CUDA C on Multiple GPUs (Ch. 11 of *CUDA By Example*)

- Systems containing multiple GPUs are becoming more common
  - `weathertop.stat.osu.edu` has 2 GPUs
- Naïvely, we would expect to double the speed if using 2 GPUs
- However, copying the same memory to each GPU can be time consuming
- Zero-copy memory speeds up copying to one GPU and portable pinned memory will allow us to do this on multiple GPUs

Zero-copy host memory

- Zero-copy lets us avoid making explicit copies of the data to and from the GPU
- Uses page-locked/pinned memory we learned about last week
- We tell the program that we intend to access the buffer from the GPU
- We also tell the program to allocate the buffer as write-combined
  - Inefficient if the CPU needs to read from the buffer
- We create a GPU pointer to the memory on the CPU
  - The pointers look like they are on the GPU, but they actually reside on the host
- Besides that, the kernel acts the same and no additional coding is needed

Comparison of zero-copy host memory

- **Originally**, we copied memory to the device using

```
a = (float*)malloc( size*sizeof(float) )
cudaMalloc( (void**)&dev_a, a, size*sizeof(float) )
cudaMemcpy( dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice )
```

- **Last week**, to allocate pinned memory

```
cudaHostAlloc((void**)&a, size*sizeof(float), cudaHostAllocDefault)
cudaMalloc( (void**)&dev_a, a, size*sizeof(float) )
cudaMemcpy( dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice )
```

- **Now**, we allocate zero-copy memory

```
cudaHostAlloc( (void**)&a,size*sizeof(float),
                cudaHostAllocWriteCombined | cudaHostAllocMapped )
cudaHostGetDevicePointer( dev_a, a, 0)
```

Performance of zero-copy host memory

- Like last week, each pinned allocation takes up physical memory, so be careful
- Zero-copy memory is not cached on the GPU. Do not use if memory gets read multiple times
- The book performed tests on two different systems and saw improvements of 35 to 45% running the dot-product example

Using Multiple GPUs

- Each GPU needs to be controlled by a different CPU thread
  - The book supplies code to make multi-threading easier
- The book recommends creating a data structure that provides space for input, output, and the GPU device ID

```
struct DataStruct {
    int     deviceID;
    int     size;
    float   *a;
    float   *b;
    float   returnValue;
};
```

- If you are using N GPUs, split the data N ways with different GPU device IDs (`data[0], data[1],..., data[N-1]`)
- For each data piece, start a thread and call a function that will execute the kernel for that data on the specified GPU

Portable pinned memory

- Portable pinned memory allows us to combine the speed-ups from zero-copy host memory and multiple GPUs
- To do so, we must be able to access the pinned memory from any GPU
- Problem: pinned memory can only **appear** pinned to a single CPU thread (the thread that allocated it)
  - Other threads will see the buffer as standard, pageable data
- A thread that did not allocated the pinned buffer will copy the data at pageable speeds (50% slower) or possibly crash
- The solution is to allocate the pinned memory as **portable**, meaning we allow any thread to view it as a pinned buffer

Technical details of portable pinned memory

- Before we allocate host memory, we have to set the (first) CUDA device on which we wish to run (`cudaSetDevice(0)`) (?)
- Memory management is very similar to what we did before

```
cudaHostAlloc( (void**)&a, N*sizeof(float),
                cudaHostAllocWriteCombined |
                cudaHostAllocPortable |
                cudaHostAllocMapped )
cudaHostGetDevicePointer( &dev_a, a, 0 )
```