COMP 529: Parallel Programming

HW-2

Najeeb Ahmad (0059486) and Mohammad Laghari (0059487)

We are competing for the best GFlops. The best Gigaflops figure we achieved is **62.700914** over 100 iterations and lambda 0.5 on coffee.pgm. We have implemented all the versions using 1-dimensional grids and blocks and have completed version 1 and version 2 using 2-dimensional grids and blocks. The reported performance is for version 2 of 2-dimensional grids and blocks as it was the best we achieved so far. We were on our way to complete the third version as well for 2-dimensional implementation, had the GPU not malfunctioned.

Introduction

The following report is divided as follows. The first section defines the parallel implementation of the program in version 1. Section 2 talks about the version 2. Section 3 discusses version 3. And Section 4 shows the results.

1. Version 1: Naive implementation

We started off from the serial implementation and added necessary CUDA directives to make sure it was functional. We also queried the GPU info so that we could map our application accordingly (Querying device memory, shared memory per block, maximum blocks/threads etc.). We allocated space in the GPU for our data structures in the global memory of the device. We also made a copy of the image and send it to the device so that the device has a copy for itself to execute the device specific part of the program. We came up with two strategies to the execution of the program on the device and we chose the parts which maximized our GFLOPS rate while keeping the solution accurate. The two strategies are discussed in the following sections.

1.1 Strategy 1: 1-dimensional grids and blocks

This was our initial strategy. We picked this because it was simpler to implement, especially for the reduction part. In this version, the representation of blocks in a grid and the threads in a block is in one direction only. This made indexing easier than the other strategy discussed later. Initially, we prepared a kernel for the first loop, that is, the loop which does the reduction. According to our strategy, the kernel takes in the image and each thread in the device calculates the value of sum and sum2 and keeps it in its designated place in the output array, called partial_sum and partial_sum2 respectively. After each thread has computed the value of

sum and sum2, we perform reduction on each block, such that after this reduction, we will obtain a single value of sum and sum2 for each respective block in the grid. This final array (which has the size equal to the number of blocks used) is then exported to the CPU (host) so that a final value of sum and sum2 can be computed serially and mean and standard deviation can be calculated.
After this, we started to implement the second kernel, which is responsible to run *Compute 1* on the device. This was straightforward to implement as it only needs to stay within the bounds of the block and that is simply achieved by adding if conditions. This kernel, which we named as Compute1, requires us to allocate for device variables for the intermediate arrays which calculate the respective derivatives in each direction. These variables were initially copied back to the host when the parallelization was being done in an incremental process, but once all the kernels were written, the need to bring back these data structure was finished.
Lastly, we implemented the third kernel for *Compute 2* and named it, Compute2. We did not need to perform any cudaMemcpy operation for this kernel as all the data this function requires is already present on the device memory. However, there was still a need to allocate memory on the device for the device variant of the variables. Again, the implementation for this kernel was straightforward as we only needed to be careful about the boundary conditions. And a simple if condition sufficed for that.

1.2 Strategy 2: 2-dimensional grids and blocks

In this strategy, we changed the design of our grid structure from one-dimensional to two-dimensional for Compute 1 and Compute 2 while retaining reduction part as one-dimensional. So now, each grid has blocks in two dimensions and each block has threads in two dimensions. Again, the value of the pixel at a particular point is computed in the designated kernels and a thread performs its computations only on this particular pixel. The data transfer between host and the device is the same as it was in the previous strategy. So all-in-all, the only difference between this strategy and the prior one to this was the way indexing was performed on the image by the device.

2. Version 2: Implementation using temporary variables

After the initial implementation was complete, this version was easier to implement as it only required us to identify the data structures that are referenced more than once from global memory and we placed them in predefined variables (registers) so that the additional cost to fetch data from memory could be reduced. For instance, in the kernel for Compute1 the value of image at a particular index is used more than once by a thread to calculate the derivative at each direction. So, instead of fetching it over and over again, we placed this in a register and used that register instead. This way we cut down the cost of 7 memory accesses by using a register.

3. Version 3: Implementation using shared memory

3.1 Strategy 1: 1-dimensional grids and blocks

As described in the previous sections, our program has two implementation, this subsection talks about the implementation done in 1D. For this part, the program required us to make changes only in the respective kernels. The first change was made in the kernel named Compute1. This version has three shared variables defined. The first one is the array that holds the image values at each index. This array has three rows and the number of columns is same as the number of threads in a block (since it will be shared by all the threads in a particular block). We use three rows for this array because we need to store the north value and the south value as well for each pixel in our implementation. Each thread in the block loads the pixel value corresponding to the global thread ID of the thread as well as north and east neighbors of the said pixel. The first and the last threads in a block additionally load west and east neighbor pixels in addition to the north and south pixels. To avoid resizing the array, we made two separate register variables for saving east and west as they would only be used for a single value each. After loading pixel pixel values into shared memory, threads in the block are synced and then each thread calculates its respective value of derivatives and perform further calculations as per algorithm. This implementation is slower than the 2-dimensional grids and blocks implementation (next section) because it has many divergent branches due to if conditions.

3.2 Strategy 2: 2-dimensional grids and blocks

The purpose of this implementation was to explore if we could further improve the performance of our code while doing it in 2-dimension grids and blocks. We identified that doing this strategy requires us to load ghost cells for each block at the boundary and needed additional checks for the blocks lying on the boundary and edges. We implemented majority of the conditions to satisfy the ghost cell boundaries. However, we were unable to complete it due to the irregularities on the GPU that were experienced on the last night of the submission deadline. We anticipated an increase in performance after implementing the program using shared memory in 2D, but were unable to complete it due to logistical issues. Conclusively, we implemented the program in 2-dimensions uptil version 2 and were in the process of completing it using shared memory as well, had it not been for the GPU malfunctioning.

4. Results

The results obtained in the **serial execution** are as follows:

```
Image Read. Width : 5184, Height : 3456, nComp: 1
Time spent in different stages of the application:
 0.000014 s => Part I: allocate and initialize variables
 0.000021 s => Part II: parse command line arguments
 0.018624 s => Part III: read image
 0.001917 s => Part IV: allocate variables
122.030437 s => Part V: compute
 2.390337 s => Part VI: write image to file
 0.046408 s => Part VII: get average of sum of pixels for testing and calculate GFLOPS
 0.046408 s => Part VIII: deallocate variables
Total time: 124.500962 s
Average of sum of pixels: 121.615486
```
GFLOPS: 0.660370

4.1 Implementation in 1-dimensional grids and blocks

4.1.1 Version 1

```
Time spent in different stages of the application:
 0.000008 s => Part I: allocate and initialize variables
 0.000026 s => Part II: parse command line arguments
 0.528381 s => Part III: read h_image
 0.002665 s => Part IV: allocate variables
 1.544828 s => Part V: compute
 2.462514 s => Part VI: write h_image to file
 0.054045 s => Part VII: get average of sum of pixels for testing and calculate GFLOPS
 0.054045 s => Part VIII: deallocate variables
Total time:  4.596137 s
Average of sum of pixels: 121.615486
```
GFLOPS: 52.164570

4.1.2 Version 2

```
Time spent in different stages of the application:
 0.000019 s => Part I: allocate and initialize variables
 0.000059 s => Part II: parse command line arguments
 0.441314 s => Part III: read h_image
 0.002412 s => Part IV: allocate variables
 1.553984 s => Part V: compute
 3.202842 s => Part VI: write h_image to file
 0.070561 s => Part VII: get average of sum of pixels for testing and calculate GFLOPS
 0.070561 s => Part VIII: deallocate variables
Total time:  5.275393 s
Average of sum of pixels: 121.615486
```
GFLOPS: 51.857219

4.1.3 Version 3

```
Time spent in different stages of the application:
 0.000024 s => Part I: allocate and initialize variables
 0.000036 s => Part II: parse command line arguments
 0.432471 s => Part III: read h_image
 0.002406 s => Part IV: allocate variables
 1.536116 s => Part V: compute
 3.207777 s => Part VI: write h_image to file
 0.073183 s => Part VII: get average of sum of pixels for testing and calculate GFLOPS
 0.073183 s => Part VIII: deallocate variables
Total time:  5.256423 s
Average of sum of pixels: 121.615486
```
GFLOPS: 52.460420

4.2 Implementation in 2-dimensional grids and blocks

4.2.1 Version 1

```
Time spent in different stages of the application:
 0.000007 s => Part I: allocate and initialize variables
 0.000019 s => Part II: parse command line arguments
 0.469281 s => Part III: read h_image
 0.004028 s => Part IV: allocate variables
 1.292492 s => Part V: compute
 2.924884 s => Part VI: write h_image to file
 0.057243 s => Part VII: get average of sum of pixels for testing and calculate GFLOPS
 0.057243 s => Part VIII: deallocate variables
Total time:  4.752049 s
Average of sum of pixels: 121.615486
```
GFLOPS: 62.348761

4.2.2 Version 2

```
Time spent in different stages of the application:
 0.000019 s => Part I: allocate and initialize variables
 0.000036 s => Part II: parse command line arguments
 0.440015 s => Part III: read h_image
 0.002375 s => Part IV: allocate variables
 1.285233 s => Part V: compute
 2.573951 s => Part VI: write h_image to file
 0.054373 s => Part VII: get average of sum of pixels for testing and calculate GFLOPS
 0.054373 s => Part VIII: deallocate variables
```

Total time: 4.356803 s
Average of sum of pixels: 121.615486
GFLOPS: 62.700914

4.2.3 Version 3

Could not complete implementation because of GPU malfunctioning.

4.3 Efficiency

The following table summarizes the efficiency achieved when compared to the serial execution. The formula we use to calculate the efficiency is as follows:

$$Efficiency = \frac{GFlops\ in\ parallelized\ execution}{GFlops\ in\ serial\ execution}$$

Version	**Version 1 (1D)**	**Version 2 (1D)**	**Version 3 (1D)**	**Version 1 (2D)**	**Version 2 (2D)**
Efficiency	78.99	78.52	79.44	94.41	94.94