# TAMING THE HYDRA MULTI-GPU PROGRAMMING WITH OPENACC

Jeff Larkin, GTC 2019, March 2019

# Why use Multiple Devices?

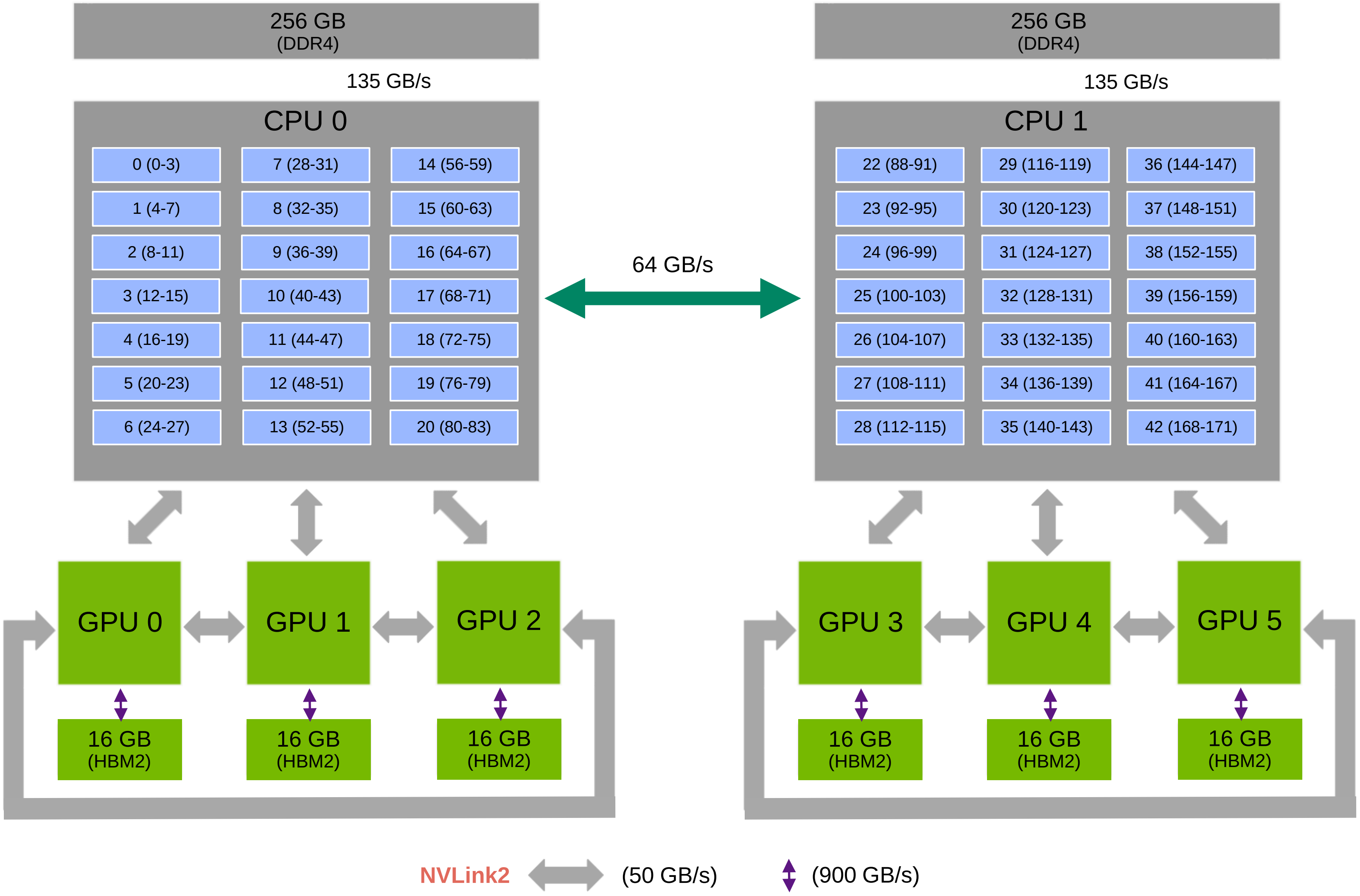
Because 2 > 1 : Effectively using multiple devices results in faster time to solution and/or solve more complex problems

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

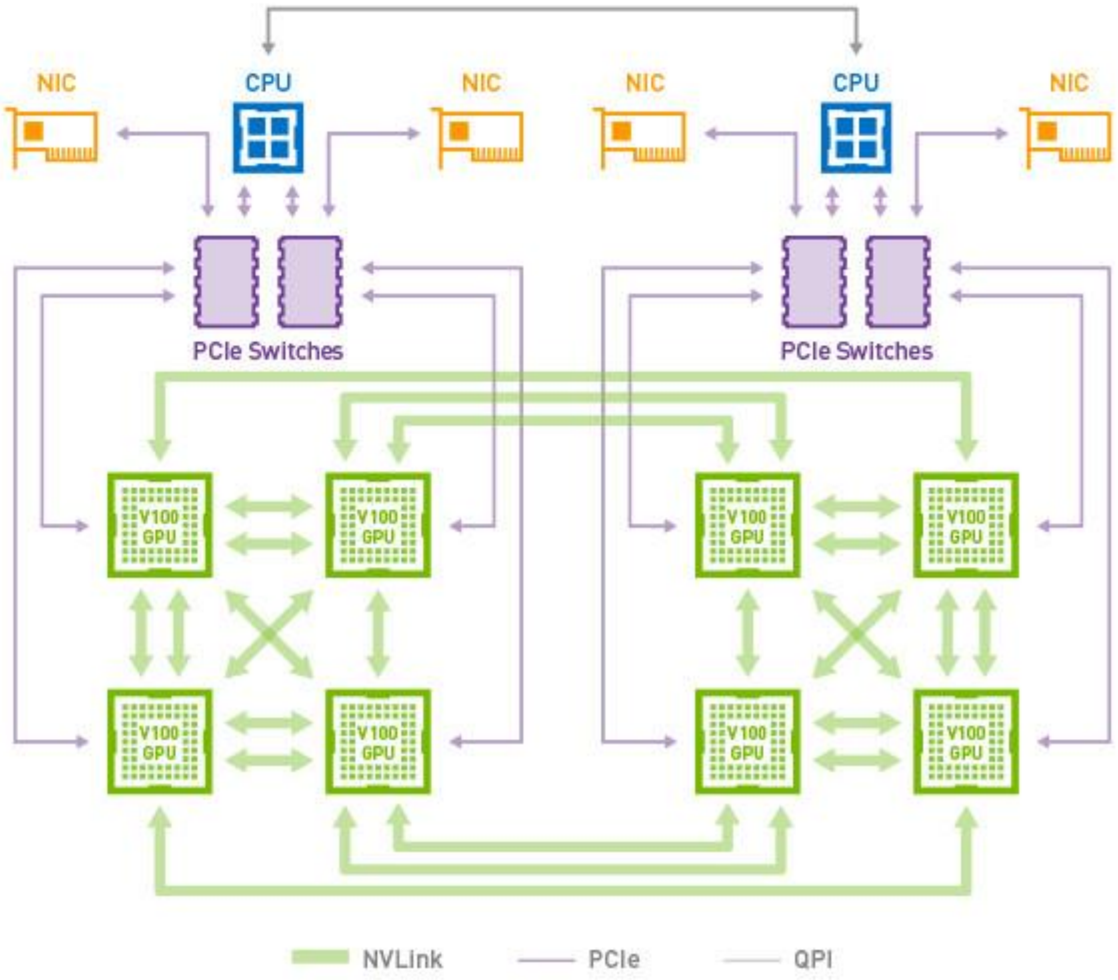
Because the code already runs on multiple nodes using MPI, so little change is necessary

# SUMMIT NODE

## (2) IBM POWER9 + (6) NVIDIA VOLTA V100

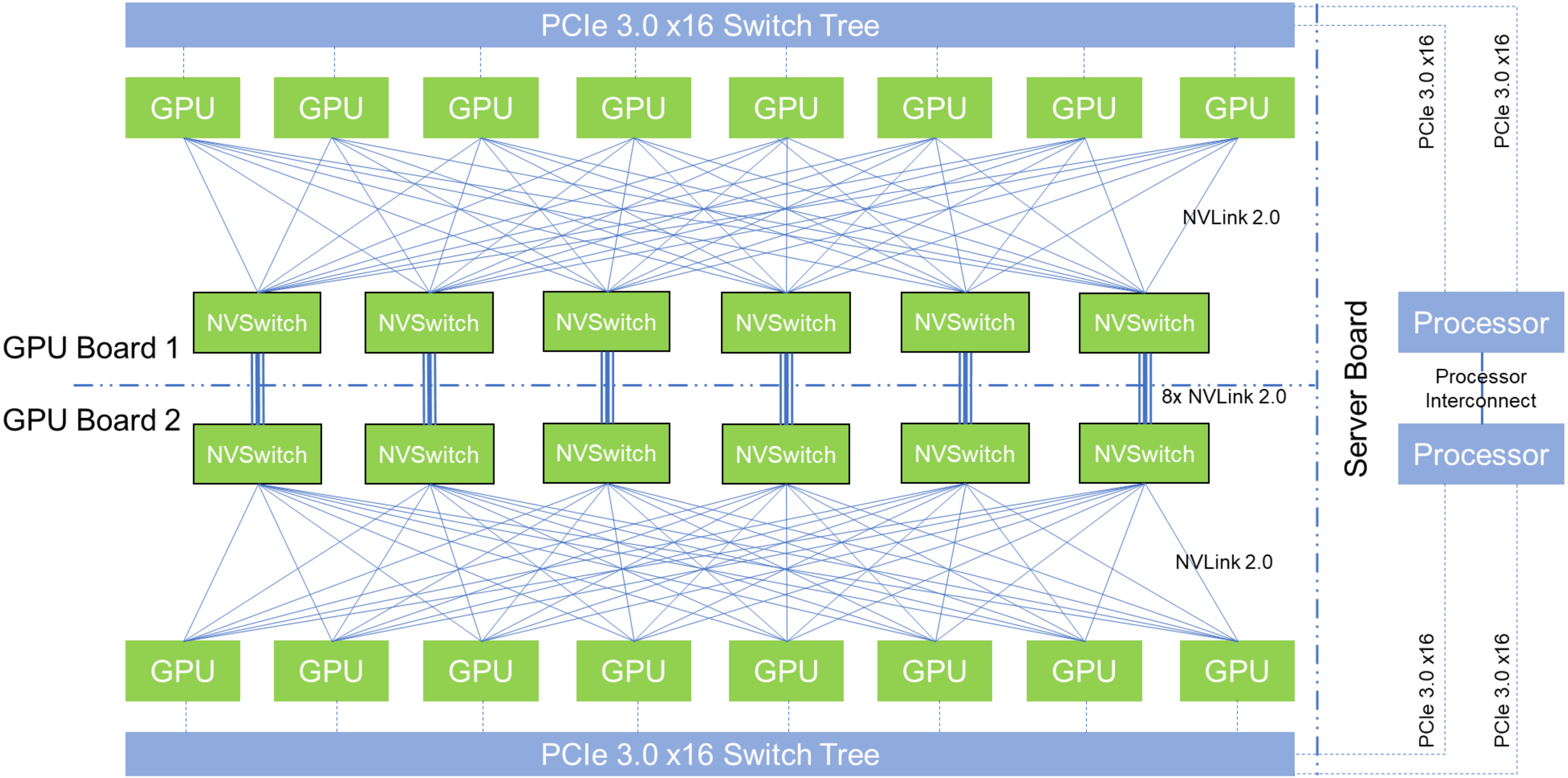


# DGX-1 NODE



NIC
CPU
NIC
NIC
CPU
NIC
PCIe Switches
PCIe Switches
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
NVLink
PCIe
QPI

# DGX-2 NODE


PCIe 3.0 x16 Switch Tree
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
NVLink 2.0
NVSwitch
NVSwitch
NVSwitch
NVSwitch
NVSwitch
NVSwitch
GPU Board 1
GPU Board 2
8x NVLink 2.0
NVSwitch
NVSwitch
NVSwitch
NVSwitch
NVSwitch
NVSwitch
NVLink 2.0
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
PCIe 3.0 x16 Switch Tree
Server Board
PCIe 3.0 x16
PCIe 3.0 x16
Processor
Processor
Interconnect
Processor
PCIe 3.0 x16
PCIe 3.0 x16

# 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

- Conceptually Simple
- Requires additional  loops
- CPU can become a bottleneck
- Remaining CPU cores often underutilized

1 CPU : Many GPUs

- Conceptually Very Simple
- Set and forget the device numbers
- Relies on external Threading API
- Can see improved utilization
- Watch affinity

Many Threads : 1 GPU (Each)

- Little to no code changes required
- Re-uses existing domain decomposition
- Probably already using MPI
- Watch affinity

1+ CPU : 1 GPU

- Easily share data between peer devices
- Coordinating between GPUs extremely tricky

Many Processes : Many GPUs

# 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

# SAMPLE CODE

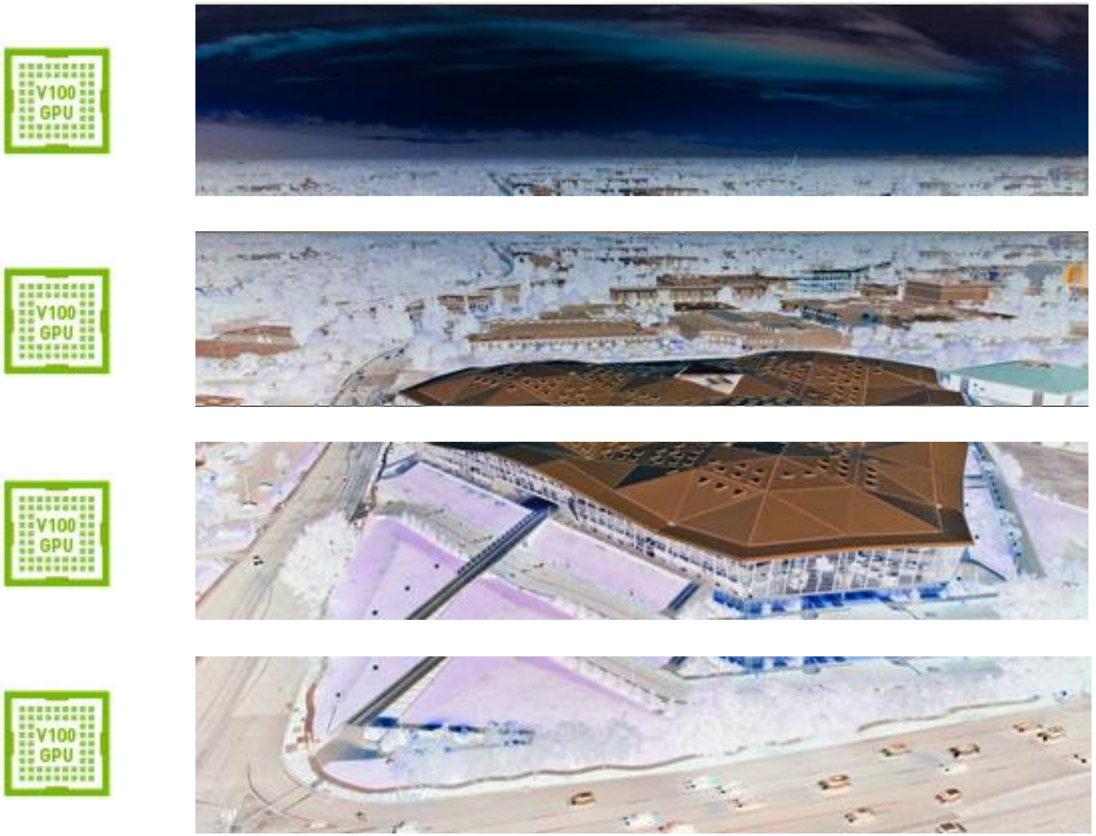
## Embarrassingly Parallel Image Filter

# SAMPLE CODE

## Embarrassingly Parallel Image Filter

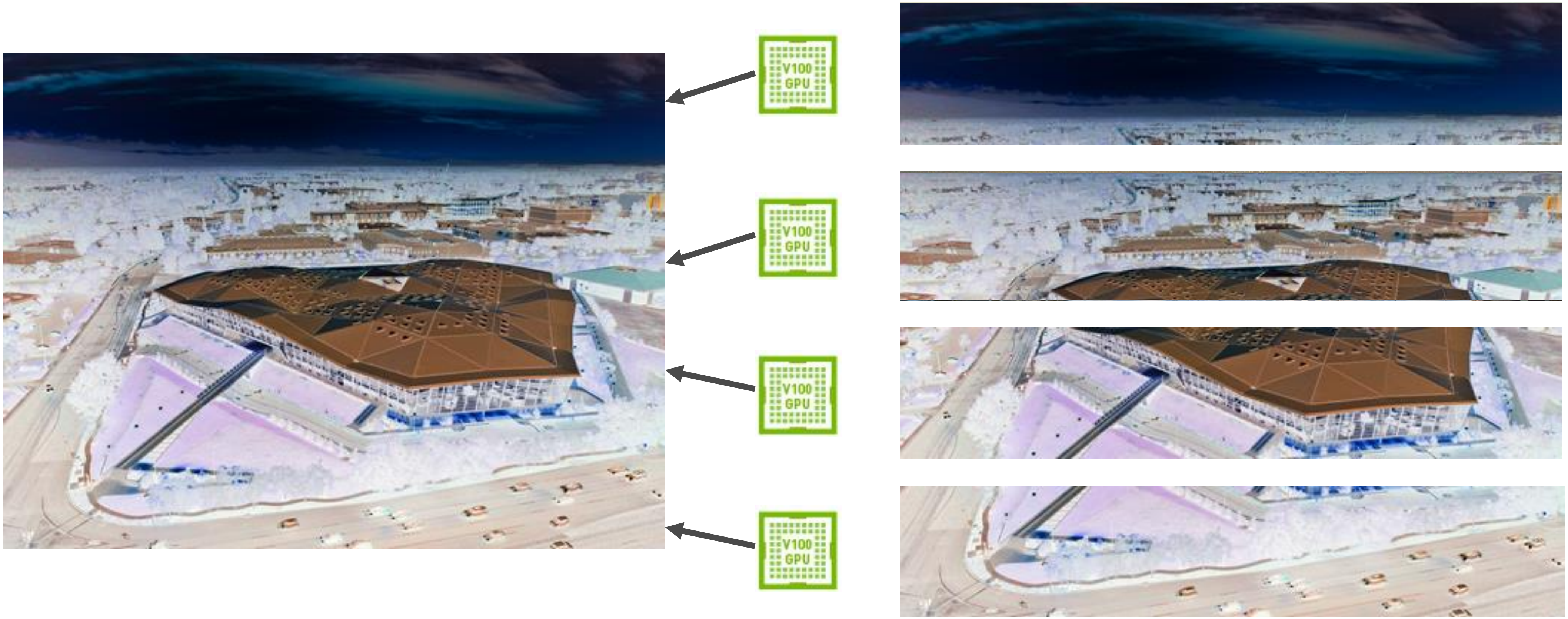
# SAMPLE CODE

## Embarrassingly Parallel Image Filter



# SAMPLE CODE

## Embarrassingly Parallel Image Filter



# SAMPLE FILTER CODE

```
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
         green += filter[fy][fx] * (double)imgData[iy * ste
         red   += filter[fy][fx] * (double)imgData[iy * step
       }
     }
     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);
   }
 }
}
```

For Each Pixel

Apply Filter

Store the Values

# 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.

# SAMPLE FILTER CODE (1:MANY)

```
 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]) \
 copyin(filter[:5][:5])
 }
  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], \
 out[lower*step:(upper-lower)*step]) async
    for(y = lower; y < upper; y++) {
#pragma acc loop
      for(x = 0; x < w; x++) {
        double blue = 0.0, green = 0.0, red = 0.0;
#pragma acc loop seq
        for(int fy = 0; fy < filtersize; fy++) {
          long iy = y - (filtersize/2) + fy;
#pragma acc loop seq
          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);
#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 exit data delete(out[lower*step:(upper-lower)*step], \
 imgData[copyLower*step:(copyUpper-copyLower)*step], filter)
 }
```


```
 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]) \
 copyin(filter[:5][:5])
}
```

# SAMPLE FILTER CODE (1:MANY)

```
 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]) \
 copyin(filter[:5][:5])
  }
```

```
  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], \
 out[lower*step:(upper-lower)*step]) async
    for(y = lower; y < upper; y++) {
#pragma acc loop
      for(x = 0; x < w; x++) {
        double blue = 0.0, green = 0.0, red = 0.0;
#pragma acc loop seq
        for(int fy = 0; fy < filtersize; fy++) {
          long iy = y - (filtersize/2) + fy;
#pragma acc loop seq
          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);
#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 exit data delete(out[lower*step:(upper-lower)*step], \
 imgData[copyLower*step:(copyUpper-copyLower)*step], filter)
  }
```

```
 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 update device(imgData[copyLower*step:(copyUpper-copyLower)*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++) {
// Apply Filter
      }
    }
#pragma acc update self(out[lower*step:(upper-lower)*step]) async
  }
```

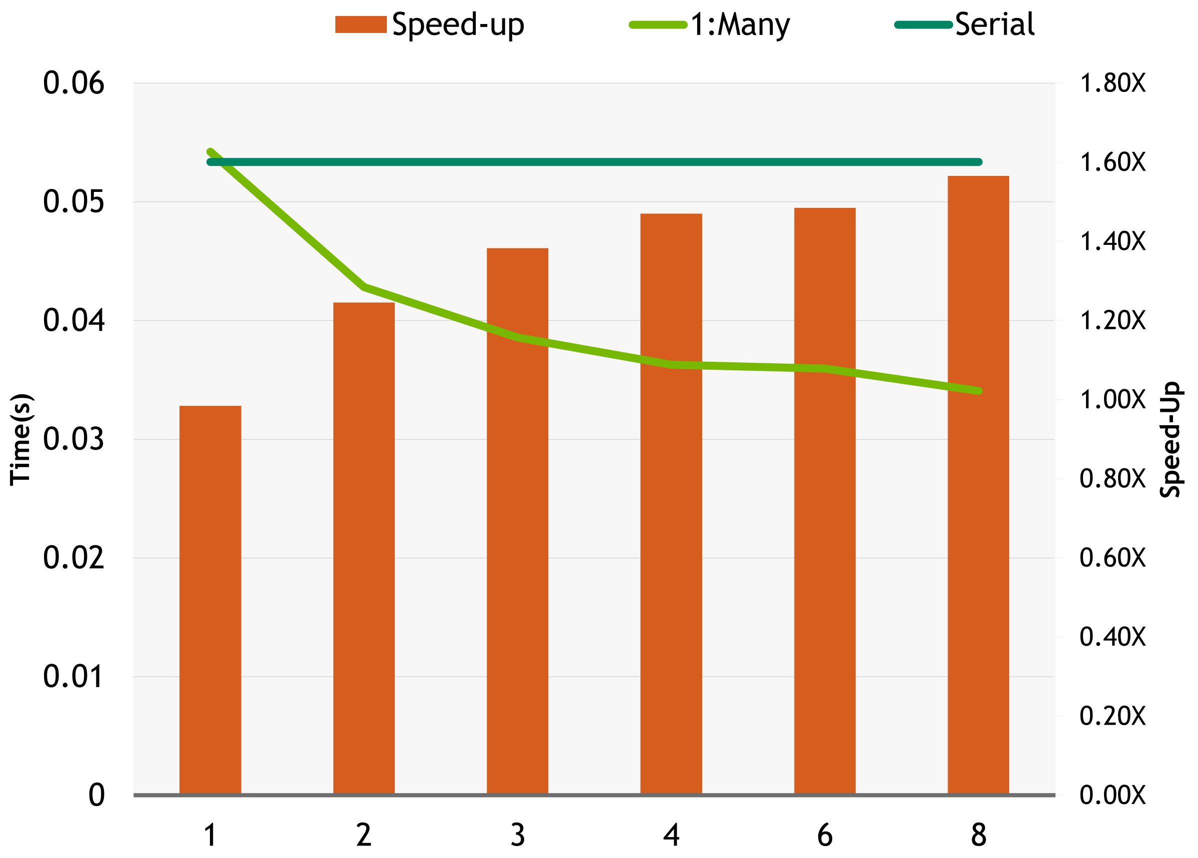
# SAMPLE FILTER CODE (1:MANY)

```
 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]) \
 copyin(filter[:5][:5])
  }

  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], \
 out[lower*step:(upper-lower)*step]) async
    for(y = lower; y < upper; y++) {
#pragma acc loop
      for(x = 0; x < w; x++) {
        double blue = 0.0, green = 0.0, red = 0.0;
#pragma acc loop seq
        for(int fy = 0; fy < filtersize; fy++) {
          long iy = y - (filtersize/2) + fy;
#pragma acc loop seq
          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);
#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 exit data delete(out[lower*step:(upper-lower)*step], \
 imgData[copyLower*step:(copyUpper-copyLower)*step], filter)
  }
```

```
 for(int device = 0; device < ndevices; device++) {
    acc_set_device_num(device, acc_device_default);
#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 exit data delete(out[lower*step:(upper-
lower)*step], \
 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

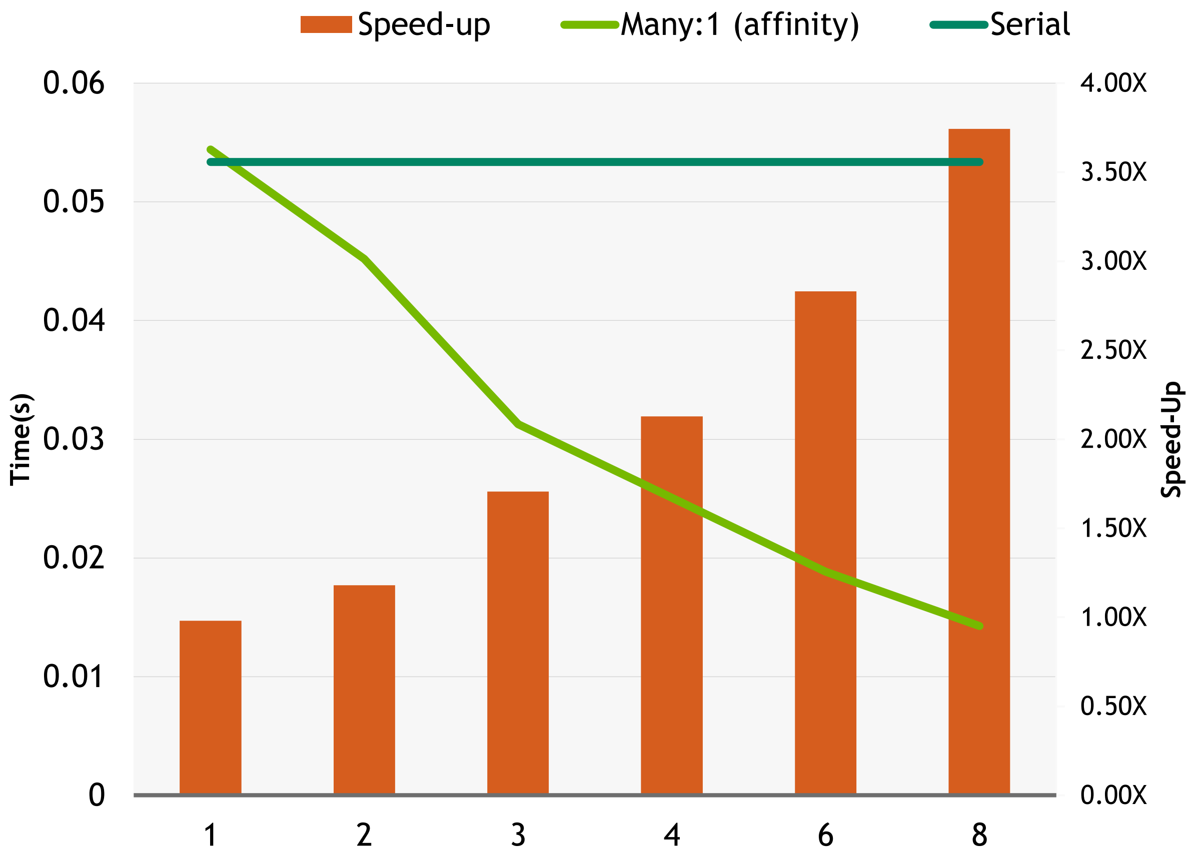
1. Spawn 1 thread per GPU using OpenMP (any threading model will do)
2. Set the device within each thread.
3. Assign an equal portion of the work on each thread.
4. The threads will automatically synchronize when all threads are complete.

# SAMPLE FILTER CODE (OPENMP)

```
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
```

| | GPU0 | GPU1 | GPU2 | GPU3 | GPU4 | GPU5 | GPU6 | GPU7 | mlx5_0 | mlx5_2 | mlx5_1 | mlx5_3 | CPU Affinity |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GPU0 | X | NV1 | NV1 | NV2 | NV2 | SYS | SYS | SYS | PIX | SYS | PHB | SYS | 0-19,40-59 |
| GPU1 | NV1 | X | NV2 | NV1 | SYS | NV2 | SYS | SYS | PIX | SYS | PHB | SYS | 0-19,40-59 |
| GPU2 | NV1 | NV2 | X | NV2 | SYS | SYS | NV1 | SYS | PHB | SYS | PIX | SYS | 0-19,40-59 |
| GPU3 | NV2 | NV1 | NV2 | X | SYS | SYS | SYS | NV1 | PHB | SYS | PIX | SYS | 0-19,40-59 |
| GPU4 | NV2 | SYS | SYS | SYS | X | NV1 | NV1 | NV2 | SYS | PIX | SYS | PHB | 20-39,60-79 |
| GPU5 | SYS | NV2 | SYS | SYS | NV1 | X | NV2 | NV1 | SYS | PIX | SYS | PHB | 20-39,60-79 |
| GPU6 | SYS | SYS | NV1 | SYS | NV1 | NV2 | X | NV2 | SYS | PHB | SYS | PIX | 20-39,60-79 |
| GPU7 | SYS | SYS | SYS | NV1 | NV2 | NV1 | NV2 | X | SYS | PHB | SYS | PIX | 20-39,60-79 |
| mlx5_0 | PIX | PIX | PHB | PHB | SYS | SYS | SYS | SYS | X | SYS | PHB | SYS | |
| mlx5_2 | SYS | SYS | SYS | SYS | PIX | PIX | PHB | PHB | SYS | X | SYS | PHB | |
| mlx5_1 | PHB | PHB | PIX | PIX | SYS | SYS | SYS | SYS | PHB | SYS | X | SYS | |
| mlx5_3 | SYS | SYS | SYS | SYS | PHB | PHB | PIX | PIX | SYS | PHB | SYS | X | |

- 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

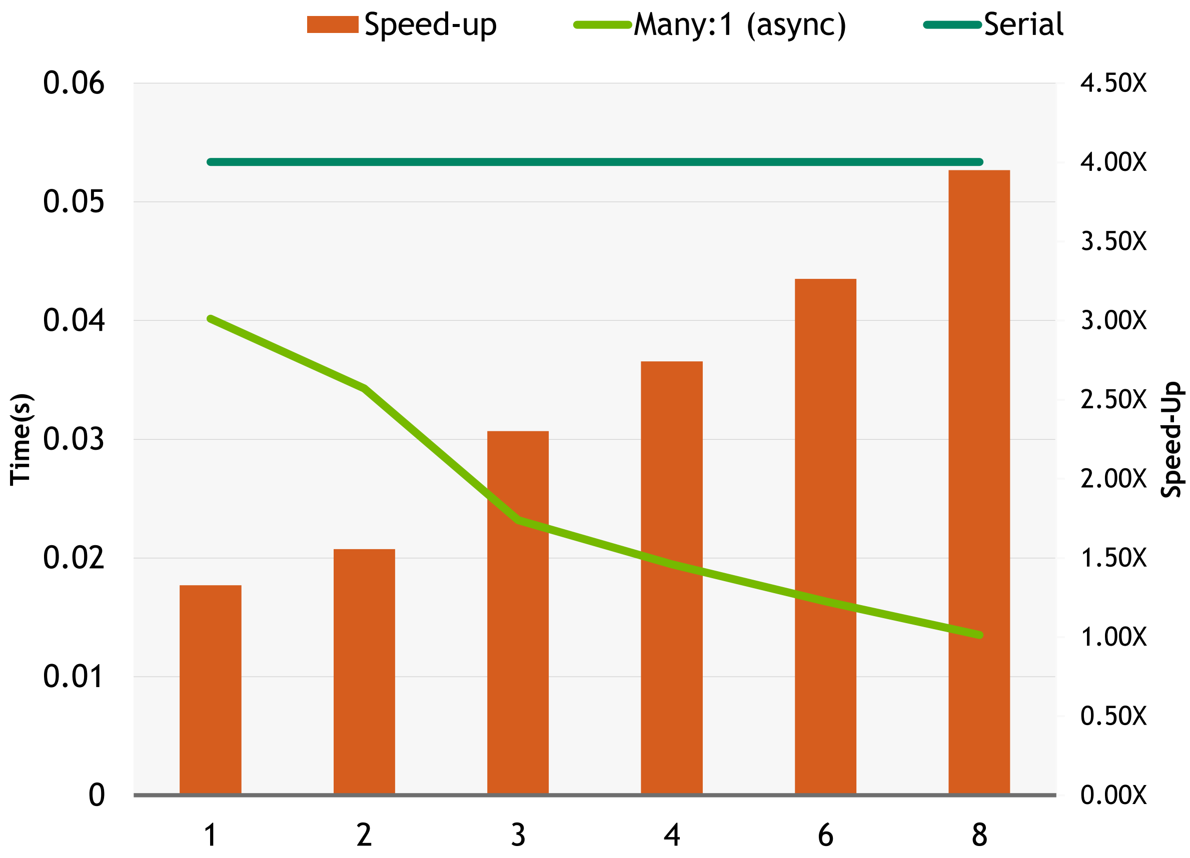
1. Spawn 1 thread per GPU using OpenMP (any threading model will do)
2. Set the device within each thread.
3. Assign an equal portion of the work on each thread.
4. Break the work further into blocks and use asynchronous queues to overlap copies and computation
5. Synchronize when all threads are complete.

# SAMPLE FILTER CODE

```
 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

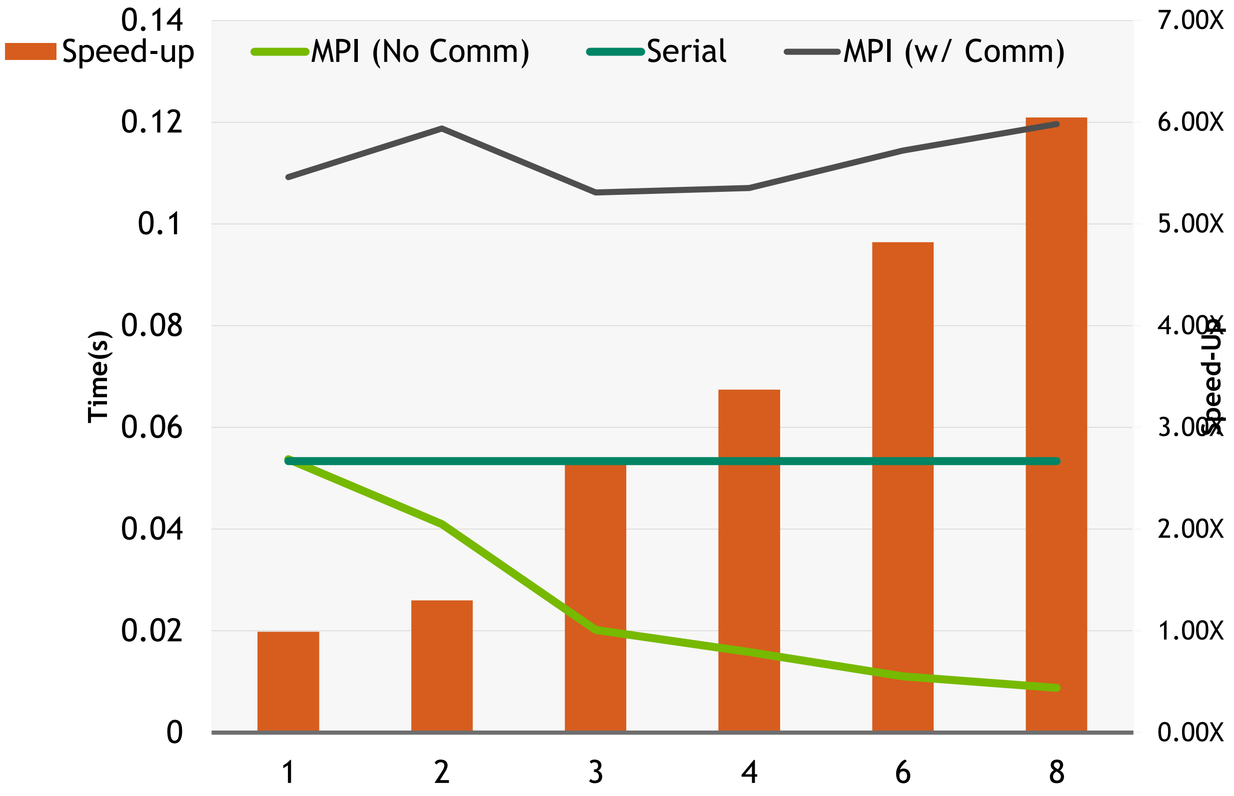
1. Use MPI to decompose the work among ranks
2. Assign each rank a GPU
3. Reassemble the result

# SAMPLE FILTER CODE (MPI)

```
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



# 1+ (RANK): 1 (GPU) (IMPROVED)

# MANY (THREADS) : 1 IMPROVED – THE PLAN

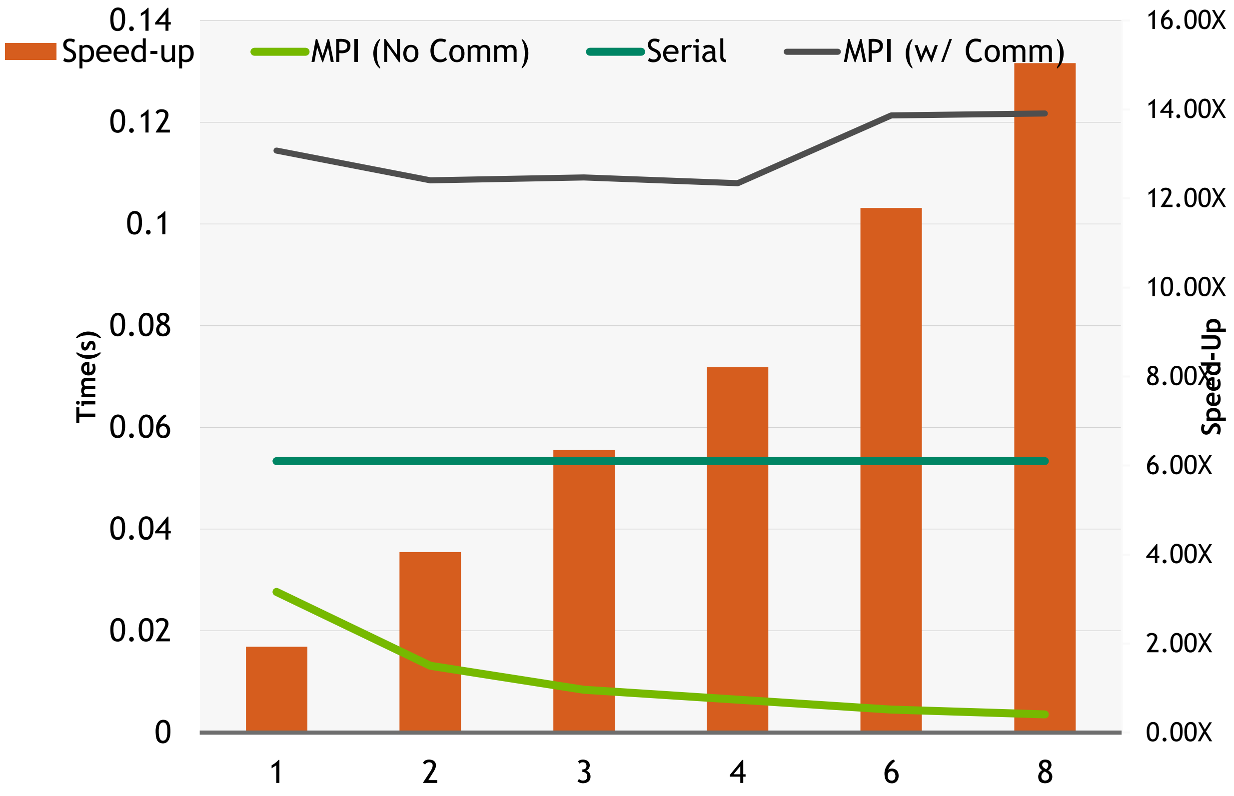
1. Use MPI to decompose the work among ranks
2. Start CUDA MPS & launch multiple ranks / GPU
3. Assign each rank a GPU
4. Reassemble the result

# SAMPLE FILTER CODE (MPI)

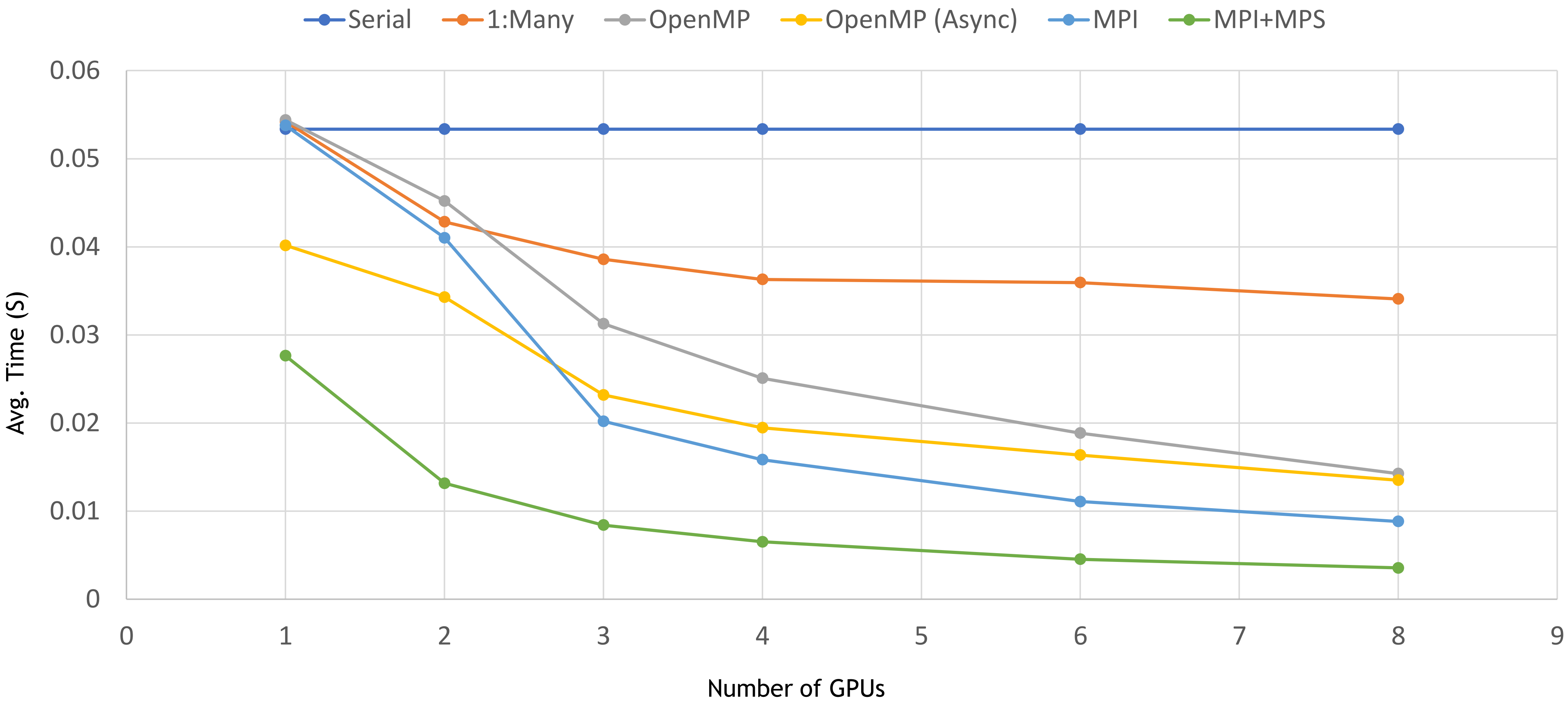
```
int ndevices = acc_get_num_devices(acc_device_default);
int sharing = nranks / ndevices;
acc_set_device_num(rank/sharing, acc_device_default);

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

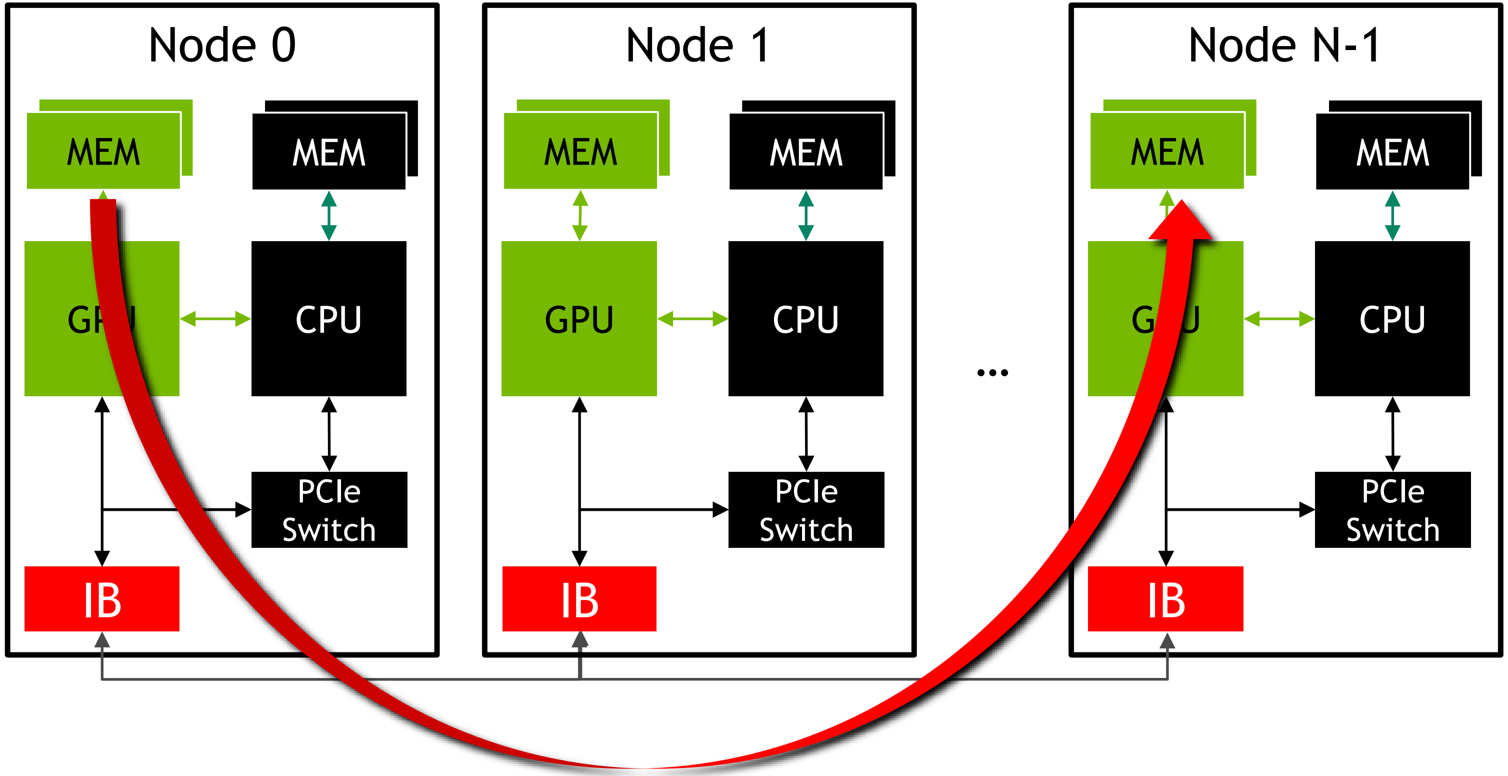



*PGI 19.1, NVIDIA DGX-1V*

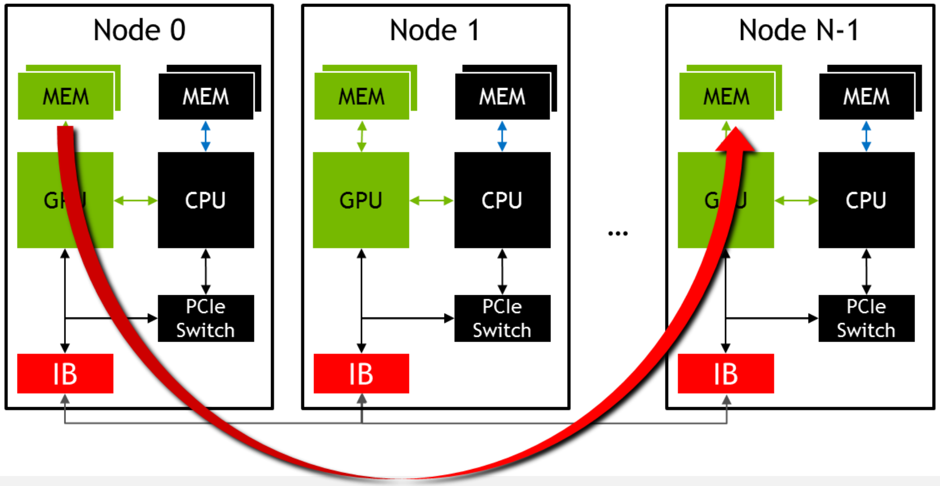
# WHAT ABOUT COMMUNICATION?

# MULTI GPU PROGRAMMING

## With MPI and OpenACC

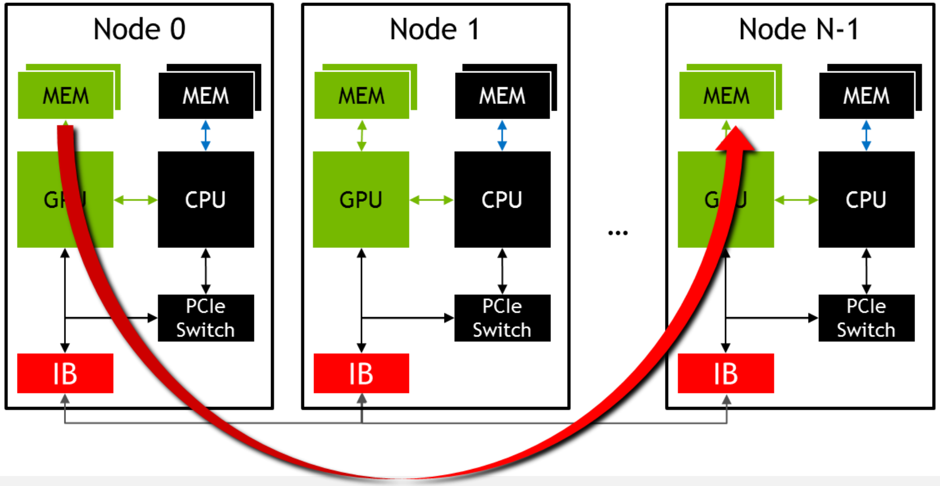


# MPI+OPENACC




```
//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