TAMING THE HYDRA MULTI-GPU PROGRAMMING WITH OPENACC

Jeff Larkin, GTC 2019, March 2019
Why use Multiple Devices?

Because 2 > 1: Effectively using multiple devices results in faster time to solution and/or solve more complex problems

Because they’re there: If running on a machine with multiple devices you’re wasting resources by not using them all

Because the code already runs on multiple nodes using MPI, so little change is necessary
SUMMIT NODE
(2) IBM POWER9 + (6) NVIDIA VOLTA V100

256 GB
(DDR4)

CPU 0

GPU 0

16 GB
(HBM2)

GPU 1

16 GB
(HBM2)

GPU 2

16 GB
(HBM2)

256 GB
(DDR4)

CPU 1

GPU 3

16 GB
(HBM2)

GPU 4

16 GB
(HBM2)

GPU 5

16 GB
(HBM2)

135 GB/s

135 GB/s

64 GB/s

NVLink2

(50 GB/s)

(900 GB/s)
DGX-1 NODE
MULTI-GPU PROGRAMMING STRATEGIES
MULTI-GPU PROGRAMMING MODELS

1 CPU : Many GPUs
• A single thread will change devices as-needed to send data and kernels to different GPUs

Many Threads : 1 GPU (Each)
• Using OpenMP, Pthreads, or similar, each thread can manage its own GPU

1+ CPU Process : 1 GPU
• Each rank acts as-if there’s just 1 GPU, but multiple ranks per node use all GPUs

Many Processes : Many GPUs
• Each rank manages multiple GPUs, multiple ranks/node. Gets complicated quickly!
## MULTI-GPU PROGRAMMING MODELS

**Trade-offs Between Approaches**

<table>
<thead>
<tr>
<th>1 CPU : Many GPUs</th>
<th>Many Threads : 1 GPU (Each)</th>
<th>1+ CPU : 1 GPU</th>
<th>Many Processes : Many GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Conceptually Simple</td>
<td>• Conceptually Very Simple</td>
<td>• Little to no code changes required</td>
<td>• Easily share data between peer devices</td>
</tr>
<tr>
<td>• Requires additional loops</td>
<td>• Set and forget the device numbers</td>
<td>• Re-uses existing domain decomposition</td>
<td>• Coordinating between GPUs extremely tricky</td>
</tr>
<tr>
<td>• CPU can become a bottleneck</td>
<td>• Relies on external Threading API</td>
<td>• Can see improved utilization</td>
<td>• CPU can become a bottleneck</td>
</tr>
<tr>
<td>• Remaining CPU cores often underutilized</td>
<td>• Watch affinity</td>
<td>• Watch affinity</td>
<td>• Remaining CPU cores often underutilized</td>
</tr>
</tbody>
</table>

- Little to no code changes required
- Re-uses existing domain decomposition
- Can see improved utilization
- Watch affinity
- Easily share data between peer devices
- Coordinating between GPUs extremely tricky
MULTI-DEVICE OPENACC

OpenACC presents devices numbered 0 - (N-1) for each device type available.

The order of the devices comes from the runtime, almost certainly the same as CUDA.

By default all data and work go to the current device. Each device has its own async queues.

Developers must change the current device and maybe the current device type using an API.
MULTI-DEVICE CUDA

CUDA by default exposes all devices, numbered 0 - (N-1), if devices are not all the same, it will reorder the “best” to device 0.

Each device has its own pool of streams.

If you do nothing, all work will go to Device #0.

Developer must change the current device explicitly using an API
SAMPLE CODE

Embarrassingly Parallel Image Filter
SAMPLE CODE

Embarrassingly Parallel Image Filter
SAMPLE CODE

Embarrassingly Parallel Image Filter
SAMPLE CODE

Embarrassingly Parallel Image Filter
for(y = 0; y < h; y++) {
    for(x = 0; x < w; x++) {
        double blue = 0.0, green = 0.0, red = 0.0;
        for(int fy = 0; fy < filtersize; fy++) {
            long iy = y - (filtersize/2) + fy;
            for (int fx = 0; fx < filtersize; fx++) {
                long ix = x - (filtersize/2) + fx;
                blue += filter[fy][fx] * (double)imgData[iy * step + ix * ch];
                green += filter[fy][fx] * (double)imgData[iy * step + ix * ch + 1];
                red += filter[fy][fx] * (double)imgData[iy * step + ix * ch + 2];
            }
        }
        out[y * step + x * ch] = 255 - (scale * blue);
        out[y * step + x * ch + 1] = 255 - (scale * green);
        out[y * step + x * ch + 2] = 255 - (scale * red);
    }
}
1: MANY
1: MANY - THE PLAN

1. Introduce loop to initialize each device with needed arrays
2. Send each device its block of the image
3. Introduce loop to copy data back from and tear down each device.
for(int device = 0; device < ndevices; device++) {
  acc_set_device_num(device, acc_device_default);
  long lower = device*rows_per_device;
  long upper = MIN(lower + rows_per_device, h);
  long copyLower = MAX(lower-(filtersize/2), 0);
  long copyUpper = MIN(upper+(filtersize/2), h);
  #pragma acc enter data create(imgData[copyLower*step:(copyUpper-copyLower)*step], \
  out[lower*step:(upper-lower)*step]) async
  #pragma acc parallel loop present(filter, \
  imgData[copyLower*step:(copyUpper-copyLower)*step], out[lower*step:(upper-lower)*step]) async
  for(y = lower; y < upper; y++) {
    for(x = 0; x < w; x++) {
      double blue = 0.0, green = 0.0, red = 0.0;
      for(int fy = 0; fy < filtersize; fy++) {
        long iy = y - (filtersize/2) + fy;
        for(int fx = 0; fx < filtersize; fx++) {
          long ix = x - (filtersize/2) + fx;
          if((iy<0) || (ix<0) || (iy>=h) || (ix>=w)) continue;
          blue += filter[fy][fx] * (double)imgData[iy*step + ix*ch];
          green += filter[fy][fx] * (double)imgData[iy*step + ix*ch + 1];
          red += filter[fy][fx] * (double)imgData[iy*step + ix*ch + 2];
        }
      }
      out[y*step + x*ch] = 255 - (scale * blue);
      out[y*step + x*ch + 1] = 255 - (scale * green);
      out[y*step + x*ch + 2] = 255 - (scale * red);
    }
  }
  #pragma acc update self out[lower*step:(upper-lower)*step]) async
}
for(int device = 0; device < ndevices; device++) {
  acc_set_device_num(device, acc_device_default);
  long lower = device*rows_per_device;
  long upper = MIN(lower + rows_per_device, h);
  long copyLower = MAX(lower-(filtersize/2), 0);
  long copyUpper = MIN(upper+(filtersize/2), h);
  #pragma acc enter data create(imgData[copyLower*step:(copyUpper-copyLower)*step], \
  out[lower*step:(upper-lower)*step]) async
  #pragma acc parallel loop present(filter, \
  imgData[copyLower*step:(copyUpper-copyLower)*step], out[lower*step:(upper-lower)*step]) async
  for(y = lower; y < upper; y++) {
    for(x = 0; x < w; x++) {
      double blue = 0.0, green = 0.0, red = 0.0;
      for(int fy = 0; fy < filtersize; fy++) {
        long iy = y - (filtersize/2) + fy;
        for(int fx = 0; fx < filtersize; fx++) {
          long ix = x - (filtersize/2) + fx;
          if((iy<0) || (ix<0) || (iy>=h) || (ix>=w)) continue;
          blue += filter[fy][fx] * (double)imgData[iy*step + ix*ch];
          green += filter[fy][fx] * (double)imgData[iy*step + ix*ch + 1];
          red += filter[fy][fx] * (double)imgData[iy*step + ix*ch + 2];
        }
      }
      out[y*step + x*ch] = 255 - (scale * blue);
      out[y*step + x*ch + 1] = 255 - (scale * green);
      out[y*step + x*ch + 2] = 255 - (scale * red);
    }
  }
  #pragma acc exit data delete(out[lower*step:(upper-lower)*step], \
  imgData[copyLower*step:(copyUpper-copyLower)*step], filter)
```c
for(int device = 0; device < ndevices; device++) {
    acc_set_device_num(device, acc_device_default);
    long lower = device*rows_per_device;
    long upper = MIN(lower + rows_per_device, h);
    long copyLower = MAX(lower - (filtersize/2), 0);
    long copyUpper = MIN(upper + (filtersize/2), h);
    #pragma acc enter data create(imgData[copyLower*step:copyUpper-copyLower+1*step], \\
                                out[lower*step:(upper-lower+1)*step]) copyin(filter[:5][:5])
}
```

```c
for(int device = 0; device < ndevices; device++) {
    acc_set_device_num(device, acc_device_default);
    printf("Launching device %d\n", device);
    long lower = device*rows_per_device;
    long upper = MIN(lower + rows_per_device, h);
    long copyLower = MAX(lower - (filtersize/2), 0);
    long copyUpper = MIN(upper + (filtersize/2), h);
    #pragma acc update device(imgData[copyLower*step:copyUpper-copyLower+1*step]) async \\
                     #pragma acc parallel loop present(filter, \\
                                imgData[copyLower*step:copyUpper-copyLower+1*step], \\
                                out[lower*step:(upper-lower+1)*step]) async \\
                  for(y = lower; y < upper; y++) {
        // Apply Filter
    }
    #pragma acc update self(out[lower*step:(upper-lower+1)*step]) async \\
}
```

```c
for(int device = 0; device < ndevices; device++) {
    acc_set_device_num(device, acc_device_default);
    long lower = device*rows_per_device;
    long upper = MIN(lower + rows_per_device, h);
    long copyLower = MAX(lower - (filtersize/2), 0);
    long copyUpper = MIN(upper + (filtersize/2), h);
    #pragma acc exit data delete(out[lower*step:(upper-lower+1)*step], \\
                                 imgData[copyLower*step:copyUpper-copyLower+1*step], filter)
}
```
SAMPLE FILTER CODE (1: MANY)

```c
for(int device = 0; device < ndevices; device++) {
    acc_set_device_num(device, acc_device_default);
    long lower = device*rows_per_device;
    long upper = MIN(lower + rows_per_device, h);
    long copyLower = MIN(lower + (filterSize/2), 0);
    long copyUpper = MIN(upper + (filterSize/2), h);
    #pragma acc enter data
    create(imgData[copyLower*step:(copyUpper-copyLower)*step], \
            out[lower*step:(upper-lower)*step]);
    copyin(filter[:5][:5])
}
```

```c
for(int device = 0; device < ndevices; device++) {
    acc_set_device_num(device, acc_device_default);
    printf("Launching device %d\n", device);
    long lower = device*rows_per_device;
    long upper = MIN(lower + rows_per_device, h);
    long copyLower = MAX(lower - (filterSize/2), 0);
    long copyUpper = MIN(upper + (filterSize/2), h);
    #pragma acc update device(imgData[copyLower*step:(copyUpper-copyLower)*step]) async
    #pragma acc parallel loop present(filter, \
            imgData[copyLower*step:(copyUpper-copyLower)*step], async
    for(y = lower; y < upper; y++) {
        for(x = 0; x < w; x++) {
            double blue = 0.0, green = 0.0, red = 0.0;
            for(int fy = 0; fy < filterSize; fy++) {
                long iy = y - (filterSize/2) + fy;
                for(int fx = 0; fx < filterSize; fx++) {
                    long ix = x - (filterSize/2) + fx;
                    if((iy<0) || (ix<0) || (iy>=h) || (ix>=w)) continue;
                    blue += filter[fy][fx] * (double)imgData[iy*step + ix * ch];
                    green += filter[fy][fx] * (double)imgData[iy*step + ix * ch + 1];
                    red += filter[fy][fx] * (double)imgData[iy*step + ix * ch + 2];
                }
            }
            out[y*step + x * ch] = 255 - (scale * blue);
            out[y*step + x * ch + 1] = 255 - (scale * green);
            out[y*step + x * ch + 2] = 255 - (scale * red);
        }
    }
    #pragma acc update self(out[lower*step:(upper-lower)*step]) async
}
```

```c
for(int device = 0; device < ndevices; device++) {
    acc_set_device_num(device, acc_device_default);
    long lower = device*rows_per_device;
    long upper = MIN(lower + rows_per_device, h);
    long copyLower = MAX(lower - (filterSize/2), 0);
    long copyUpper = MIN(upper + (filterSize/2), h);
    #pragma acc wait
    long lower = device*rows_per_device;
    long upper = MIN(lower + rows_per_device, h);
    long copyLower = MAX(lower - (filterSize/2), 0);
    long copyUpper = MIN(upper + (filterSize/2), h);
    #pragma acc update device(imgData[copyLower*step:(copyUpper-copyLower)*step], filter)
    }
```
1: MANY - THE GOOD & BAD

- The Good
  - Relatively simple to understand

- The Bad
  - Adds code that will be superfluous with single or no device
  - One thread will eventually become a bottleneck
MANY (THREADS) : 1
MANY (THREADS) : 1 - THE PLAN

1. Spawn 1 thread per GPU using OpenMP (any threading model will do)
2. Set the device within each thread.
3. Assign an equal portion of the work on each thread.
4. The threads will automatically synchronize when all threads are complete.
SAMPLE FILTER CODE (OPENMP)

```c
long rows_per_device = (h+(ndevices-1))/ndevices;
#pragma omp parallel for num_threads(acc_get_num_devices(acc_device_type_default))
    for(int device = 0; device < ndevices; device++) {
        long lower = device*rows_per_device;
        long upper = MIN(lower + rows_per_device, h);
        long copyLower = MAX(lower-(filtersize/2), 0);
        long copyUpper = MIN(upper+(filtersize/2), h);

        acc_set_device_num(device, acc_device_default);

        #pragma acc declare copyin(filter)
        #pragma acc parallel loop \
        copyin(imgData[copyLower*step:(copyUpper-copyLower)*step]) \ 
        copyout(out[lower*step:(upper-lower)*step]) present(filter)
            for(y = lower; y < upper; y++) {
                for(x = 0; x < w; x++) {
                    // Apply Filter
                }
            }
    }
} // end omp parallel for
```
1:MANY - THE GOOD & BAD

- The Good
  - Relatively simple to understand
  - Fewer code changes
- The Bad
  - Introduces an additional threading API
  - Need to be conscious of affinity (OMP_PLACES)
### ABOUT AFFINITY

```
$ nvidia-smi topo -m

<table>
<thead>
<tr>
<th></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>GPU0</td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV2</td>
<td>NV2</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>PIX</td>
<td>SYS</td>
<td>PHB</td>
<td>SYS</td>
<td>0-19,40-59</td>
</tr>
<tr>
<td>GPU1</td>
<td>NV1</td>
<td>X</td>
<td>NV2</td>
<td>NV1</td>
<td>NV2</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>PIX</td>
<td>SYS</td>
<td>PHB</td>
<td>SYS</td>
<td>0-19,40-59</td>
</tr>
<tr>
<td>GPU2</td>
<td>NV1</td>
<td>NV2</td>
<td>X</td>
<td>NV2</td>
<td>SYS</td>
<td>SYS</td>
<td>NV1</td>
<td>SYS</td>
<td>PHI</td>
<td>SYS</td>
<td>PIX</td>
<td>SYS</td>
<td>0-19,40-59</td>
</tr>
<tr>
<td>GPU3</td>
<td>NV2</td>
<td>NV1</td>
<td>NV2</td>
<td>X</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>NV1</td>
<td>PHB</td>
<td>SYS</td>
<td>PIX</td>
<td>SYS</td>
<td>0-19,40-59</td>
</tr>
<tr>
<td>GPU4</td>
<td>NV2</td>
<td>SYS</td>
<td>SYS</td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV2</td>
<td>SYS</td>
<td>PIX</td>
<td>SYS</td>
<td>PHB</td>
<td>20-39,60-79</td>
<td></td>
</tr>
<tr>
<td>GPU5</td>
<td>SYS</td>
<td>NV2</td>
<td>SYS</td>
<td>SYS</td>
<td>NV1</td>
<td>X</td>
<td>NV2</td>
<td>NV1</td>
<td>SYS</td>
<td>PIX</td>
<td>SYS</td>
<td>PHB</td>
<td>20-39,60-79</td>
</tr>
<tr>
<td>GPU6</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>NV1</td>
<td>NV2</td>
<td>X</td>
<td>NV2</td>
<td>SYS</td>
<td>PHB</td>
<td>SYS</td>
<td>PIX</td>
<td>20-39,60-79</td>
<td></td>
</tr>
<tr>
<td>GPU7</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>NV1</td>
<td>NV2</td>
<td>X</td>
<td>SYS</td>
<td>SYS</td>
<td>X</td>
<td>SYS</td>
<td>PHB</td>
<td>20-39,60-79</td>
<td></td>
</tr>
<tr>
<td>mlx5_0</td>
<td>PIX</td>
<td>PIX</td>
<td>PHB</td>
<td>PHB</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>X</td>
<td>SYS</td>
<td>PHB</td>
<td>SYS</td>
<td>20-39,60-79</td>
<td></td>
</tr>
<tr>
<td>mlx5_2</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>PIX</td>
<td>PIX</td>
<td>PHB</td>
<td>PHB</td>
<td>SYS</td>
<td>X</td>
<td>SYS</td>
<td>PHB</td>
<td>20-39,60-79</td>
<td></td>
</tr>
<tr>
<td>mlx5_1</td>
<td>PHB</td>
<td>PHB</td>
<td>PIX</td>
<td>PIX</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>PHB</td>
<td>SYS</td>
<td>X</td>
<td>SYS</td>
<td>20-39,60-79</td>
<td></td>
</tr>
<tr>
<td>mlx5_3</td>
<td>SYS</td>
<td>SYS</td>
<td>SYS</td>
<td>PHB</td>
<td>PHB</td>
<td>PIX</td>
<td>PIX</td>
<td>SYS</td>
<td>PHB</td>
<td>SYS</td>
<td>X</td>
<td>20-39,60-79</td>
<td></td>
</tr>
</tbody>
</table>
```

- GPUs have affinity to certain CPUs, meaning they connect directly rather than through an inter-CPU bus
- OMP_PLACES, numctl, or your job launcher can help place your threads close to each GPU
MANY (THREADS) : 1 (IMPROVED)
MANY (THREADS) : 1 IMPROVED - THE PLAN

1. Spawn 1 thread per GPU using OpenMP (any threading model will do)
2. Set the device within each thread.
3. Assign an equal portion of the work on each thread.
4. Break the work further into blocks and use asynchronous queues to overlap copies and computation
5. Synchronize when all threads are complete.
SAMPLE FILTER CODE

```c
int ndevices = acc_get_num_devices(acc_device_default);
long rows_per_device = (h+(ndevices-1))/ndevices; long rows_per_block = (h+(ndevices*3-1))/(ndevices*3);
#pragma omp parallel num_threads(ndevices)
{
    int tid = omp_get_thread_num();
    acc_set_device_num(tid, acc_device_default);
    #pragma acc declare copyin(filter)
    long lower = tid*rows_per_device; long upper = MIN(lower + rows_per_device, h);
    long copyLower = MAX(lower-(filtersize/2), 0); long copyUpper = MIN(upper+(filtersize/2), h);
    #pragma acc data create(imgData[copyLower*step:(copyUpper-copyLower)*step])
        create(out[lower*step:(upper-lower)*step])
    #pragma omp for
    for(int block = 0; block < ndevices*3; block++) {
        lower = block*rows_per_block; upper = MIN(lower + rows_per_block, h);
        copyLower = MAX(lower-(filtersize/2), 0); copyUpper = MIN(upper+(filtersize/2), h);
        #pragma acc update device(imgData[copyLower*step:(copyUpper-copyLower)*step]) async(block)
        #pragma acc parallel loop default(present) async(block)
        for(y = lower; y < upper; y++) {
            #pragma acc loop
            for(x = 0; x < w; x++) {
                // Apply Filter
            }
        }
        #pragma acc update self(out[lower*step:(upper-lower)*step]) async(block)
    } // end omp for
    #pragma acc wait
} // end omp parallel
```
1: MANY - THE GOOD & BAD

- The Good
  - Relatively simple to understand
  - Fewer code changes
  - Overlaps PCIe Copies

- The Bad
  - Introduces an additional threading API
  - Need to be conscious of affinity (OMP_PLACES)
1+ (RANK): 1 (GPU)
MANY (THREADS) : 1 - THE PLAN

1. Use MPI to decompose the work among ranks
2. Assign each rank a GPU
3. Reassemble the result
int ndevices = acc_get_num_devices(acc_device_default);
acc_set_device_num(rank*ndevices, acc_device_default);

MPI_Scatterv((void*) imgData, sendcounts, senddispls, MPI_UNSIGNED_CHAR,
(void*) (imgData + copyLower*step), (int) copySize, MPI_UNSIGNED_CHAR, 0,
MPI_COMM_WORLD);

#pragma acc parallel loop copyin(imgData[copyLower*step:copySize]) \
  copy(out[lower*step:size]) \ 
  copyin(filter[:5][:5])

for(y = lower; y < upper; y++) {
    for(x = 0; x < w; x++) {
        // Apply Filter
    }
}

MPI_Gatherv((void*) (out+lower*step), (int) size, MPI_UNSIGNED_CHAR,
(void*) out, recvcounts, recvdispls, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
1:1 - THE GOOD & BAD

- The Good
  - Little/No Code change if you already use MPI
  - Affinity handled by MPI launcher

- The Bad
  - Dependence on MPI (if not already using)
  - Need “enough” work to overcome communication cost

![Graph showing speed-up and time comparison between MPI (No Comm), Serial, and MPI (w/ Comm)]
1+ (RANK): 1 (GPU) (IMPROVED)
MANY (THREADS) : 1 IMPROVED - THE PLAN

1. Use MPI to decompose the work among ranks
2. Start CUDA MPS & launch multiple ranks / GPU
3. Assign each rank a GPU
4. Reassemble the result
SAMPLE FILTER CODE (MPI)

```c
int ndevices = acc_get_num_devices(acc_device_default);
int sharing = n ranks / ndevices;
acc_set_device_num(rank/sharing, acc_device_default);

MPI_Scatterv((void*) imgData, sendcounts, senddispls, MPI_UNSIGNED_CHAR,
              (void*) (imgData + copyLower*step), (int) copySize, MPI_UNSIGNED_CHAR, 0,
              MPI_COMM_WORLD);

#pragma acc parallel loop copyin(imgData[copyLower*step:copySize])
       copy(out[lower*step:size])
       copyin(filter[:5][:5])

    for(y = lower; y < upper; y++) {
        for(x = 0; x < w; x++) {
            // Apply Filter
        }
    }

MPI_Gatherv((void*) (out+lower*step), (int) size, MPI_UNSIGNED_CHAR,
            (void*) out, recvcounts, recvdispls, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
```
1:1 - THE GOOD & BAD

- The Good
  - Little/No Code change if you already use MPI
  - Affinity handled by MPI launcher
  - Memcopies overlapped automatically

- The Bad
  - Dependence on MPI (if not already using)
  - Need “enough” work to overcome communication cost
PERFORMANCE COMPARISON

- PGI 19.1, NVIDIA DGX-1V

<table>
<thead>
<tr>
<th>Number of GPUs</th>
<th>Avg. Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>2</td>
<td>0.03</td>
</tr>
<tr>
<td>3</td>
<td>0.04</td>
</tr>
<tr>
<td>4</td>
<td>0.05</td>
</tr>
<tr>
<td>5</td>
<td>0.06</td>
</tr>
</tbody>
</table>

- Serial
- 1:Many
- OpenMP
- OpenMP (Async)
- MPI
- MPI+MPS
WHAT ABOUT COMMUNICATION?
MULTI GPU PROGRAMMING
With MPI and OpenACC
MPI+OPENACC

//MPI rank 0
#pragma acc host_data use_device( sbuf )
MPI_Send(sbuf, size, MPI_DOUBLE, n-1, tag, MPI_COMM_WORLD);

//MPI rank n-1
#pragma acc host_data use_device( rbuf )
MPI_Recv(rbuf, size, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI+OPENACC (IMPROVED)

//MPI rank 0
#pragma acc update self( sbuf[0:size] )
MPI_Send(sbuf, size, MPI_DOUBLE, n-1, tag, MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(rbuf, size, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
#pragma acc update device( rbuf[0:size] )
CONCLUSIONS
CONCLUSIONS
The Hydra has been Tamed.

OpenACC provides a simple API for using multiple devices.

A single CPU thread is not sufficient to feed multiple GPUs.

Multiple OpenMP threads is a simple way to control multiple devices.

MPI is an even simpler way to control multiple threads, if you can overcome communication cost.

Overlapping data copies is still a benefit in multi-device codes.