Lecture 9

Optimizing memory copying between GPU and CPU

- Sometimes one can reduce or eliminate the time spent on GPU-CPU data copying if it is done in parallel (asynchronously) with host computations:

```
// On host:

// This memory copying will be asynchronous only in regards to the host code:
cudaMemcpyAsync (d_a, h_a, size, cudaMemcpyHostToDevice, 0);

// This host code will be executed in parallel with memory copying
host_computation ();
```

Optimizing memory copying between GPU and CPU (cont)

- One can also run memory transfer operation concurrently with another (opposite direction) memory transfer operation, or a kernel. For that, one has to create and use streams.
- Only works with pinned host memory

```
// This memory copying will be asynchronous in regards to the host and stream ID[1]:
cudaMemcpyAsync (d_a, h_a, size, cudaMemcpyHostToDevice, ID[0]);

// The kernel doesn't need d_a, and will run concurrently with the previous line:
kernel1 <<<N, M, 0, ID[1]>>> ();
```

Optimizing memory copying between GPU and CPU (cont)

- To save on memory copying overheads, one should try to bundle up multiple small transfers into one large one
- This can be conveniently achieved by creating a single structure, with the individual memory copying arguments becoming elements of the structure

Optimizing memory copying between GPU and CPU (cont)

```
// Host code:
cudaMemcpyToSymbol (d_A, &h_A, sizeof(h_A), 0, cudaMemcpyHostToDevice);
cudaMemcpyToSymbol (d_B, &h_B, sizeof(h_B), 0, cudaMemcpyHostToDevice);
cudaMemcpyToSymbol (d_C, &h_C, sizeof(h_C), 0, cudaMemcpyHostToDevice);
```

```
// Header file:
struct  my_struc {
  double A[1000];
  double B[2000];
  int C[1000];
};
__device__  struct  my_struc  d_struc;
struct  my_struc  h_struc;
// Host code:
cudaMemcpyToSymbol (d_struc, &h_struc, sizeof(h_struc), 0, cudaMemcpyHostToDevice);
```

Optimizing memory copying between GPU and CPU (cont)

- If you use dynamic memory allocation on host, you can usually accelerate copying to/from the device by using cudaMallocHost instead of malloc.
  - This will force the compiler to use page-locked memory for host allocations, which has much higher bandwidth to the device
  - Use this sparingly, as the performance can actually degrade when not enough of system memory is available for paging

```
// Host code:
float *h_A;

cudaMallocHost (&h_A,  N*sizeof(float));
```

Minimizing warp divergence

- The smallest independent execution unit in CUDA is a warp (a group of 32 consecutive threads in a block)
- Within a warp, execution is synchronous (that is, warp acts as a 32-way vector processor)
- Any flow control instruction (if, switch, do, for, while) acting on individual threads within a warp will result in warp divergence (with the different execution paths serialized), resulting in poor performance
- Warp divergence minimization is hence an important CUDA optimization step

Minimizing warp divergence (cont)

- Ideally, controlling conditions should be identical within a warp:

```
// On device:
__global__  void  MyKernel ()
{
int i = threadIdx.x + blockDim.x * blockIdx.x;
int warp_index = i / warpSize;  // Remains constant within a warp

if (d_A[warp_index] == 0)  // Identical execution path within a warp (no
divergence)
    do_one_thing (i);
else
    do_another_thing (i);
}
```

Minimizing warp divergence (cont)

- As warps can't span thread blocks, conditions which are only a function of block indexes result in non-divergent warps

```
// On device:
__global__  void  MyKernel ()
{
int i = threadIdx.x + blockDim.x * blockIdx.x;

if (d_A[blockIdx.x] == 0)  // No divergence, since warps can't span thread
blocks
    do_one_thing (i);
else
    do_another_thing (i);
}
```

Minimizing warp divergence (cont)

- More generally, making a condition to span at least a few consecutive warps results in acceptably low level of warp divergences (even when the condition is not always aligned with warp boundaries)

```
// On device:
__global__  void  MyKernel ()
{
int i = threadIdx.x + blockDim.x * blockIdx.x;
int cond_index = i / N_CONDITION;  // Is okay if N_CONDITION >~ 5*warpSize

if (d_A[cond_index] == 0)  // Only a fraction of warps will have divergences
    do_one_thing (i);
else
    do_another_thing (i);
}
```

Accuracy versus speed

- Situation with double precision speed in CUDA improved dramatically in the recent years, but it is still slower than single precision
  - The ratio was 1:8 for capability 1.3, and 1:2 for capability 2.0 (Fermi).
  - The ratio became 1:3 for Kepler, then back to 1:2 for Pascal, Volta, and Ampere.
- Use double precision only where it is absolutely necessary

Optimal kernel parameters

- Number of threads per block (BLOCK_SIZE): total range 1...1024; much better if multiples of 32; better still if multiples of 64.
- Number of threads per multiprocessor: at least 768 for capability 2.x to completely hide read-after-write register latency. That means >43,000 threads for P100*.
- Number of blocks in a kernel: at least equal to the number of multiprocessors (≥56 for P100s on Graham), to keep all multiprocessors busy.

Hands on exercise

- Primes: converting a serial code for the largest prime number search to CUDA

CUDA on multiple GPUs

Need multiple GPUs for:

- Problems which require more memory that is available on a single GPU
- Problems which take too long to compute on a single GPU
- number of approaches available
- the CUDA version you are using and Compute Capability of the GPU are important here - the more advanced, the more you can do with multiple GPUs
- good time to introduce a very useful feature of later versions of CUDA - Unified Virtual Addressing, or Unified Address Space

Unified Virtual Addressing

- Makes the separate memory of host and attached GPUs appear as a single region of memory
- Allows easier (for the programmer) memory access between host and GPUs, without requiring a cudaMemcpy operation in all cases
- Easy does not necessarily mean fast - the fundamental limitations of the bandwidth between host and GPU will still apply
- Nevertheless, in some cases this type of access will be faster, since computation and memory transfer will be overlapped by default
- Some of this functionality was present in older versions of CUDA via a more complicated, less convenient mechanism. We will not cover it.

UVA simplifies cudaMemcpy

- Can now use the argument cudaMemcpyDefault instead of cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost etc.
- CUDA will now automatically detect where the memory referred to by pointers supplied as cudaMemcpy arguments resides
- more convenient for the programmer, avoids errors
- IMPORTANT: for this to work for host memory, you must allocate it as pinned memory via CUDA (with cudaMallocHost). It will not work for memory allocated via malloc.
- remember to compile for arch 2.0 or higher

Zero copy memory access

- Unified address space means kernel can access host memory directly via a host pointer passed as argument to kernel
- The memory on host must be allocated as pinned memory for UVA to work
- Called “zero copy” because an explicit copy operation is not required
- There is still a cost to accessing host memory from GPU. If you need to do a lot of GPU computation on data, it’s better to move it to GPU memory
- The advantage of zero-copy is that computation and memory transfer can now be made to overlap automatically by CUDA*

Be careful with “Unified”

- certain operations permitted but not all
- you cannot access the GPU memory directly from host

```
...
cudaMalloc((void **) &y_0, memsize)   // allocated y_0 on GPU
for ( i = 0; i < n; i++) y_0[i] = 1.0; //try to modify y_0 from host, this will fail!
...
```

Exercise

- revisit SAXPY problem, now using UVA
- change CUDA memory copies to use the default direction keyword
- starting file is located on graham in:

  ~syam/CSE746/saxpy_uva

Exercise

- modify code so that the GPU kernel does its work in host memory
- compare performance

Possible multiple GPUs paradigms

- single host thread controlling multiple GPUs which are connected directly to the host via PCI bus. Thread can control only 1 GPU at a time, so it will be switching between them.
- multiple host (OpenMP) threads controlling multiple GPUs which are connected to the host via PCI bus. Could assign a single GPU to each thread.
- multiple MPI processes, each on node with some GPUs connected via PCI bus, nodes connected with network. Each MPI process could be assigned on GPU.
- mixtures of above (threads + MPI) also possible

Single host thread - multiple GPUs

- only one GPU can be controlled at a time
- program sets which GPU is controlled with
  cudaSetDevice(gpu_number);
  where gpu_number can be 0,1,... up (number of GPUs -1)
- after cudaSetDevice is called, all subsequent CUDA calls running on GPUs and kernels will run on GPU selected in gpu_number
- when programming, it is a good idea to add cudaSetDevice before every GPU call, to be sure which GPU it's executed on:

```
...
cudaSetDevice(gpu_number); saxpy_gpu<<<nBlocks, blockSize>>>(y_host, x_host, alpha, n);
cudaSetDevice(gpu_number); cudaDeviceSynchronize();
...
```

Multi-GPU synchronization

- cudaDeviceSynchronize() will only synchronize host with the currently set GPU
- if multiple GPUs are in use and all need to be synchronized, cudaDeviceSynchronize has to be called separately for each one

```
...
/* in this example have 2 GPUs which we need to synchronize */
cudaSetDevice(0); cudaDeviceSynchronize();
cudaSetDevice(1); cudaDeviceSynchronize();
...
```

Exercise

- modify code from 1b so that SAXPY operation is done on 2 GPUs
- simply have each GPU handle one half of the vector
- carefully modify the CUDA timing mechanisms so correct timing is obtained on each GPU
- compare performance with code from exercise 1

Multiple GPUs with multiple threads

- can use OpenMP threads, and assign a GPU to each thread

```
/* compile with:
nvcc -Xcompiler -fopenmp -arch=sm_60 -O2 code.cu -o code.x
run with
OMP_NUM_THREADS=2 ./code.x
...
#include <omp.h>
...
#pragma omp parallel private(tid,error)
 {
  tid = omp_get_thread_num();
  cudaSetDevice(tid);
...
  }
```

Multiple host threads accessing same GPU?

- The common approach is to have a single thread of a process assigned to access a particular GPU
- One way different threads can access different GPUs is if each thread creates and uses its own stream.
- Starting from Kepler (capability 3.5), even simpler alternative exists: Multi-Process Service (also called Hyper-Q). This approach allows to share one GPU between multiple CPU threads or processes without the need to make any changes to the code. More about it later.

Multiple GPUs via MPI - detection

- With MPI approach programmer has to be more careful which GPU MPI process binds to, since multiple MPI processes could be assigned to the same node

```
/*  compile:
module unload intel openmpi
module load gcc/4.8.2 openmpi/gcc/1.8.3
nvcc -I/opt/sharcnet/openmpi/1.8.3/gcc/include/ -L/opt/sharcnet/openmpi/1.8.3/gcc/lib/ -lmpi  test_mpi.cu -o test.x
run: mpirun -np 2 -o test.x
*/
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "cuda.h"
int main(int argc, char *argv[]){
      int numprocs, rank,namelen;
      int devcount;
      char processor_name[MPI_MAX_PROCESSOR_NAME];
      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Get_processor_name(processor_name, &namelen);
      cudaGetDeviceCount(&devcount);
      printf("Process %d of  %d running on node %s  is detecting %d GPU devices \
n",rank,numprocs,processor_name,devcount);
      MPI_Finalize();return 0;}
```