GPU Computing Workshop: Hands on exercises (Day 1)

Dr Paul Richmond (University of Sheffield)

Accessing Amazon GPUs and Setup

Navigate to the NVlabs website and begin the CUDA Training course. You will be provided log in details for an Amazon AWS virtual machine instance which has a Kepler series GPU. Log into the instance using ssh (via Putty on windows) with the log in details provided to you.

Get the Starting code from github by cloning the master branch of the handoncuda_day1 repository from the AccleratedComputing github page. E.g.

```
$git clone
https://github.com/AcceleratedComputing/handsoncuda_day1.git
```

This will check out all the starting code for you to work with.

A Simple Example

For our first CUDA program we are going to implement a simple hello world application. From the code the you checked out of github view the `helloworld.cu` file. You can use `nano` e.g.

```
$nano helloworld.cu
```

Compile the code using nvcc with eth following command.

```
$nvcc helloworld.cu –o helloworld
```

Execute the `helloworld` application and you should get the following output

```
Hello World from Thread 0
Hello World from Thread 1
Hello World from Thread 2
Hello World from Thread 3
Hello World from Thread 4
Hello World from Thread 5
Hello World from Thread 6
Hello World from Thread 7
Hello World from Thread 8
Hello World from Thread 9
```

Try modifying the grid and block dimensions to see how the thread index changes. Try using more than one block and add the block index (from `blockIdx.x`) to the `printf` statement. If you want to use 2D or 3D blocks then use a dim3 variable to define the grid and block size. E.g.

```
dim3 grid(2, 2, 1);
dim3 block(3,3, 1);
```

You can now launch your kernel using these variables.

```
helloworld<<<grid, block>>>();
```

To view the `y` and `z` index of the thread or the block use the `y` and `z` member variables of the `threadIdx` or `blockIdx` dims.

Exercise 01

Exercise 1 requires that we de-cipher some encrypted text. The text provided in the file *encrypted01.bin* has been encrypted by using an affine cipher. The affine cypher is a very simple type of monoalphabetic substitution cypher where each numerical character of the alphabet is encrypted using a mathematical function. The encryption function is defined as;

$$E(x) = (Ax + B)\ mod\ M$$

Where A and B are keys of the cypher, mod is the modulo operation and A and M are co-prime. For this exercise the value of A is `15`, B is `27` and M is `128` (the size of the ASCII alphabet). The affine decryption function is defined as

$$D(x) = A^{-1}(x - B)\ mod\ M$$

Where A^{-1} is the modular multiplicative inverse of A modulo M. For this exercise A^{-1} has a value of `111`. *Note: The mod operation is not the same as the remainder operator (`%`) for negative numbers. A suitable mod function has been provided for the example.*

As each of the encrypted character values are independent we can use the GPU to decrypt them in parallel. To do this we will launch a thread for each of the encrypted character values and use a kernel function to perform the decryption. Starting from the code provided in `exercise01.cu`, complete the following;

1.1 Modify the `modulo` function so that it can be called on the device by the `affine_decrypt` kernel.
1.2 Implement the decryption kernel for a single block of threads with an `x` dimension of `N (1024)`. The function should store the result in `d_output`. You can define the inverse modulus `A`, `B` and `M` using a pre-processor definition.
1.3 Allocate some memory on the device for the input (`d_input`) and output (`d_output`).
1.4 Copy the host input values in `h_input` to the device memory `d_input`.
1.5 Configure a single block of `N` threads and launch the `affine_decrypt` kernel.
1.6 Copy the device output values in `d_output` to the host memory `h_output`.
1.7 Compile and execute your program. If you have performed the exercise correctly, you should decrypt the text.

1.8 Don't go running off through the forest just yet! Modify your code to complete the `affine_decrypt_multiblock` kernel which should work when using multiple blocks of threads. Change your grid and block dimensions so that you launch `8` blocks of `128` threads.

Exercise 02

In exercise 2 we are going to extend the vector addition example from the lecture. The file *exercise02.cu* has been provided as a starting point. Perform the following modifications.

2.1 The code has an obvious mistake. Rather than correct it implement a CPU version of the vector addition (Called `vectorAddCPU`) storing the result in an array called `c_ref`. Implement a new function '`validate`' which compares the GPU result to the CPU result. It should print an error

for each value which is incorrect and return a value indicating the total number of errors. You should also print the number of errors to the console. Now fix the error and confirm your error check code works.

2.2 Change the value of `N` to `2050`. Your code will now produce an error as you are writing to GPU memory beyond the bounds which you have allocated. You can confirm this by using the CUDA memory checker in the CUDA debugger (`cuda-gbd`).

After compilation run the `cuda-gdb` program from command line and use the commands highlighted in bold. The debugger will report the error and line number.

```
(cuda-gdb) set cuda memcheck on
(cuda-gdb) run
Starting program: exercise02
[Thread debugging using libthread_db enabled]
 [New Thread 0x7ffff6fe1710 (LWP 7783)]
[Context Create of context 0x6218a0 on Device 0]
[Launch of CUDA Kernel 0 (memset32_post<<<(1,1,1),(64,1,1)>>>)
on Device 0]
Running unaligned_kernel [Launch of CUDA Kernel 1
(unaligned_kernel<<<(1,1,1),(1,1,1)>>>) on Device 0]
Memcheck detected an illegal access to address
(@global)0x400100001
Program received signal CUDA_EXCEPTION_1, Lane Illegal
Address.
[Switching focus to CUDA kernel 1, grid 2, block (0,0,0),
thread (0,0,0), device 0, sm 0, warp 0, lane 0]
0x000000000078b8b0 in unaligned_kernel<<<(1,1,1),(1,1,1)>>> ()
at memcheck_demo.cu:6
6     *(int*) ((char*)&x + 1) = 42;
(cuda-gdb) print &x
$1 = (@global int *) 0x400100000
(cuda-gdb) continue
Continuing.
[Termination of CUDA Kernel 1
(unaligned_kernel<<<(1,1,1),(1,1,1)>>>) on Device 0]
[Termination of CUDA Kernel 0
(memset32_post<<<(1,1,1),(64,1,1)>>>) on Device 0]
Program terminated with signal CUDA_EXCEPTION_1, Lane Illegal
Address.
The program no longer exists.
(cuda-gdb)
```

2.3 Correct the error by performing a check in the kernel so that you do not write beyond the bounds of the allocated memory. Test in the CUDA debugger and ensure that you no longer have any errors.

Exercise 03

We are going to implement a matrix addition kernel. In matrix addition, two matrices of the same dimensions are added entry wise. If you modify your code from exercise 2 by copying the file to a new file called `exercise03.cu`. It will require the following changes;

3.1 Modify the value of `size` so that you allocate enough memory for a matrix size of `N x N` and moves the correct amount of data using `cudaMemcpy`. Set `N` to `2048`.
3.2 Modify the `random_ints` function to generate a random matrix rather than a vector.
3.3 Rename your CPU implementation to `matrixAddCPU` and update the validate function.
3.4 Change your launch parameters to launch a 2D grid of thread blocks with `256` threads per block. Create a new kernel (`matrixAdd`) to perform the matrix addition. Hint: You might find it helps to reduce `N` to a single thread block to test your code.
3.5 Finally modify your code so that it works with none square arrays of `N x M` for any size.

Exercise Solutions

The exercise solutions are available from the solution branch of the repository. To check these out either clone the repository using the branch command to a new directory as follows;

```
$git clone –b solutions
https://github.com/AcceleratedComputing/handsoncuda_day1.git
```

Alternately commit your changes and switch branch

```
$git commit –m "my local changes to src files"
$git checkout solutions
```

You will need to commit your local changes to avoid overwriting your changes when switching to the solutions branch. You can then return to your modified versions by returning to the master branch.