Com 4521 Parallel Computing with GPUs: Lab 04

Spring Semester 2022

Author: Dr Paul Richmond

Lab Assistants: Robert Chisholm, Jack Ashurst, John Charlton

Department of Computer Science, University of Sheffield

Learning Outcomes

- Understand how to launch CUDA kernels
- Understand and demonstrate how to allocate and move memory to and from the GPU
- Understand CUDA thread block layouts for 1D and2D problems
- Learn how to error check code by implementing a reference version
- Learn how to memory check code using the NSIGHT profiler

Prerequisites

Install CUDA Toolkit from the software centre.

Exercise 01

Exercise 1 requires that we de-cipher some encoded text. The provided text (in the file *encrypted01.bin*) has been encoded by using an affine cipher. The affine cypher is a 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. The provided function takes the form of* `modulo(int a, int b)` *where* `a` *in this case is everything left of the affine decryption functions mod operator (e.g. $A^{-1}(x - B)$) and* `b` *is everything to the right of the mod operator (e.g M).*

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, complete the exercise by completing 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. Create a new CUDA project and import the starting code (*exercise02.cu*). 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.

not perfectly divisible by threads per block,
so there are remainders not computed by threads

2.2 Change the value of `N` to `2050`. Your code will now produce an error. Why? Modify your code so that you launch enough threads to account for the error.
2.3 If you performed the above without considering the extra threads then chances are that you have written to GPU memory beyond the bounds which you have allocated. This may not necessarily raise an error. We can check our program for out of bounds exceptions by using the CUDA debugger. Without inserting any breakpoints select the *NSIGHT* menu and ensure that '*Enable CUDA Memory Checker*' is enabled. Start the CUDA debugger, the first time you do this a firewall dialog will appear, press cancel and *NSIGHT* will continue as planned. It will halt in your kernel due to the memory checker detecting an access violation. View the *NSIGHT Output Window* to view a summary of these. 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 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.