Section 1.2.3: Lab Exercises for GPU Module

For all exercises in this document, you need to login on a machine with a GPU card. You can login to one of the ODU machines with GPU card as outlined below. You need to use your MIDAS ID and Password to login.

# Log in at ODU machine with GPU cards

1. Login to the “Turing” cluster with SSH to “turing.hpc.odu.edu”
2. Use the “gpu-login” script (invokes SGE to allocate an interactive session in a GPU node):

gpu-login

This script is in your path at login.

1. Get a list of your loaded modules in the compute node:

module list

The output may be blank.

1. Load the NVIDIA CUDA Toolkit module (includes “nvcc”):

module load /cm/shared/modulefiles/cuda/cuda70/toolkit/7.0

1. You can confirm that you have access to CUDA Toolkit by running nvcc --version from a command line. You will see something like this:

*[mzubair@appro-006 ~]$ nvcc --version*

*nvcc: NVIDIA (R) Cuda compiler driver*

*Copyright (c) 2005-2015 NVIDIA Corporation*

*Built on Mon\_Feb\_16\_22:59:02\_CST\_2015*

*Cuda compilation tools, release 7.0, V7.0.27*

*[mzubair@appro-006 ~]$*

NOTE: Some of the exercises here are adapted from exercises provided by Jared Hoberock of NVIDIA which is licensed under the Apache License version 2.0

# Exercise 1.2.3a:

To make sure everything is set up correctly, let's compile and run a trivial CUDA program to ensure all the tools work together correctly.

Step 1. Type in the following program and save it in a file ‘test\_run.cu’:

#include <stdio.h>

\_\_global\_\_ void foo()

{

}

int main()

{

foo<<<1,1>>>();

printf("CUDA error: %s\n", cudaGetErrorString(cudaGetLastError()));

return 0;

}

Step 2. Compile your program.

*[mzubair@appro-006 ~/CS724]$ nvcc test\_run.cu -arch=sm\_35 -o test\_run*

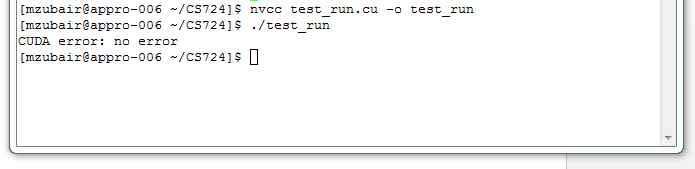
Step 3. Run your program, and you should see the following output if everything is working OK.

*[mzubair@appro-006 ~/CS724]$ ./test\_run*

*CUDA error: no error*

# Submission

Please submit a screen shot of your console that shows you were able to successfully compile and execute the test\_run program (see sample below).



# Exercise 1.2.3b:

Step 1. Type in the following program and save it in a file ‘hello\_world.cu’:

#include <stdio.h>

\_\_global\_\_ void helloCUDA()

{

printf("Hello from thread %d\n", threadIdx.x);

}

int main()

{

helloCUDA<<<1, 10>>>();

cudaDeviceReset();

}

Step 2. Compile your program.

*[mzubair@appro-006 ~/CS724]$ nvcc -arch=sm\_35 hello\_world.cu -o hello\_world*

Step 3. Run your program.

*[mzubair@appro-006 ~/CS724]$ ./hello\_world*

*Hello from thread 0*

*Hello from thread 1*

*Hello from thread 2*

*Hello from thread 3*

*Hello from thread 4*

*Hello from thread 5*

*Hello from thread 6*

*Hello from thread 7*

*Hello from thread 8*

*Hello from thread 9*

# Submission

Please submit a screen shot of your console that shows you were able to successfully compile and execute the program.

# Exercise 1.2.3c:

The code in the file cuda\_memory\_model.cu (available under the resources) introduces CUDA's heterogeneous memory model by demonstrating the difference between the host and device memory spaces. The program, which is executed by the serial CPU, or host, allocates and interacts with host memory using functions which should be familiar to C programmers. Similarly, the host uses analogous system calls, such as cudaMalloc and cudaFree to manage device space memory.

Please save the source code (cuda\_memory\_model.cu ) along with the make file in a folder. Compile and run your program using make command.

make

nvcc -o cuda\_memory\_model -arch=sm\_35 cuda\_memory\_model.cu

[mzubair@appro-006 Lab03]$ ./cuda\_memory\_model

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

The result is a sequence of zeros, as expected. Note that in order to access the device space result, the host was required to explicitly copy the array to host memory. Even though the cudaMalloc system call returns what appears to be a normal pointer, the memory it references is unavailable to the host. Attempts by the host to dereference it will likely yield a segmentation fault. Similarly, attempts by the device to dereference a host pointer will also meet with disaster.

# Submission

Please submit a screen shot of your console that shows you were able to successfully compile and execute the program.

# Exercise 1.2.3d:

\_\_global\_\_ functions expose a particular form of parallel computation called data parallelism. The basic idea of data parallelism is to distribute a large task composed of many similar but independent pieces across a set of computational resources. In CUDA, the task to be performed is described by a \_\_global\_\_ function, and the computational resources are CUDA threads. Individual threads work on individual subtasks, and since they are independent, these subtasks can be performed in parallel.

Consider the data parallel task of filling an array with a particular value. We have almost all the tools we need to implement this operation with a \_\_global\_\_ function in CUDA: we know how to allocate device memory, and we know how to write \_\_global\_\_ functions and instantiate CUDA threads which execute them. The missing piece is the mapping of subtasks (setting a particular array element to the given value) to particular CUDA threads. In order to map each particular CUDA thread to a particular subtask, CUDA provides built-in variables which uniquely identify each thread in its grid of thread blocks. To see how to use them, let's look at some code which fills a device array with the value 7.





The variables of interest are used inside the \_\_global\_\_ function, kernel. threadIdx.x uniquely identifies each CUDA thread within its thread block. Similarly, blockIdx.x uniquely identifies each thread block within the grid at large. These variables are combined with blockDim.x, which encodes the number of threads in each thread block, to yield an index which uniquely identifies each CUDA thread within the entire grid:

int index = blockIdx.x \* blockDim.x + threadIdx.x;

With this index, it's easy to ensure each CUDA thread has a 1-to-1 mapping to a unique array element if we've been careful to launch exactly as many CUDA threads as we have array locations to write.

In fact, when expressed this way, it's easy to recognize the relationship between the body of a \_\_global\_\_ function and the body of a traditional C for loop:

for(int index = 0; index < num\_elements; ++index)

{

array[index] = 7;

}

The crucial difference is that each for loop iteration executes sequentially while a \_\_global\_\_ function executes in parallel. You might have noticed that in this example, we passed an argument to our \_\_global\_\_ function, specifically, a pointer to the device array we wished to fill. \_\_global\_\_ functions are much like regular C functions in this regard -- they may accept parameters, as long as they are not C++ references. On the other hand, one important way that \_\_global\_\_ functions differ from normal C functions is that they must return void.

Please save the source code (global\_functions.cu) available on the resource page. Compile and run your program using make command.

*nvcc -o global\_functions -arch=sm\_35 global\_functions .cu*

*[mzubair@appro-006 Lab04]$ ./global\_functions*

*7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 …….*

# Submission

Please submit a screen shot of your console that shows you were able to successfully compile and execute the program.

# Exercise 1.2.3e:

Multidimensional Kernel Launch

The following code listing demonstrates an example of a 2D kernel launch, and shows how to map two dimensional thread and block indices to the physical one dimensional layout of device memory.

int main(void)

{

int num\_elements\_x = 16;

int num\_elements\_y = 16;

int num\_bytes = num\_elements\_x \* num\_elements\_y \* sizeof(int);

int \*device\_array = 0;

int \*host\_array = 0;

// malloc a host array

host\_array = (int\*)malloc(num\_bytes);

// cudaMalloc a device array

cudaMalloc((void\*\*)&device\_array, num\_bytes);

// if either memory allocation failed, report an error message

if(host\_array == 0 || device\_array == 0)

{

printf("couldn't allocate memory\n");

return 1;

}

// choose a two dimensional launch configuration

// use the dim3 type when launches are not one dimensional

// create 4x4 thread blocks

dim3 block\_size;

block\_size.x = 4;

block\_size.y = 4;

// configure a two dimensional grid as well

dim3 grid\_size;

grid\_size.x = num\_elements\_x / block\_size.x;

grid\_size.y = num\_elements\_y / block\_size.y;

// grid\_size & block\_size are passed as arguments to the

// triple chevrons as usual

kernel<<<grid\_size,block\_size>>>(device\_array);

// download and inspect the result on the host:

cudaMemcpy(host\_array, device\_array, num\_bytes, cudaMemcpyDeviceToHost);

// print out the result element by element

for(int row = 0; row < num\_elements\_y; ++row)

{

for(int col = 0; col < num\_elements\_x; ++col)

{

printf("%2d ", host\_array[row \* num\_elements\_x + col]);

}

printf("\n");

}

printf("\n");

// deallocate memory

free(host\_array);

cudaFree(device\_array);

}

The host code of this example uses a type we haven't seen before: dim3. In order to configure a two-dimensional grid, we pass values of this type to the triple chevrons instead of the plain integer arguments we've used before for one-dimensional kernels. dim3 is a simple vector type native to CUDA, and you can assume its definition looks something like:

struct dim3

{

unsigned int x, y, z;

...

};

You'll note that our example didn't set either of block\_size.z or grid\_size.z. By default, they are initialized to 1. Though the dimensionality of our problem is 2D, notice we configured the launch using arithmetic analogous to our previous 1D examples. First, we described the total size of the data parallel task at hand:

int num\_elements\_x = 16;

int num\_elements\_y = 16;

Next, we decided how to partition the task into blocks of threads:

dim3 block\_size;

block\_size.x = 4;

block\_size.y = 4;

And finally, we decided how many blocks of threads were needed to completely cover the size of the problem:

dim3 grid\_size;

grid\_size.x = num\_elements\_x / block\_size.x;

grid\_size.y = num\_elements\_y / block\_size.y;

Our \_\_global\_\_ function, kernel similarly performs arithmetic analogous to our previous examples to uniquely identify each thread involved in the computation at large. The built-in variables threadIdx, blockIdx, and blockDim are munged together to provide a unique, global index. This time, we do it in 2D:

int index\_x = blockIdx.x \* blockDim.x + threadIdx.x;

int index\_y = blockIdx.y \* blockDim.y + threadIdx.y;

This time, we used a new built-in variable we haven't seen before, gridDim, which encodes the number of thread blocks in each dimension. We used it to map the 2D global indices we computed, index\_x and index\_y, to a single, linear 1D index:

int grid\_width = gridDim.x \* blockDim.x;

int index = index\_y \* grid\_width + index\_x;

This was necessary in order to assign each individual CUDA thread to a unique element of array. The actual work our kernel does is trivial. Similarly to how we assigned a unique 1D index to each CUDA thread, we used blockIdx and gridDim to assign each thread block a unique index, which we wrote out to array.

To test the program, download the source code file, two\_dimensional\_kernel\_launch.cu, available on the resource page. Compile and run the program. The output of the program indicates the 2D structure of threads.

*[mzubair@appro-006 Lab05]$ nvcc -o two\_dimensional\_kernel\_launch. -arch=sm\_35 two\_dimensional\_kernel\_launch.cu*

*[mzubair@appro-006 Lab05]$ ./two\_dimensional\_kernel\_launch.*

*0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3*

*0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3*

*0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3*

*0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3*

*4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7*

*4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7*

*4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7*

*4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7*

*……*

# Submission

Please submit a screen shot of your console that shows you were able to successfully compile and execute the program.

# Exercise 1.2.3f:

\_\_device\_\_ Functions

The \_\_device\_\_ keyword lets us mark functions as callable from threads executing on the device. The syntax is like the \_\_global\_\_ keyword: we just prepend it to the function signature:

\_\_device\_\_ float my\_device\_function(float x)

{

return x + 1;

}

Though \_\_device\_\_ functions are similar to \_\_global\_\_ functions in that they are executed by device threads, they actually behave more like normal C functions. Unlike \_\_global\_\_ functions, \_\_device\_\_ functions cannot be configured (no <<<B,T>>> needed) and aren't subject to any special restrictions on the types of their parameters or results. Host code isn't allowed to call \_\_device\_\_ functions directly -- if we want access to the functionality in a \_\_device\_\_ function, we need to write a \_\_global\_\_ function to call it for us!

As you might expect, \_\_device\_\_ functions can call other functions decorated with \_\_device\_\_:

\_\_device\_\_ float my\_second\_device\_function(float y)

{

return my\_device\_function(y) / 2;

}

As long as they don't call themselves:

\_\_device\_\_ int my\_illegal\_recursive\_device\_function(int x)

{

if(x == 0) return 1;

return x \* my\_illegal\_recursive\_device\_function(x-1);

}

The following code listing shows how we might use \_\_device\_\_ functions to package up various bits of code when developing a CUDA kernel.

// This example introduces \_\_device\_\_ functions, which are special functions

// which may be called from code executing on the device.

#include <stdlib.h>

#include <stdio.h>

// \_\_device\_\_ functions may only be called from \_\_global\_\_ functions or other

// \_\_device\_\_ functions. Unlike \_\_global\_\_ functions, \_\_device\_\_ functions are

// not configured, and have no restriction on return type.

\_\_device\_\_ int get\_constant(void)

{

// just return 7

return 7;

}

\_\_device\_\_ int get\_block\_index(void)

{

// return the index of the current thread's block

return blockIdx.x;

}

\_\_device\_\_ int get\_thread\_index(void)

{

// return the index of the current thread within its block

return threadIdx.x;

}

\_\_device\_\_ int get\_global\_index(void)

{

// return the index of the current thread across the entire grid launch

return blockIdx.x \* blockDim.x + threadIdx.x;

}

// kernel1 returns the result of calling the \_\_device\_\_ function return\_constant():

\_\_global\_\_ void kernel1(int \*array)

{

int index = get\_global\_index();

array[index] = get\_constant();

}

// kernel2 returns the result of calling the \_\_device\_\_ function return\_block\_index():

\_\_global\_\_ void kernel2(int \*array)

{

int index = get\_global\_index();

array[index] = get\_block\_index();

}

// kernel3 returns the result of calling the \_\_device\_\_ function return\_thread\_index():

\_\_global\_\_ void kernel3(int \*array)

{

int index = get\_global\_index();

array[index] = get\_thread\_index();

}

// kernel4 returns the result of calling the \_\_device\_\_ function return\_thread\_index():

\_\_global\_\_ void kernel4(int \*array)

{

int index = get\_global\_index();

array[index] = get\_global\_index();

}

int main(void)

{

int num\_elements = 256;

int num\_bytes = num\_elements \* sizeof(int);

int \*device\_array = 0;

int \*host\_array = 0;

// malloc a host array

host\_array = (int\*)malloc(num\_bytes);

// cudaMalloc a device array

cudaMalloc((void\*\*)&device\_array, num\_bytes);

// if either memory allocation failed, report an error message

if(host\_array == 0 || device\_array == 0)

{

printf("couldn't allocate memory\n");

return 1;

}

// choose a launch configuration

int block\_size = 128;

int grid\_size = num\_elements / block\_size;

// launch each kernel and print out the results

kernel1<<<grid\_size,block\_size>>>(device\_array);

cudaMemcpy(host\_array, device\_array, num\_bytes, cudaMemcpyDeviceToHost);

printf("kernel1 results:\n");

for(int i=0; i < num\_elements; ++i)

{

printf("%d ", host\_array[i]);

}

printf("\n\n");

kernel2<<<grid\_size,block\_size>>>(device\_array);

cudaMemcpy(host\_array, device\_array, num\_bytes, cudaMemcpyDeviceToHost);

printf("kernel2 results:\n");

for(int i=0; i < num\_elements; ++i)

{

printf("%d ", host\_array[i]);

}

printf("\n\n");

kernel3<<<grid\_size,block\_size>>>(device\_array);

cudaMemcpy(host\_array, device\_array, num\_bytes, cudaMemcpyDeviceToHost);

printf("kernel3 results:\n");

for(int i=0; i < num\_elements; ++i)

{

printf("%d ", host\_array[i]);

}

printf("\n\n");

kernel4<<<grid\_size,block\_size>>>(device\_array);

cudaMemcpy(host\_array, device\_array, num\_bytes, cudaMemcpyDeviceToHost);

printf("kernel4 results:\n");

for(int i=0; i < num\_elements; ++i)

{

printf("%d ", host\_array[i]);

}

printf("\n\n");

// deallocate memory

free(host\_array);

cudaFree(device\_array);

}

This is the first time we've included more than one \_\_global\_\_ function in our example. This is fine -- they're just like C functions in this way. Note also that both kernel1 and kernel2 can call any of the \_\_device\_\_ functions they can "see". The usual C/C++ scoping rules apply.

For kernel1, we've essentially taken our \_\_global\_\_ function from a previous section and refactored it into \_\_device\_\_ functions get\_global\_index and get\_constant. get\_constant isn't so useful, but get\_global\_index encapsulates the tedious calculation of each CUDA thread's global index in the grid. Rather than repeating ourself, both kernels can simply call the \_\_device\_\_ function. Note that get\_global\_index uses the built-in variables blockIdx, blockDim, and threadIdx, which are available to \_\_device\_\_ functions as they are to \_\_global\_\_ functions.

To test the program, download the source code file, device\_functions.cu, available on the resource page. Compile and run the program.

*[mzubair@appro-006 Lab06]$ nvcc -o device\_functions.cu -arch=sm\_35 device\_functions.cu*

*[mzubair@appro-006 Lab06]$ ./device\_functions.cu*

*kernel1 results:*

*7 7 7 7 7 7 7 7…*

*kernel2 results:*

*0 0 0 0 0 0 0 0…*

# Submission

Please submit a screen shot of your console that shows you were able to successfully compile and execute the program.

# Exercise 1.2.3g:

In this exercise we will compile and run the vector addition program discussed in the module overview video. Download the source code file, vector\_addition.cu.cu, available on the resource page. Compile and run the program.

*[mzubair@appro-006 Lab07]$ nvcc -o vector\_addition -arch=sm\_35 vector\_addition.cu*

*[mzubair@appro-006 Lab07]$ ./vector\_addition*

*result 0: 0.0 + 0.8 = 0.8*

*result 1: 1.0 + 0.4 = 1.4*

*result 2: 2.0 + 0.8 = 2.8*

*result 3: 3.0 + 0.8 = 3.8*

*……*

# Submission

Please submit a screen shot of your console that shows you were able to successfully compile and execute the program.