over Interconnects on a Sierra-like GPU-enabled OpenPOWER System

<table>
<thead>
<tr>
<th>Library</th>
<th>GPU HBM2 (GB/s)</th>
<th>3-lane NVLink2 CPU-GPU</th>
<th>3-lane NVLink2 GPU-GPU</th>
<th>X-Bus (GB/s)</th>
<th>InfiniBand EDR×2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpectrumMPI</td>
<td>329 (36.55%)</td>
<td>21.74 (31.61%)</td>
<td>67.14 (95.20%)</td>
<td>39.16 (94.60%)</td>
<td>23.45 (99.20%)</td>
</tr>
<tr>
<td>OpenMPI+UCX</td>
<td>0.457 (0.05%)</td>
<td>23.63 (34.35%)</td>
<td>67.22 (95.40%)</td>
<td>31.77 (76.73%)</td>
<td>22.55 (95.40%)</td>
</tr>
<tr>
<td>MVAPICH2-GDR</td>
<td>390.88 (43.43%)</td>
<td>26.84 (39.02%)</td>
<td>67.15 (95.30%)</td>
<td>39.28 (94.97%)</td>
<td>23.56 (99.70%)</td>
</tr>
</tbody>
</table>

- >95% of peak bandwidth for inter-node IB communication with multi-rail support
- 31%-39% of peak bandwidth for NVLink CPU-GPU (Open issue for all MPI libraries)
- Both SpectrumMPI and MVAPICH2-GDR outperform OpenMPI+UCX for HBM2 and X-Bus
- Key takeaways:
  - Limited bandwidth of HBM2 when sharing memory
  - Host-to-Device and Device-To-Host communications aren’t using NVLink efficiently
  - MPI Libraries cannot fully utilize the bi-directional X-bus
Thank You!

Anthony.301@osu.edu