CME 213, Introduction to parallel computing
Eric Darve
Spring 2023

Stanford University

Homework 3

Total number of points: 140.

In this programming assignment, you will use NVIDIA's Compute Unified Device Architecture (CUDA) language to implement basic recurrence and pagerank algorithms. In the process, you will learn how to write general-purpose GPU programming applications and consider some optimization techniques. You must turn in your own copy of the assignment as described below. Although you can collaborate with your classmates on the assignment, sharing solutions is strictly prohibited. If you have any queries regarding the task, kindly post them on the forum.

Every time you open the terminal, you have to run

```
$ module load cuda
```

Alternatively, you can add this to your `.bashrc`. To compile and run your code, run `sbatch hw3.sh`. The output will be in `slurm.sh.out`.

For all questions asking to comment on plots, make sure to describe the shape and different regions (such as increasing performance or asymptotic behavior) of the graph and explain why these patterns may emerge.

CUDA

"C for CUDA" is a programming language subset and extension of the C programming language and is commonly referenced as simply CUDA. Many languages support wrappers for CUDA, but in this class, we will develop in C for CUDA and compile with `nvcc`.

The programmer creates a general-purpose kernel to be run on a GPU, analogous to a function or method on a CPU. The compiler allows you to run C++ code on the CPU and the CUDA code on the device (GPU). Functions which run on the host are prefaced with `__host__` in the function declaration. Kernels run on the device are prefaced with `__global__`. Kernels that are run on the device and that are only called from the device are prefaced with `__device__`.

The first step you should take in any CUDA program is to move the data from the host memory to device memory. The function calls `cudaMalloc` and `cudaMemcpy` allocate and copy data, respectively. `cudaMalloc` will allocate a specified number of bytes in the device main memory and return a pointer to the memory block, similar to malloc in C. You should not try to dereference a pointer allocated with `cudaMalloc` from a host function.

The second step is to use `cudaMemcpy` from the CUDA API to transfer a block of memory from the host to the device. You can also use this function to copy memory from the device to the host. It takes four parameters, a pointer to the device memory, a pointer to the host memory, a size, and the direction to move data (`cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`). We have already provided the code to copy the string from the host memory to the device memory space, and to copy it back after calling your shift kernel.

Kernels are launched in CUDA using the syntax `kernelName<<<...>>>(...)`. The arguments inside of the chevrons (`<<<blocks, threads>>>`) specify the number of thread blocks and thread per block to be launched for the kernel. The arguments to the kernel are passed by value like in normal C/C++ functions.

There are some read-only variables that all threads running on the device possess. The three most valuable to you for this assignment are `blockIdx`, `blockDim`, and `threadIdx`. Each of these variables contains fields x, y, and z. `blockIdx` contains the x, y, and z coordinates of the thread block where this thread is

located. `blockDim` contains the dimensions of thread block where the thread resides. `threadIdx` contains the indices of this thread within the thread block.

We encourage you to consult the development materials available from NVIDIA, particularly the CUDA Programming Guide and the Best Practices Guide available at
`http://docs.nvidia.com/cuda/index.html`

Unit test fixtures

Sometimes when testing your code, you may notice that you are doing very similar operations to set up certain tests. In such cases, GoogleTest provides test fixtures that enable you to reuse the same object configuration for multiple tests. For the recurrence problem below, we will employ this approach to streamline our testing procedures. Additional information on test fixtures can be found in the GoogleTest documentation:
`http://google.github.io/googletest/primer.html#same-data-multiple-tests`

Fixture tests must use the macro `TEST_F`. The first argument of the macro must be the name of the test fixture class, `RecurrenceTestFixture`. Using test fixtures, the following sequence of code is executed:

1. A `RecurrenceTestFixture` object is created.
2. The first test
   `TEST_F(RecurrenceTestFixture, GPUAllocationTest_1)`
   runs. This test is able to access objects and subroutines in the test fixture object `RecurrenceTestFixture`.
3. Once the test completes, the test fixture object is destructed.

This sequence is repeated for all subsequent fixture tests `TEST_F`.

Problem 1 Recurrence

The purpose of this problem is to give you experience writing your first simple CUDA program. This program will help us examine how various factors can affect the achieved performance.

Inspired by the Mandelbrot Set, we want to perform the following recurrence for several values of c:

$$z_{n+1} = z_n^2 + c.$$

z is in general a complex number but for simplicity we will use `floats` in this homework. For each value of c, you can study the sequence z_n. If z_n does not diverge (starting from $z_0 = 0$) then the point c belongs to the Mandelbrot set. In Figure 1, the coordinates of each pixel correspond to the real and imaginary parts of c. The color of a pixel is determined by computing the smallest iteration n for which $|z_n| > 2$. One can prove that if $|z_n| > 2$ for some n then $|z_n| \to \infty$ as $n \to \infty$. The recurrence is done for a maximum of `num_iter` iterations, and the values of the c's are set in `initialize_array()` in `main_q1.cu`.

Code

You should be able to take the files we give you and type `make main_q1` to build the executable. The executable will run, but since the CUDA code hasn't been written yet (that's your job), it will report errors and quit. All locations where you need to write code are noted by a `TODO` in the comments. For this problem we provide the following starter code (* means you should *not* modify the file):

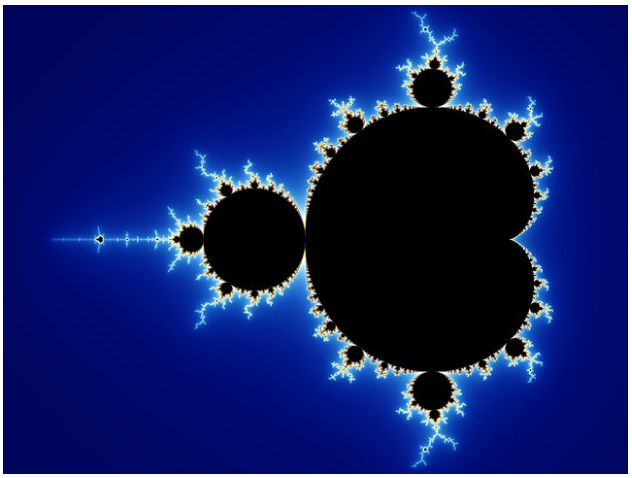

Figure 1: Mandelbrot Set. Source: Wikipedia. The black points in the image belong to the Mandelbrot set. The sequence z_n does not diverge for the corresponding c. Although not obvious, the Mandelbrot set is a connected set.

- `main_q1.cu`—This is the main file. We have already written most of the code for this assignment so you can concentrate on the CUDA code. We take care of computing the host solution and checking your results against the host reference. There is also code to generate the tables you will need to do the benchmarking questions. You will do questions 1 and 2 in this file.
- `recurrence.cuh`—This file already contains the necessary function headers—do not change these. You should fill in the body of the kernel and launch the kernel from `doGPURecurrence()`.
- `*test_recurrence.h`—This file contains the functions we will use to check your output. Do not modify this file.
- `*Makefile`—`make run1` will build and run the binary. `make main_q1` will build the binary. `make clean` will remove the executables. You should be able to build and run the program when you first download it, however only the host code will run.
- `hw3.sh`—This script is used to submit jobs to the queue. You need to comment out the other lines in the file if you only want to run `./main_q1`.

Question 1.1

10 points. Allocate GPU memory for the input and output arrays for the recurrence within the test fixture constructor. Free this GPU memory at the end in the destructor. This code (approx. 4 lines) should be in `main()` in `main_q1.cu`.

Question 1.2

10 points. Implement `initialize_array()`, the function that initializes an array of a given size. The values are random floats between -1 and 1. These will be the constants c in the recurrence. This code should be in `main_q1.cu`.

Question 1.3

20 points. Implement the recurrence kernel and launch it. These should be implemented in `recurrence()` and `doGPURecurrence()` respectively in `recurrence.cuh`. You can see a CPU implementation of the recur-

rence in `host_recurrence()` and a sample launch of the kernel in `main()`, both in `main_q1.cu`. Add the output of the code (it should be 3 tables) to your PDF submission. The whole run may take 10 minutes.

Question 1.4

10 points. Run the same setup but with the number of blocks to be 72, the number of iterations to be 40,000, and the array size (number of constants we test) to be 1,000,000 (this code has already been written for you). Vary the number of threads per block as 32, 64, 96, …, 1024. Take the table that is generated and plot the performance in TFlops/sec vs. the number of threads. Comment on and explain the shape of the graph.

Question 1.5

10 points. Run the same setup with the number of threads per block to be 128, the number of iterations to be 40,000, and the array size (number of constants we test) to be 1,000,000 (this code has already been written for you). Vary the number of blocks as 36, 72, 108, …, 1152. Take the table that is generated and plot the performance in TFlops/sec vs. the number of blocks. Comment on and explain the shape of the graph. Hint: this GPU has 72 SMs and 8 blocks per SM.

Question 1.6

10 points. Run the same setup with the number of threads per block to be 256, the number of blocks to be 576, and the array size (number of constants we test) to be 1,000,000 (this code has already been written for you). Vary the number of iterations as in the code. Take the table that is generated and plot the performance in TFlops/sec vs. the number of iterations. Comment on and explain the shape of the graph.

Problem 2 PageRank

PageRank was the link analysis algorithm responsible (in part) for the success of Google. It generates a score for every node in a graph by considering the number of in links and out links of a node. We are going to compute a simplified model of pagerank, which, in every iteration computes the pagerank score as a vector π and updates π as

$$\pi(t+1) = \frac{1}{2}\, A\pi(t) + \frac{1}{2N}\, \mathbf{1}$$

where A is a normalized adjacency matrix (so that each column sums to 1), N is the number of nodes in the graph and $\mathbf{1}$ is a vector with all 1's. Each entry in the vector π corresponds to the score for one node. The matrix A is sparse and each row i corresponds to the node n_i, the non-zero entries correspond to the nodes n_j that have a directed edge to n_i (i.e., $A_{ij} > 0 \Leftrightarrow (n_j, n_i) \in E$, where E is the set of directed edges). Since we normalize the columns of A, the entries in the j'th column are all proportional to $1/\text{outDegree}(n_j)$. We will choose the *average* number of connections for a node to be $\mu \in \mathbb{N}_+$ and then have the actual number of connections per node vary from 1 to $2\mu - 1$. The total number of edges is given by $|E| = \mu N$.

In the actual algorithm this operation is performed until the change between successive π vectors is sufficiently small. In our case we will choose a fixed number of iterations to more easily compare performance across various numbers of nodes and edges. If you wish to learn more about the algorithm itself, check `http://en.wikipedia.org/wiki/PageRank`

For this problem, we provide the following starter code (* means you should *not* modify the file):

- `*main_q2.cu`—contains the code that sets up the problem and generates the reference solution. It also has a result generating loop that will generate a table of timing results for various numbers of edges and nodes. You should not need modify this file.
- `pagerank.cuh`—this is the file you will need to modify and submit. Do not change the function headers but fill in the bodies and follow the hints/requirements in the comments.

- `*Makefile`
  `$ make run2`
  will build and run the pagerank binary.
  `$ make main_q2`
  will build the pagerank binary.
  `$ make clean`
  will remove the executables. You should be able to build and run the program when you first download it. However, only the host code will run.

- `hw3.sh`—This script is used to submit jobs to the queue. You need to comment out the other lines in the file if you only want to run `./main_q2`.

Question 2.1

35 points. Fill in the functions so that the program no longer reports any errors.

Question 2.2

10 points. What is the formula for the total number of bytes **read from and written to** the global memory by the algorithm? Analyze the code you've written and do the calculation "on paper" instead of running actual code. *Hint: your answer should be based on the number of nodes, the average number of edges, and the number of iterations. Don't include any data transfer between CPU and GPU in this calculation.*

Add in the bandwidth calculation in the function `get_total_bytes` to reflect your answer to Question 2.2 in `pagerank.cuh`.

Question 2.3

5 points. From the table of results, plot the memory bandwidth (GB/sec) vs. problem size for an average number of edges equal to 10. Make sure the plot is readable. Do not comment the plot in this question.

Question 2.4

10 points. Comment on the plot. What does the memory access pattern look like for this problem? Using your answer to this question, explain the difference in bandwidth between Problem 2 and the maximum bandwidth of about 480 GB/sec (measured on the icme-gpu cluster).

Problem 3 Benchmarking with Strided Memory Access

For this problem, we will benchmark our device by performing strided memory accesses. The file `benchmark.cuh` performs a benchmark using two very long input arrays x and y, as well as an output array z, by computing $z[i] = x[i] + y[i]$ at stride lengths between 1 and 32. That is, $z[i] = x[i] + y[i]$ for $i \in \{0, 1, 2, 3, \dots\}, i \in \{0, 2, 4, 6, \dots\}, \dots, i \in \{0, 32, 64, 96, \dots\}$.

For this problem, we provide the following starter code (* means you should not modify the file):

- `*main_q3.cu`—sets up the CUDA runtime and launches your benchmarking kernel with stride lengths in $1 \dots 32$

- `benchmark.cuh`—this is the file you will need to modify and submit. Do not change the function headers but fill in the bodies and follow the hints/requirements in the comments.

- `*Makefile`
  `$ make run3`
  will build and run the benchmarking binary.
  `$ make main_q3`

will build the benchmarking binary.
`$ make clean`
will remove the executables. You should be able to build and run the program when you first download it. However, your results will be incorrect as your kernel won't be performing any memory accesses.

- `hw3.sh`—This script is used to submit jobs to the queue. You need to comment out the other lines in the file if you only want to run `./main_q3`.

Question 3.1

5 points. Perform the strided memory access in benchmark.cuh. Then, in the terminal, run `make benchmark`. In your writeup, display the results on a semilogy plot of throughput in GB/s as a function of stride length. Do not comment on the plot under this part.

Question 3.2

5 points. Comment on and explain the shape of the graph. Why do we observe the trend that we do as the stride length increases?

A Submission instructions

To submit:

1. For all questions that require explanations and answers besides source code, put those explanations and answers in a separate PDF file and upload this file on Gradescope.

2. The homework should be submitted using a submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.

3. Copy your submission files to `cardinal.stanford.edu`. The script `submit.py` will copy only the files below to a directory accessible to the CME 213 staff. Only these files will be copied. Any other required files (e.g., Makefile) will be copied by us. Therefore, make sure you make changes only to the files below. You are free to change other files for your own debugging purposes, but make sure you test it with the default test files before submitting. Also, do not use external libraries, additional header files, etc, that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   ```
   main_q1.cu
   recurrence.cuh
   pagerank.cuh
   benchmark.cuh
   ```

   The script will fail if one of these files does not exist.

4. To check your code, we will run the following on `icme-gpu`:
   `$ make`
   This should produce 3 executables: `main_q1`, `main_q2` and `main_q3`.

5. To submit, type:

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw3 <directory with your submission files>
```

B Advice and Hints

- To perform a batch update, we use two pagerank vectors in our algorithm and switch their roles on every iteration (reading from one and writing to the other).
- For debugging it will be helpful to limit the number of cases being run to 1. In the recurrence problem, do this by using 1 value instead of the arrays for the 3 for loops. In the pagerank problem, change the values of `num_nodes` and `num_edges`.
- If you need some documentation on CUDA, you can look at the documents linked on canvas or visit the CUDA website at `https://docs.nvidia.com/cuda/index.html`.
- For Problem 2, make sure you understand how the sparse matrix is encoded in memory. This will greatly help you figure out the code to write.
- An easy way to transfer the table output into a plot is to copy the space-separated program output, paste it into a Google Sheet, highlight the column that contains the data, and click "Data→Split text to columns" in the top banner, then hightlight your new columns and click "Insert→Chart."