CS 610 Coding Assignment 1

Winter 2026

Total points: 11 (including 1 extra bonus point)

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 and 2D problems
- Learn how to error check code by implementing a reference version

For all tasks:

- Use the A100 GPU on Talapas to complete all tasks. You need to follow the instruction 'How To Use GPUs on Talapas' to set up GPU runtime environment.
- Compile each task by first load the CUDA module using '**load cuda**' and compile with '**nvcc <source code filename> -o <executable name>**' (replace <source code filename> and <executable name> with the actual source code filename and executable name)
- Run each task by executing './<executable name>' in either batch mode or interactive mode on Talapas.
- After you finish all tasks, submit your completed *task01.cu, task02.cu, and task03.cu to Canvas.*

Task 01 (5 points)

Task 1 requires that we de-cipher some encoded text. The provided text (in the file *encrypted.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 task 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 task 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. The decrypted the text is provided as reference.

Starting from the code provided (*task01.cu*), complete the task by completing the following.

1.1 (0.5 point) Modify the `modulo` function so that it can be called on the device by the `affine_decrypt` kernel.

1.2 (1 point) Implement the decryption kernel for a single block of threads with a one 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. You decrypted text using this kernel should match the provided reference to earn credits.

1.3 (0.5 point) Allocate some memory on the device for the input (`d_input`) and output (`d_output`).

1.4 (0.5 point) Copy the host input values in `h_input` to the device memory `d_input`.

1.5 (0.5 point) Configure a single block of `N` threads and launch the `affine_decrypt` kernel.

1.6 (0.5 point) Copy the device output values in `d_output` to the host memory `h_output`.

1.7 (0.5 point) Free all GPU memory allocations before the program exits.

1.8 (1 point) 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. Modify the kernel invocation statement so it calls the multiblock kernel. Again, you decrypted text should match the provided reference using this new kernel to earn credits.

Task 02 (2 points)

In task 2 we are going to extend the vector addition example from the lecture. Use the start code *task02.cu*, and perform the following modifications.

2.1 (0.5 point) The code has an obvious mistake. Rather than correct it now, let's first 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.

2.2 (0.5 point) Now fix the error and confirm your error check code works.

2.3 (0.5 point) 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.

2.4 (0.5 point) 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. Correct the error by performing a check in the kernel so that you do not write beyond the bounds of the allocated memory.

Task 03 (4 points)

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 task 2 it will require the following changes*. (Make a copy of task02.cu and name the new file as task03.cu)*

3.1 (0.5 point) 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 (0.5 point) Modify the `random_ints` function to generate a random matrix rather than a vector.

3.3 (0.5 point) Rename your CPU implementation to `matrixAddCPU` and also update the validation function.

3.4 **(1.5 point)** Change your launch parameters to launch a 2D grid of 2D thread blocks with `16*16` 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 (1 point) Modify your code so that it works with non-square matrices of `N x M` for any size.