MODULE NINE: MULTI-DEVICE PROGRAMMING

Speaker, Date

OpenACC
More Science, Less Programming

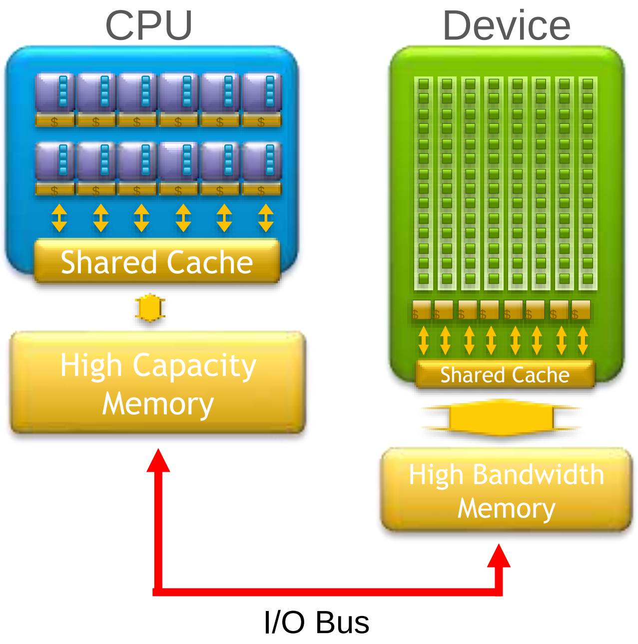
MODULE OVERVIEW

Topics to be covered

- Programming for a multiple-device architecture with:
- OpenACC
- OpenMP + OpenACC
- MPI + OpenACC
- How these concepts scale to a very large, multi-node systems

MULTI-DEVICE ARCHITECTURE

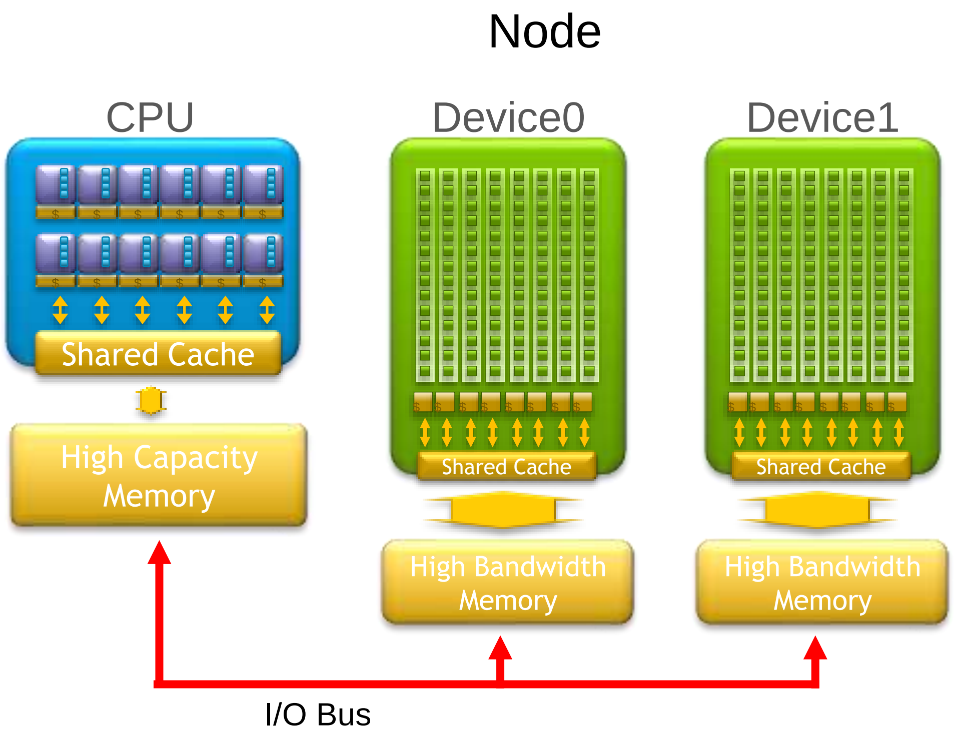
MULTIPLE DEVICES ON A SINGLE MACHINE



- So far we have only programmed for a single-device system
- However, there are many machines that contain more than one device
- A very common example of this is a device with multiple GPUs
- In this module, we will discuss how to implement multi-device programming in several different ways

MULTIPLE DEVICES ON A SINGLE MACHINE

- **In order to utilize multiple devices, we have 3 options:**
- Queue up multiple device operations using the OpenACC built-in functions and async clause
- Use OpenMP threads and assign each a separate device
- Use MPI and assign each MPI rank a separate device



OpenACC

MULTIDEVICE WITH OPENACC ONLY

MULTIDEVICE WITH OPENACC ONLY

- To utilize multiple devices with OpenACC, we will need to use some functions built-in to the OpenACC specification
- We will also need to use some of the concepts from OpenACC asynchronous programming

FIND NUMBER OF AVAILABLE DEVICES

```
int num_devices = acc_get_num_devices(acc_device_default);
```

- Each device on the machine will be assigned a unique number (starting at zero)
- We will use the number of available devices to split up the computational work between them

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.30                 Driver Version: 390.30                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K20Xm         Off  | 00000000:02:00.0 Off |                    0 |
| N/A   39C    P0    57W / 235W |      0MiB /  5700MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20Xm         Off  | 00000000:03:00.0 Off |                    0 |
| N/A   53C    P0    59W / 235W |      0MiB /  5700MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K20Xm         Off  | 00000000:83:00.0 Off |                    0 |
| N/A   42C    P0    57W / 235W |      0MiB /  5700MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K20Xm         Off  | 00000000:84:00.0 Off |                    0 |
| N/A   40C    P0    61W / 235W |      0MiB /  5700MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+
```

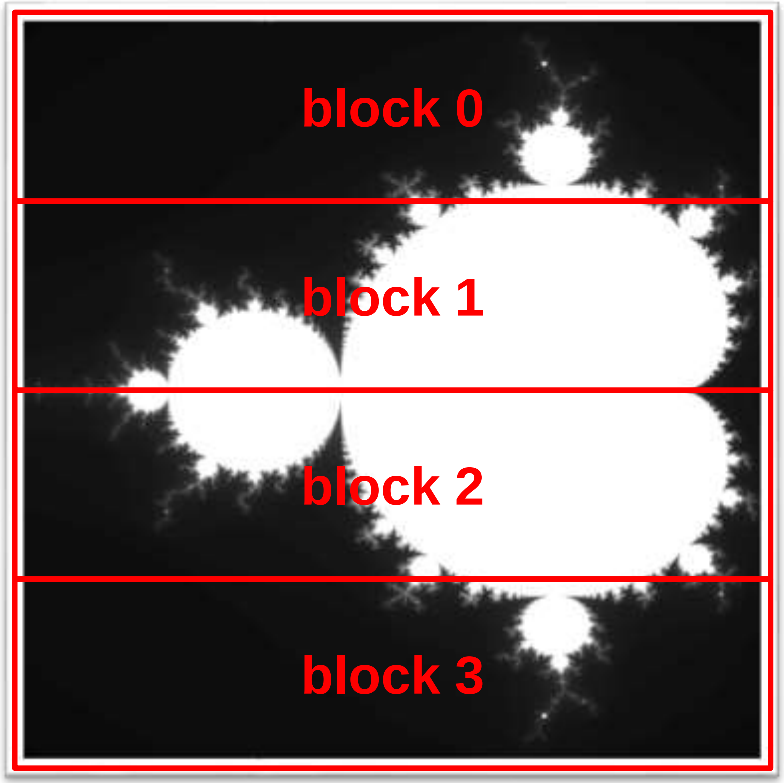
CHANGE ACTIVE DEVICE

```
acc_set_device_num(device_num, acc_device_default);
```

```
acc_set_device_num(0, acc_device_default);
#pragma acc parallel loop async
for(int i = 0; i < size; i++) {
    ...
}

acc_set_device_num(1, acc_device_default);
#pragma acc parallel loop async
for(int j = 0; j < size; j++) {
    ...
}
```

- This function will switch the active device
- By using this function, we will be able to queue up a device operation and then switch to another device
- By marking the loop as async, the code is able to move to the next loop, and launch it on a separate device without waiting for the first loop to finish

REVISITING MANDELBROT



- To demonstrate the different methods for multidevice programming, we will revisit the mandelbrot code
- Our goal with this code is to break the image into several blocks, and then compute each block on a separate device
- We will first walk through the pure OpenACC code

MULTI-DEVICE MANDELBROT: DATA

Loop over all devices and allocate memory on each

Loop over all devices again to deallocate memory

```
int main() {
     ...
     int ndevices = acc_get_num_devices(acc_device_default);
     int block_size = WIDTH*HEIGHT/ndevices;
     for(int device=0; device < ndevices; device++) {
          acc_set_device_num(device, acc_device_default);
#pragma acc enter data \
   create(image[block_size*device:block_size])
     }

     ... // Compute

     for(int device=0; device < ndevices; device++) {
          acc_set_device_num(device, acc_device_default);
#pragma acc exit data delete(image)
     }
}
```

MULTI-DEVICE MANDELBROT: DATA

- We need to allocate the memory ahead of time because allocation/deallocation does not always work asynchronously
- We also avoid allocating the entire array on each device just because we do not need to

```
int main() {
     ...
     int ndevices = acc_get_num_devices(acc_device_default);
     int block_size = WIDTH*HEIGHT/ndevices;
     for(int device=0; device < ndevices; device++) {
          acc_set_device_num(device, acc_device_default);
#pragma acc enter data \
   create(image[block_size*device:block_size])
     }

     ... // Compute

     for(int device=0; device < ndevices; device++) {
          acc_set_device_num(device, acc_device_default);
#pragma acc exit data delete(image)
     }
}
```

MULTI-DEVICE MANDELBROT: COMPUTE

Loop over all devices

Launch the compute (async)

Launch the data transfer (async)

Continue to the next device

```
int main() {
     int ndevices = acc_get_num_devices(acc_device_default);
     int block_size = WIDTH*HEIGHT/ndevices;
     ... // Allocate Data
     for(int device=0; device<ndevices; device++) {
          int yStart = device*(HEIGHT/ndevices);
          int yEnd   = yStart+(HEIGHT/ndevices);
          acc_set_device_num(device, acc_device_default);
#pragma acc parallel loop async
          for(int y=yStart;y<yEnd;y++) {
               for(int x=0;x<WIDTH;x++) {
                    image[y*WIDTH+x]=mandelbrot(x,y);
               }
          }
#pragma acc update \
 host(image[yStart*WIDTH:block_size]) async


     }
     ... // Wait
     ... // Deallocate Data
}
```

MULTI-DEVICE MANDELBROT: COMPUTE

- When using async for multiple devices, each device will have separate *async queues*
- So when we have: `#pragma acc parallel loop async`
- Each device will use its own default async queue automatically
- Lastly, when using multiple devices, we will have to do a little bit of extra work to ensure that all devices finish before moving on with other code…

```
int main() {
     int ndevices = acc_get_num_devices(acc_device_default);
     int block_size = WIDTH*HEIGHT/ndevices;
     ... // Allocate Data
     for(int device=0; device<ndevices; device++) {
          int yStart = device*(HEIGHT/ndevices);
          int yEnd   = yStart+(HEIGHT/ndevices);
          acc_set_device_num(device, acc_device_default);
#pragma acc parallel loop async
          for(int y=yStart;y<yEnd;y++) {
               for(int x=0;x<WIDTH;x++) {
                    image[y*WIDTH+x]=mandelbrot(x,y);
               }
          }
#pragma acc update \
 host(image[yStart*WIDTH:block_size]) async


     }
     ... // Wait
     ... // Deallocate Data
}
```

MULTI-DEVICE MANDELBROT: WAIT

- A single wait is not able to cover all devices
- For multi-device, we need to loop over each device, and tell it individually to wait

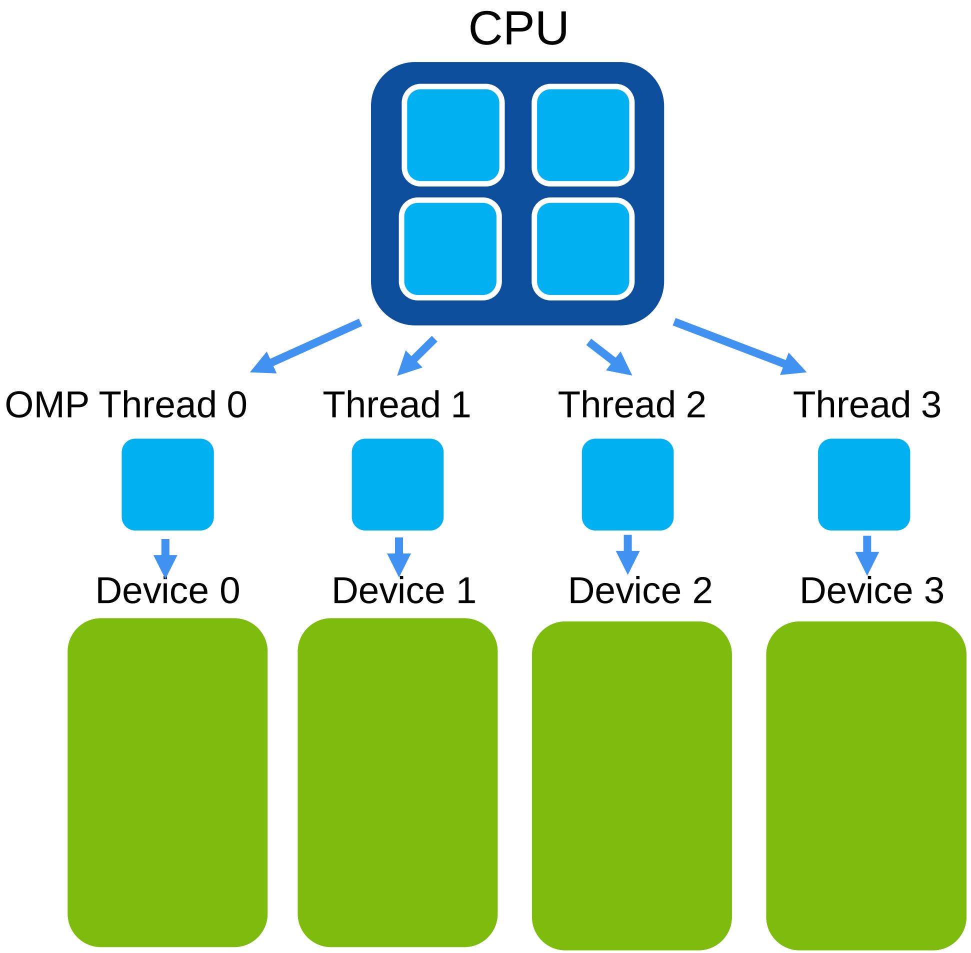
Loop over each device and call **wait**

```
int main() {
    int ndevices = acc_get_num_devices(acc_device_default);
    int block_size = WIDTH*HEIGHT/ndevices;
    ... // Allocate Data

    ... // Compute

    for(int device=0; device < ndevices; device++) {
        acc_set_device_num(device, acc_device_default);
        #pragma acc wait
    }

    ... // Deallocate Data
}
```

MULTI-DEVICE WITH OPENMP

MULTI-DEVICE WITH OPENMP

- We will assign each of our devices to an OpenMP thread
- This will allow us to utilize multiple devices, similar as before, without needing to use the async clause
- You are also able to assign multiple threads to the same device, which may be advantageous if you cannot fully utilize the device with a single thread



MULTI-DEVICE WITH OPENMP

```
ndevices =
  acc_get_num_devices(acc_device_default);

#pragma omp parallel num_threads(ndevices)
{
    int tid = omp_get_thread_num();
    acc_set_device_num(tid, acc_device_default);
    #pragma acc parallel loop
    for(int j = 0; j < size; j++) {
        // GPU code
    }
}
```

- We loop over all of our available devices like before
- However, now we break them up among OpenMP threads
- Each thread will utilize its own device
- No async is needed for the OpenMP variation, but it still helps (more on this later)

OPENMP MANDELBROT

Now we parallelize the outer loop with OpenMP

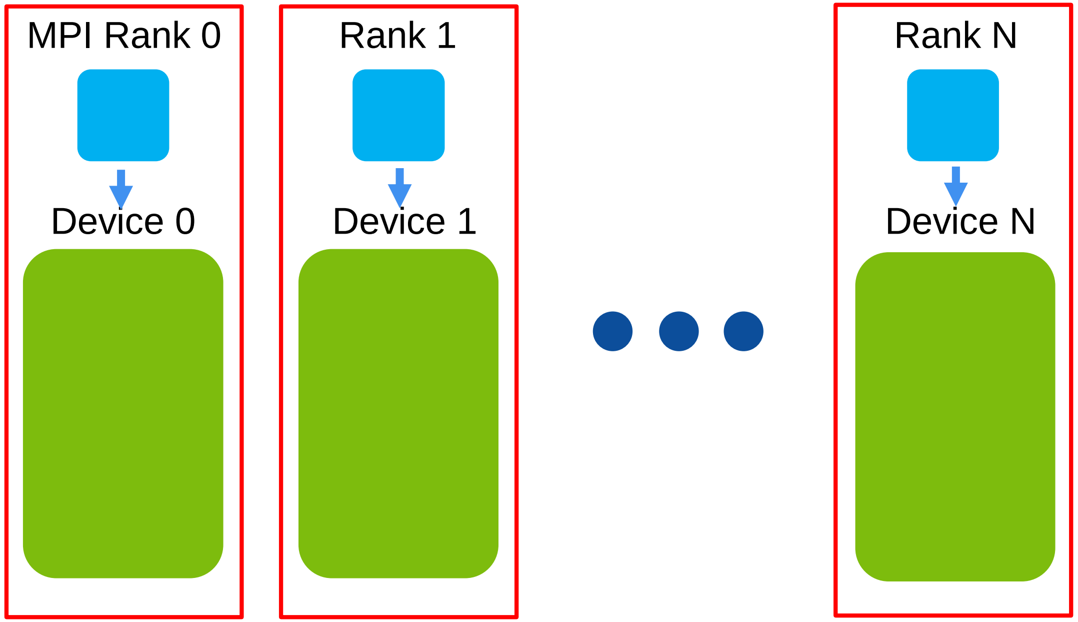
We assign each thread a different device

Since OpenMP is a shared memory model, all threads can copy their output to the same array

```
int main() {
     int ndevices = acc_get_num_devices(acc_device_default);
     int block_size = WIDTH*HEIGHT/ndevices;
#pragma omp parallel num_threads(ndevices)
{
     int tid = omp_get_thread_num();
     acc_set_device_num(tid, acc_device_default);
     int yStart = device*(HEIGHT/ndevices);
     int yEnd   = yStart+(HEIGHT/ndevices);
#pragma acc enter data create(image[yStart*WIDTH:block_size])
#pragma acc parallel loop
     for(int y=yStart;y<yEnd;y++) {
          for(int x=0;x<WIDTH;x++) {
               image[y*WIDTH+x]=mandelbrot(x,y);
          }
     }
#pragma acc update self(image[yStart*WIDTH:block_size])
#pragma acc exit data delete(image)
} // end omp parallel
}
```

MULTI-DEVICE WITH MPI

MULTI-DEVICE WITH MPI



- MPI is more difficult to implement, but is generally more flexible and scalable
- The last two programming models we covered did not behave differently from what we expect from OpenACC
- However, our MPI + OpenACC code will look more like a standard MPI code with some OpenACC "sprinkled in"

MULTI-DEVICE WITH MPI

```
MPI_Init(&argc,&argv);
int rank, nranks;
MPI_Comm_size(MPI_COMM_WORLD, &nranks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

acc_set_device_num(rank, acc_device_default);

MPI_Scatter(...);

#pragma acc parallel loop
for(int i = rank*N; i < rank*(N+1); i++)
    // Loop

MPI_Gather(...);
```

- By using MPI, we can have different bits of code run by different MPI ranks
- This means that each rank could run a different portion of a loop, or different loops entirely
- When using multiple devices, we can assign each MPI rank its own device
- These ranks could also share devices if the device is being under-utilized

OpenACC

MPI MANDELBROT

Because MPI uses distributed memory, each rank will compute its own *subimage*

Then at the end, all subimages will be gathered into a single *image*

```
int main(int argc, char** argv ) {
     MPI_Init(&argc,&argv);
     int rank, nranks;
     MPI_Comm_size(MPI_COMM_WORLD, &nranks);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     int block_size = WIDTH*HEIGHT/nranks;
     double *subimage = new double[block_size];
     ...
     acc_set_device_num(rank, acc_device_default);
#pragma acc parallel loop
     for(int y=0; y<HEIGHT/nranks; y++) {
          for(int x=0; x<WIDTH; x++) {
               subimage[y*WIDTH+x]=mandelbrot(x,y*(HEIGHT/rank));
          }
     }
#pragma acc update host(subimage[0:block_size])
     MPI_Gather(subimage, ..., image, ...);
}
```

MPI MANDELBROT

Each MPI rank uses a different device based on its rank

We will also alter the value of *y* based on rank, since each MPI rank needs to compute its own portion of the image

```
int main(int argc, char** argv ) {
     MPI_Init(&argc,&argv);
     int rank, nranks;
     MPI_Comm_size(MPI_COMM_WORLD, &nranks);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     int block_size = WIDTH*HEIGHT/nranks;
     double *subimage = new double[block_size];
     ...
     acc_set_device_num(rank, acc_device_default);
#pragma acc parallel loop
     for(int y=0; y<HEIGHT/nranks; y++) {
          for(int x=0; x<WIDTH; x++) {
               subimage[y*WIDTH+x]=mandelbrot(x,y*(HEIGHT/rank));
          }
     }
#pragma acc update host(subimage[0:block_size])
     MPI_Gather(subimage, ..., image, ...);
}
```

MPI DATA MOVEMENT BETWEEN DEVICES

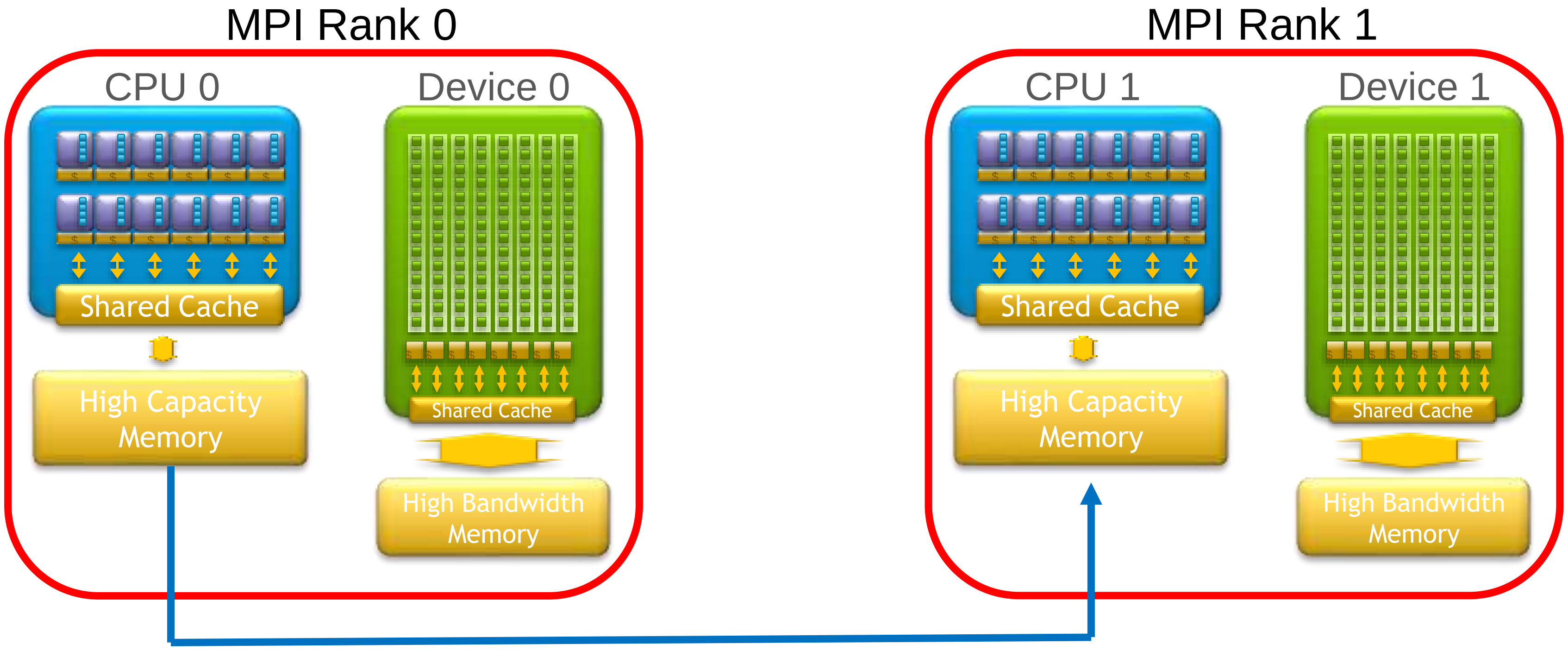
- MPI is also able to facilitate device-to-device data transfers using the *host_data* directive and *use_device* clause
- We simply need to pass device pointers into the MPI_Gather call, and it should move data device-to-device if the architecture is able to

We can use the OpenACC host_data/use_device to move data directly between devices

```
int main(int argc, char** argv ) {
    MPI_Init(&argc,&argv);
    int rank, nranks;
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int block_size = WIDTH*HEIGHT/nranks;
    double *image = new double[WIDTH*HEIGHT];
    double *subimage = new double[block_size];
    if(rank == 0) {
#pragma acc enter data create(image[0:WIDTH*HEIGHT])
    }
#pragma acc enter data create(subimage[0:block_size])

    ... // GPU Computation on subimage

#pragma acc host_data use_device(subimage, image)
    MPI_Gather(subimage, block_size, image, WIDTH*HEIGHT,
      MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

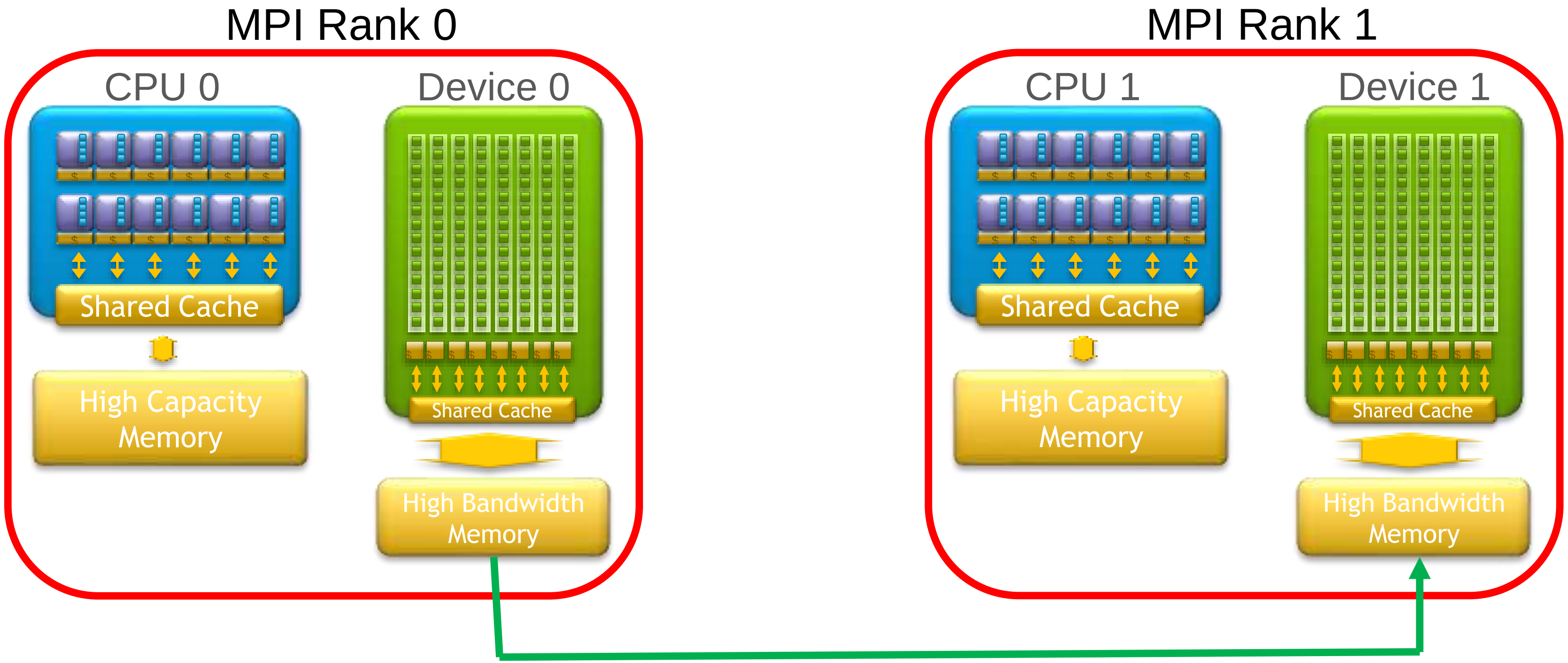
HOST DATA TRANSFER WITH MPI

```
MPI_Send( /* From Rank 0 -> Rank 1 */ );
```



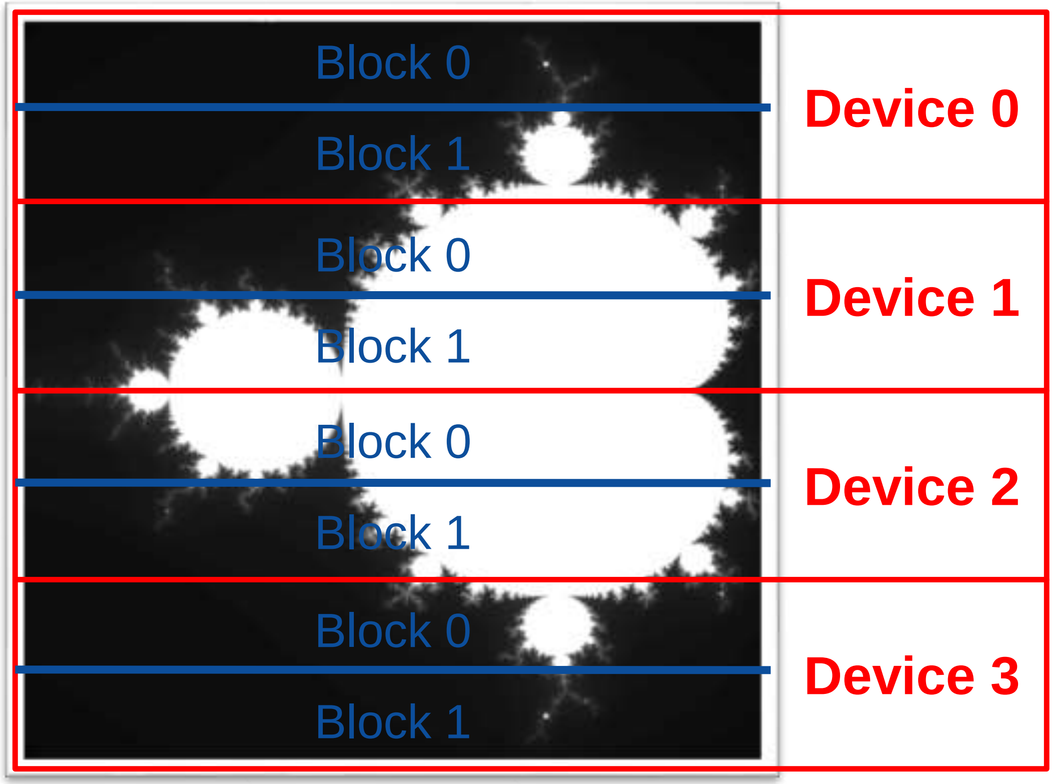
DEVICE DATA TRANSFER WITH MPI

```
#pragma acc host_data use_device(array)
MPI_Send( /* From Rank 0 -> Rank 1 */ );
```



MULTIDEVICE OPTIMIZED MANDELBROT

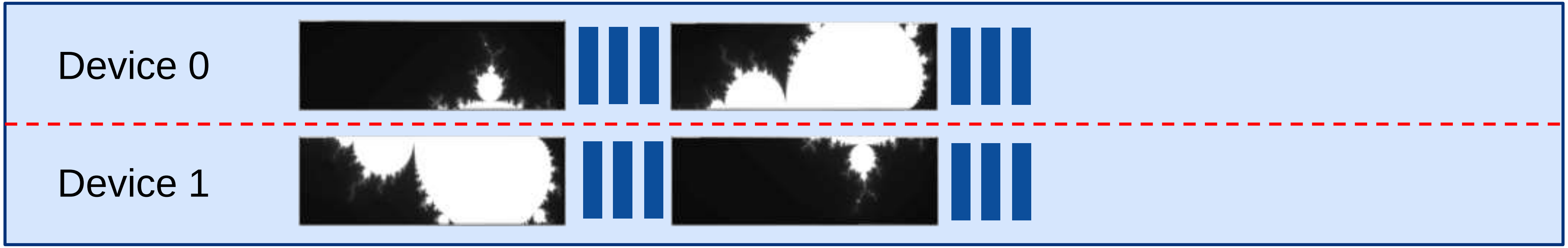
MULTIDEVICE OPTIMIZED MANDELBROT



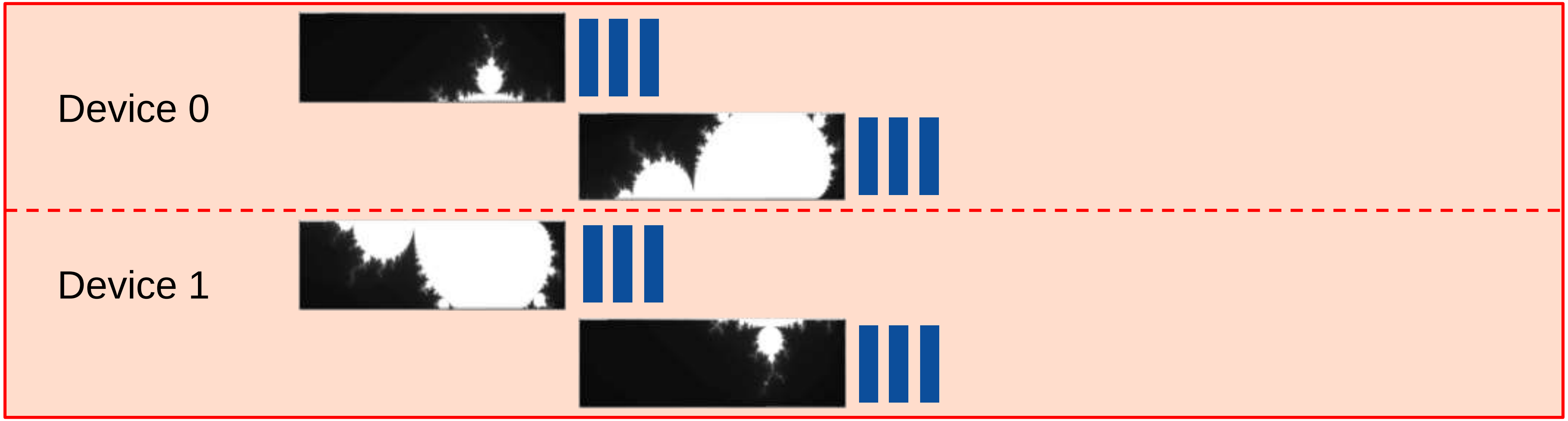
- As a final example, we want to reimplement the async optimizations from Module 7 into our multi-device setup
- We will use OpenMP to breakup the work among multiple devices
- And we will use OpenACC async to optimize the code further on each device

OUR GOAL

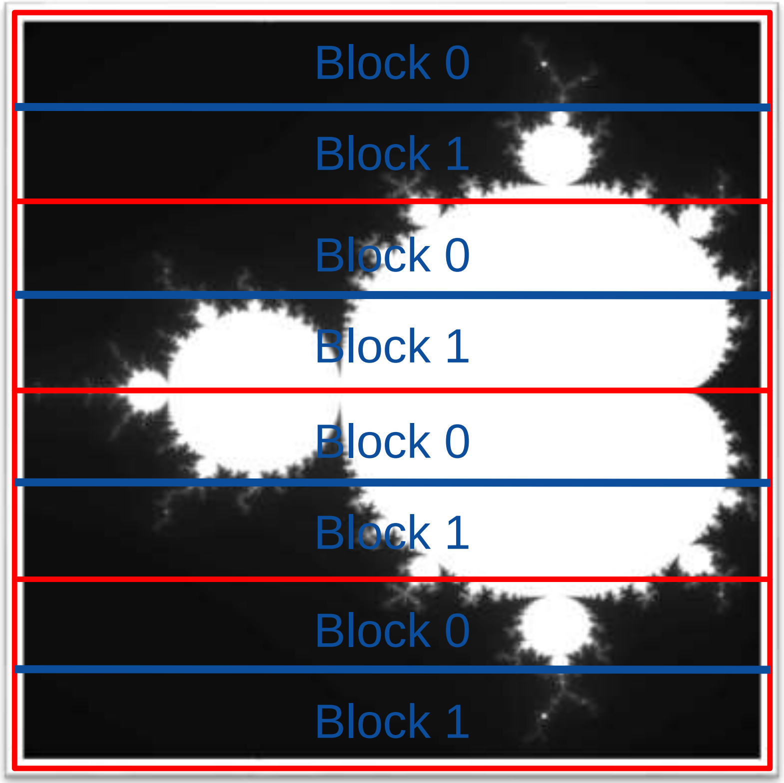
MULTI-DEVICE



MULTI-DEVICE + ASYNC



FINISHED MANELBROT



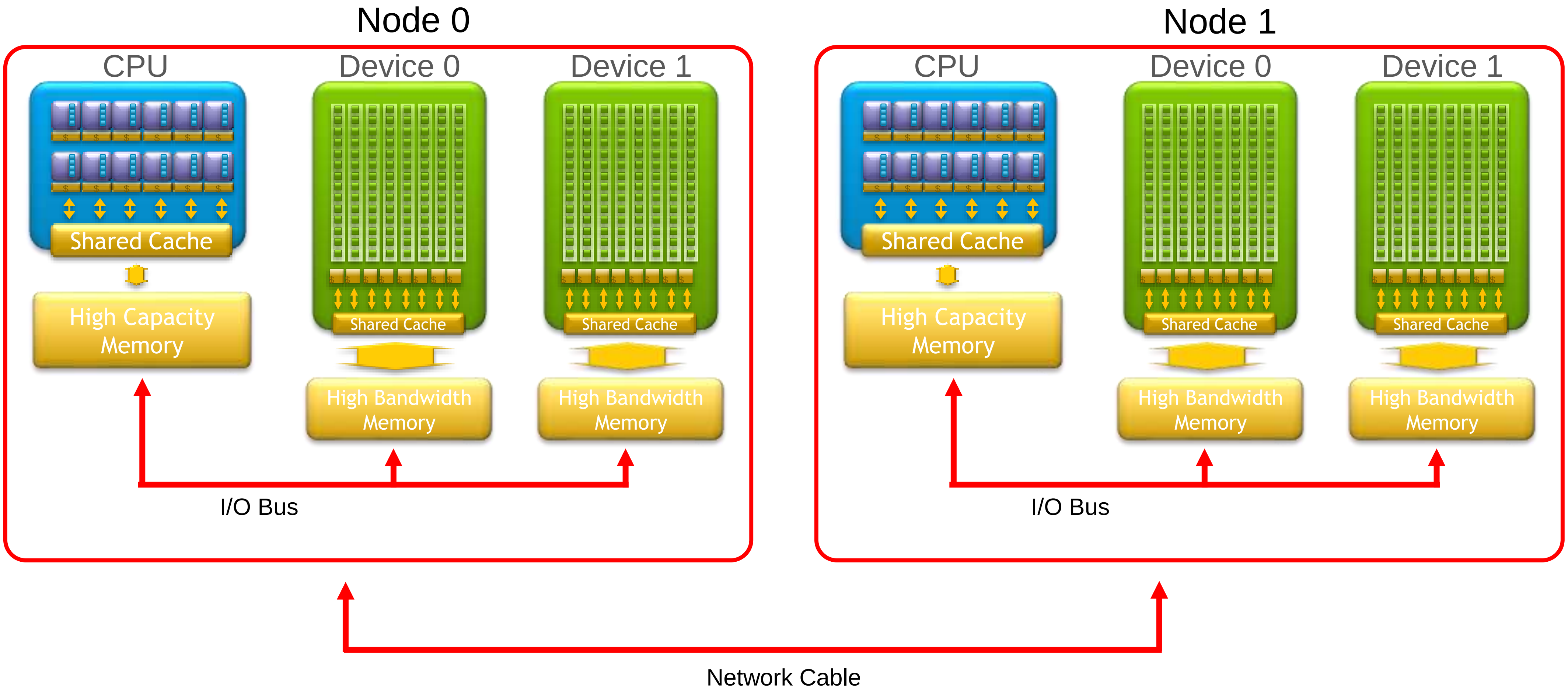

```
int main() {
    int per_device = WIDTH*HEIGHT/ndevices;
    int block_size = per_device/nblocks;
#pragma omp parallel num_threads(ndevices)
{
    int tid = omp_get_thread_num();
    for(int block=0; block<nblocks; block++) {
        int yStart = block*(HEIGHT/nblocks);
        int yEnd   = yStart+(HEIGHT/nblocks);
#pragma acc parallel loop async(block%2)
        for(int y=yStart;y<yEnd;y++) {
            for(int x=0;x<WIDTH;x++) {
                image[y*WIDTH+x]=mandelbrot(x,y);
            }
        }
#pragma acc update \
host(image[yStart*WIDTH:block_size]) async(block%2)
    }
#pragma acc wait
} // end omp parallel
}
```

LARGE-SCALE OPENACC

LARGE-SCALE OPENACC

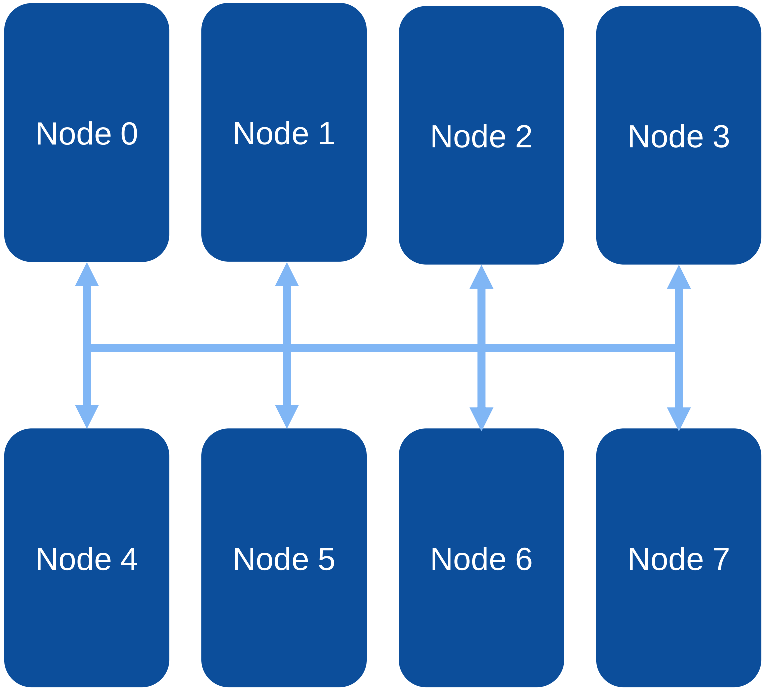
- This module we have focused on programming for a single machine (commonly called a *node*)
- Some computer applications are run across many nodes, sometimes upwards of hundreds to thousands of nodes
- On top of this, each node could have one or more accelerator device (such as a computer having multiple GPUs)
- With MPI + OpenACC, we can program for this large-scale, multi-node multi-device architecture

MULTIPLE NODES, MULTIPLE DEVICES



OpenACC

MULTIPLE NODES, MULTIPLE DEVICES



- In order to program this style of architecture:
- You would use MPI to manage node-to-node communication
- You would use OpenACC to accelerate code on the devices
- With a large architecture like this you will need a good understanding of MPI, but all of the MPI + OpenACC concepts discussed earlier would apply here

WRAP-UP

- Overall, for multidevice programming each method will have pros and cons
- Using OpenACC async and OpenMP + OpenACC are both good for utilizing one node with multiple devices
- Using MPI + OpenACC might be the most difficult to program, but will allow application scaling across a larger number of devices
- Depending on what is comfortable to you and your use case would decide which method to use

THANK YOU

OpenACC
More Science, Less Programming