hw1

ECE 3803

Jeff Epstein

Contents

1 Introduction

This assignment will help you practice the CUDA tool chain to build a simple kernel for manipulating vectors. If you haven't yet done so, make sure you know how to use ICE to compile and run CUDA programs. Consult the ICE tutorial on Canvas.

2 Assignment Part 1: Cypher (35 points)

2.1 Cypher

The *Caesar cypher* is a simple method for encrypting data, wherein each element of an array (or letter of a message) is disguised, by shifting its value within the alphabet by a fixed number of units. For example, if we shift by three letters, then each “A” would be “D”, as shown below. Letters at the end of the alphabet “wrap around” to the beginning, so that a “Z” shifted by three would become a “C”, assuming that we’re using the usual alphabet of 26 letters.

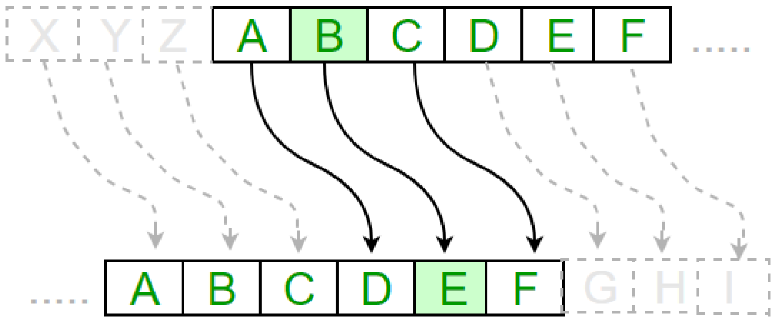

Figure 1: a Caesar cypher of shift 3

Thus,

```
Input : ATTACKATONCE
Shift: 4
Output: EXXEGOEXSRGI
```

You are given a program that creates a large array initialized with random integers, selects a random shift, and then performs that shift on each element in that array. The program includes both a CPU implementation and a GPU implementation of this algorithm. However the GPU version is incomplete.

2.2 Code

Download the associated zip file. Extract the contents with the `unzip` command. Read all source code before you start to write code.

Follow the instructions in the ICE tutorial to log in to ICE, copy your files there, load the CUDA compiler, and allocate a GPU.

Compile the program using the `make` command, which will automatically invoke the `nvcc` CUDA compiler. You can then run the program by typing `./cypher` and hitting enter.

2.3 Your task

Your task is to complete the shift_cypher function, replacing the “TODO your code here” comment with the GPU implementation of your cypher function. Your implementation should produce identical results to the CPU version, host_shift_cypher. Both functions read data from their input_array parameter, write output to their output_array parameter, shift each element by shift_amount, wrapping to alphabet_max, such that both arrays are of size array_length.

Add your code to the shift_cypher function, but do not add or modify any other code in the file.

When you run the program, it will print “Success” or “Failure”, depending on whether your GPU implementation’s results match the CPU implementation’s results. In addition, it will print out performance metrics for each implementation. Please note the timings will be initially unstable due to caching; to get accurate measurements, you should run the program several times until the timings stabilize.

Your program is evaluated for its correctness (in terms of matching the CPU implementation’s results) and efficiency (effectively using the GPU resources). You will need to take into account the kernel’s block size and grid size in order to design a kernel that avoids redundant work.

Use only the techniques that we’ve discussed in class so far this semester.

3 Assignment Part 2: Particles (55 points)

3.1 Particles

We can use Coulomb’s law to calculate the force between two particles.

$$|F| = k_\mathrm{e}\frac{|q_1||q_2|}{r^2}$$

> The magnitude of the electrostatic force F between two point charges q_1 and q_2 is directly proportional to the product of the magnitudes of charges and inversely proportional to the square of the distance between them.

You are given a program that creates a large array initialized with random particle data and calculates the force vector between selected pairs of particles. The program includes both a CPU implementation and a GPU implementation of this algorithm. However the GPU version is incomplete.

3.2 Code

Download the associated zip file. Extract the contents with the unzip command. Read all source code before you start to write code.

Follow the instructions in the ICE tutorial to log in to ICE, copy your files there, load the CUDA compiler, and allocate a GPU.

Compile the program using the make command, which will automatically invoke the nvcc CUDA compiler. You can then run the program by typing ./particle and hitting enter.

3.3 Your task

Your task is to complete two functions:

- `force_eval`, the CUDA kernel for calculating the force between the specified particles.
- `charged_particles`, the host-side launcher code for `force_eval`. Here, you will have to allocate device memory, copy data, launch the kernel, and deallocate device memory.

The parameters to `charged_particles` are described as follows:

- `set_A` and `set_B` are arrays of type `float4`, where each element contains four floats representing the X, Y, and Z coordinates of a particle, as well as its charge.
- `array_length` is the length of the `indices` array.
- `indices` is an array of size `array_length`, such that each value `j` at index `i` in the array is the position in the `set_B` array of the particle whose force value should be calculated with the element in `set_A` at position `i`; however, some values of the elements in `indices` are invalid, being either less than zero or larger than or equal to `array_length`.
- `force_vectors` is an output parameters, so that when the function ends `force_vectors[i]` contains the force vector of `set_A[i]` and `set_B[indices[i]]`, if `indices[i]` is valid; if, however, `indices[i]` is out of range, the corresponding element in `force_vectors` should be set to the zero vector.

In both functions, replace the "TODO" comments with the appropriate GPU code. Your implementation should produce identical results to the CPU version `host_force_eval`.

Add your code to the above functions, but do not add or modify any other code in the file. Use the provided `CHECK_ERROR` macro and `check_launch` function to catch any potential runtime errors.

When you run the program, it will print "Success" or "Failure", depending on whether your GPU implementation's results match the CPU implementation's results. In addition, it will print out performance metrics for each implementation. Please note the timings will be initially unstable due to caching; to get accurate measurements, you should run the program several times until the timings stabilize.

Your program is evaluated for its correctness (in terms of matching the CPU implementation's results) and efficiency (effectively using the GPU resources). You will need to select an appropriate block size and grid size, and take that information into account in the kernel, in order to produce optimal results.

Use only the techniques that we've discussed in class so far this semester.

3.4 Utilities

As a convenient way to use four-dimensional vectors, this program uses the `float4` type and `make_float4` function provided by CUDA. Their definitions (slightly simplified) are given below.

```
struct float4 {
    float x, y, z, w;
};

float4 make_float4(float x, float y, float z, float w) {
  float4 t;
  t.x = x; t.y = y; t.z = z; t.w = w;
  return t;
}
```

These utilities are usable in both host code and device code.

4 Assignment Part 3: Questions (10 points)

Answer the following questions in a plain text file (not Word, not PDF) named `hw1.txt`.

1. We know that CPUs typically have better latency while GPUs typically have better throughput. We'd like to quantify this contrast using data based on the execution of your `cypher` program.

   Let's assume that GPU kernel execution entails a fixed cost of device memory allocation and transfer to device, which does not vary with the quantity of data (this isn't quite an accurate assumption, but it's accurate enough for this exercise). In addition, let's assume a variable cost of GPU kernel execution that scales linearly with the size of the data set. On the CPU side, assume there are no fixed costs, but only a variable cost associated with the calculation that scales linearly with the size of the data set.

   For the GPU fixed cost, GPU variable cost, and CPU variable cost, use the timing numbers produced by your execution of the `cypher` program. Derive an execution model that will allow us to calculate the quantity of data at which point the two competing hardware paradigms "break even," i.e. how much data will cause the CPU execution to be just as fast as the GPU execution.

   Give your raw data acquired from the execution of `cypher`. Show all work and calculations. Express your final answer as a number of bytes.

2. Consider the following incomplete kernel:

   ```
   __global__ hypothetical_kernel(int *array2d) {
       int x = threadIdx.x * 3 + blockIdx.x * blockDim.x;
       int y = threadIdx.y * 2 + blockIdx.y * blockDim.y;
       ....
   }
   ```

   After the kernel finishes executing, all elements of `array2d` have been processed. How many elements of `array2d` would you expect to be processed by each thread? Explain your answer.

5 Hints and common errors

If you get the error "CUDA driver version is insufficient for CUDA runtime version", it means that you are running the program on a computer without a GPU. Make sure you are following the instructions in the ICE tutorial for allocating a GPU.

If you get the error "nvcc: No such file or directory", it means that you have not loaded the CUDA compiler. Make sure you are following the instructions in the ICE tutorial for loading the CUDA compiler.

If you get the error “the provided PTX was compiled with an unsupported toolchain”, it means that are using a wrong version of the CUDA compiler. Make sure you are following the instructions in the ICE tutorial for loading the CUDA compiler. In particular, please load the CUDA compiler module only after allocating a compute node on ICE. You may have to run `make clean` to delete the old executables.

If you get the error “No targets specified and no makefile found”, it means that you have not copied and extracted the assignment files or you are not in the correct directory when you run `make`. Check the contents of your current directory to ensure that it contains `Makefile` and the other files required for this assignment.

Note the use of the provided `CHECK_ERROR` macro and `check_launch` function for checking potential errors returned by CUDA function calls running on the host. You must use this macro, or a similar technique, to catch errors in your own host-side CUDA code.

6 Rules

6.1 Development environment

Your code will be tested in our ICE environment, as described in the syllabus. Please test your code in that environment before submission. Your code must work with the operating system and CUDA development tool chain installed there.

You may use the CUDA libraries, as described in the ICE tutorial; as well as standard libraries that form part of the compiler. Do not use any other libraries.

6.2 Evaluation

Your submitted work will be evaluated on the following metrics, among others:

- your submission’s fulfillment of the goals and requirements of the assignment. That is, does your program work?
- your submission’s correct application of the techniques and principles studied in this course. That is, does your program demonstrate your understanding of the course material?
- your submission’s performance characteristics. That is, does your program run fast enough?

Failure to satisfy the assignment’s requirements, to apply relevant techniques, or to meet performance expectations may result in a grade penalty.

Only work submitted before the due date will be evaluated. Exceptions to this policy for unusual circumstances are described in the syllabus.

Ensure that your program compiles and runs without errors or warnings in the course’s development environment. If your code cannot be compiled and run, it will not be graded.

In addition, you are obligated to adhere to the stylistic conventions of quality code:

- Indentation and spacing should be both consistent and appropriate, in order to enhance readability.
- Names of variables, parameters, types, fields, and functions should be descriptive. Local variables may have short names if their use is clear from context.

- Where appropriate, code should be commented to describe its purpose and method of operation. If your undocumented code is so complex or obscure that the grader can't understand it, a grade penalty may be applied.
- Your code should be structured to avoid needless redundancy and to enhance maintainability.

In short, your submitted code should reflect professional quality. Your code's quality is taken into account in assigning your grade.

6.3 Academic integrity

You should write this assignment entirely on your own. You are specifically prohibited from submitting code written or inspired by someone else, including code written by other students. Code may not be developed collaboratively. Do not use any artificial intelligence resource at any point in your completion of this assignment. Please read the course syllabus for detailed rules and examples about academic integrity.

7 Submission

Submit only your completed `hw1.txt`, `cypher.cu`, and `particle.cu` files on Gradescope. Do not submit binary executable files. Do not submit data files.