CSE 746 - Parallel and High Performance Computing Lecture 10 - CUDA on multiple GPUs

Pawel Pomorski, *HPC Software Analyst*

SHARCNET, University of Waterloo

ppomorsk@sharcnet.ca

http://ppomorsk.sharcnet.ca/

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 1 a

- revisit SAXPY problem, now using UVA
- change CUDA memory copies to use the default direction keyword
- starting file is from Lecture 2 code, located in:

  /home/ppomorsk/CSE746_lec10/saxpy_uva

Exercise 1 b

- 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 synchornize */
cudaSetDevice(0); cudaDeviceSynchronize();
cudaSetDevice(1); cudaDeviceSynchronize();
...
```

Exercise 2

- 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_20 -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);
...
   }
```

Exercise 3

- modify code from exercise 2 to use 2 OpenMP threads
- you will only need to call cudaSetDevice once, at the beginning of parallel region
- to keep things simple, you do not have to use CUDA timing calls in this case

multiple host threads accessing same GPU?

- not typically done, the general approach is to have a single thread of a process assigned to access a particular GPU
- if unavoidable, should have each thread access using its own stream to access the GPU, this is anyway required for the threads to run in parallel with each other

Peer-to-Peer (P2P) transfer between GPUs

- in UVA model, GPU to GPU transfer is possible in code, but it actually still goes through host memory
- it is possible to enable transfer directly between GPUs over the PCI bus
- PCI bus is still slow, but you gain a little bit of time if you can avoid the host memory during the transfer
- If P2P is not available, program will fall back to normal copy
- To enable peer transfer between gpu0 and gpu1: cudaSetDevice(gpu0); cudaDeviceEnablePeerAccess(gpu1, 0);
- second argument must always be set to zero, and is currently meaningless (reserved for future use)

Peer-to-Peer (P2P) transfer between GPUs

- it is good to check if P2P is enabled. Even on a single node particular GPUs may not be on same PCI bus, so P2P not available

```
printf("\nChecking GPU(s) for support of peer to peer memory access...\n");
 int can_access_peer_0_1, can_access_peer_1_0;
 cudaDeviceCanAccessPeer(&can_access_peer_0_1, 0, 1);
 cudaDeviceCanAccessPeer(&can_access_peer_1_0, 1, 0);
 printf("%d %d \n",can_access_peer_0_1, can_access_peer_1_0);

 if(can_access_peer_0_1==1 && can_access_peer_1_0==1){
    printf("peer access possible between 0 and 1 \n");
 }
 else{
    printf("no peer access possible \n");
 }

 printf("Enabling peer access between GPU%d and GPU%d...\n", 0, 1);
 cudaSetDevice(0);cudaDeviceEnablePeerAccess(1, 0);
 cudaSetDevice(1);cudaDeviceEnablePeerAccess(0, 0);
```

Exercise 4

- write a program which copies memory between 2 GPUs
- compare performance with peer transfer enabled and without
- start code in: /home/ppomorsk/CSE746_lec10/p2p_transfer

Multiple GPUs via MPI

- each MPI process is independent, and can be single and multithreaded
- for each process, we have same binding mechanisms to GPU
- can run on one node where all GPUs are visible to all processes, or on multiple nodes where the MPI process only sees GPUs available on its node
- MPI and CUDA code can be combined in one source file

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;}
```

Multiple GPUs via MPI

- with UVA and other advanced features, can use CUDA allocated arrays in MPI calls directly (this did not work in the past)
- if two processes reside on same node, CUDA-aware MPI should be able to take advantage of P2P transfer between them

Exercise 5

- write program testing speed of memory transfer between 2 GPUs, each attached to one MPI process
- start code in:
  /home/ppomorsk/CSE746_lec10/mpi_cuda
  has host to host transfer MPI already implemented
- implement
  gpu0 -> host0 -> host1 -> gpu1
- then implement
  gpu0 -> gpu1
- pure MPI code (no CUDA) is provided also for testing purposes