CPEN512 Assignment 3: CUDA

Jiaqi Zhang (63174551)

1. Background

With the development of graphics cards, the GPUs are becoming more and more powerful. Due to its parallel processing capability, it can efficiently handle huge calculations from matrix multiplication to deep neural network training. The objective of this assignment is to parallelize the gaussian elimination by taking advantage of GPUs.

2. Implementation

The first step is to prepare data from CPU, or host. Then we copy this matrix of NxN to device side, or GPU. There are two main steps in Gaussian forward elimination: normalization of a row with one of its element (pivot), and for all the below rows eliminate with this row such that the pivot column becomes 0.

For the first step, we use normal blocks and threads to concurrently normalize based on the pivot. After syncing threads, for the 2d elimination process we will make use of dim3 structure for both blocks and threads so their sizes are *(NUM_BLOCKS, NUM_BLOCKS)* and *(NUM_THREADS, NUM_THREADS)*. In this way we can achieve maximum paralysis. Finally, we copy data back to CPU and evaluate time spent on GPU calculation.

3. Performance

3.1 Overview

In general, the performance has vastly improved compared with the CPU version of GE. One obvious improvement is that we only need one for-loop structure in CPU side to loop through forward elimination rows, compared to three for-loops vanilla gaussian elimination. On device side we don't need any looping at all since all the calculations are done in parallel. There only needs to be a sync_thread in the end to ensure all threads are at the same point before moving on to next row.

The results will be shown below. One defect of my implementation is load-balancing. Since GE is processed in a triangular-like manner, when we are on lower rows the memory for blocks and threads on upper rows will be wasted. This might result in worse performance. One possible optimization is to assign different thread size for each block each iteration. But I did not see much improvement for that method.

3.2 Observation from the Matrix size

As we can see from the graph below, runtime for CUDA version of Gaussian elimination increases steadily as matrix size increases, while CPU GE starts to climb after a certain matrix size (around 1,700). When matrix size is 4096 we can see CUDA version is half the runtime. This is because in CUDA thousands of threads are running, so it performs well on large matrix sizes. Meanwhile, serial runtime increases quadratically. We can anticipate that when matrix size is even larger the runtime difference will also be huge. However, when matrix size is quite small CUDA is sometimes even slower than serial version. This is expected because creating/assigning threads have costs and when matrix is small the cost is too much compared to calculations.

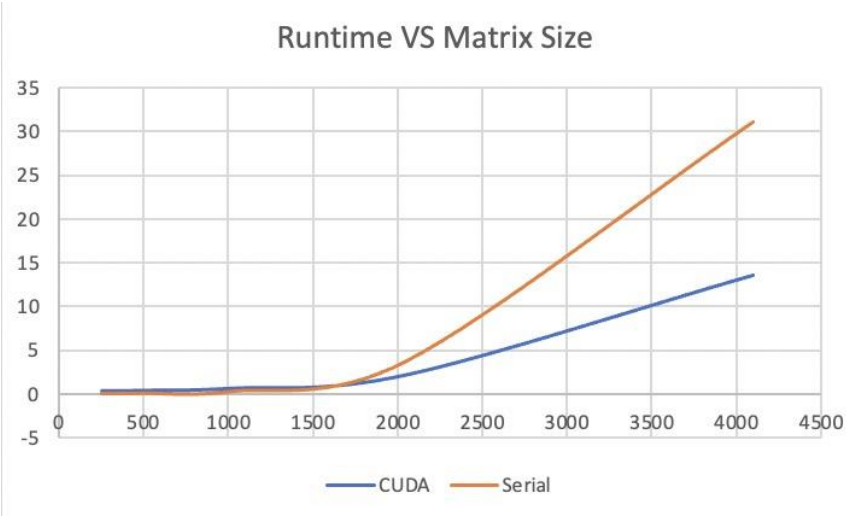

Figure 1. Runtime (second) vs. Matrix Size

3.3 Observation from Threads number

For our implementation, the thread number of Gaussian Normalization is NUM_THREADS, the threads for Gaussian Elimination is dim3(NUM_THREADS, NUM_THREADS). The domain for the NUM_THREADS is (0,32], since the total thread number for a single block could not exceed 1024.

1) As we can see from the line chart below, the runtime decreases tremendously as the thread size increases from 1 to 10. This is easy to explain since we increase the level of concurrency. After this point, however, we could not observe any improvements and runtime is steady at around 10 seconds. I think this is because the server has reached maximum capacity of parallelism at a certain point, so increasing thread number could not help much.

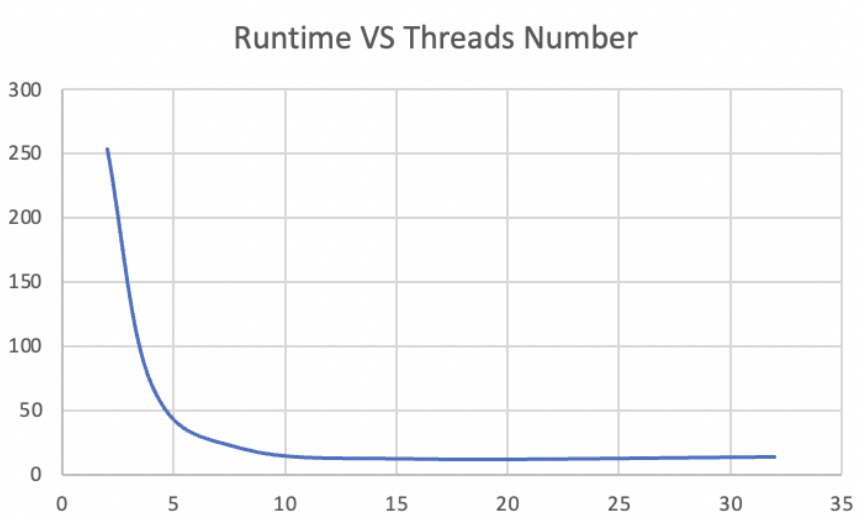

Figure 2. Runtime (second) vs. Thread number

3.4 Observation from Optimization Setting

Unlike the previous findings of Open_MPI and pthreads, the optimization settings show minor runtime improvements for CUDA. This is because -O1, -O2, -O3 can only optimize CPU side. But for CUDA version, most of the work are done on GPU.

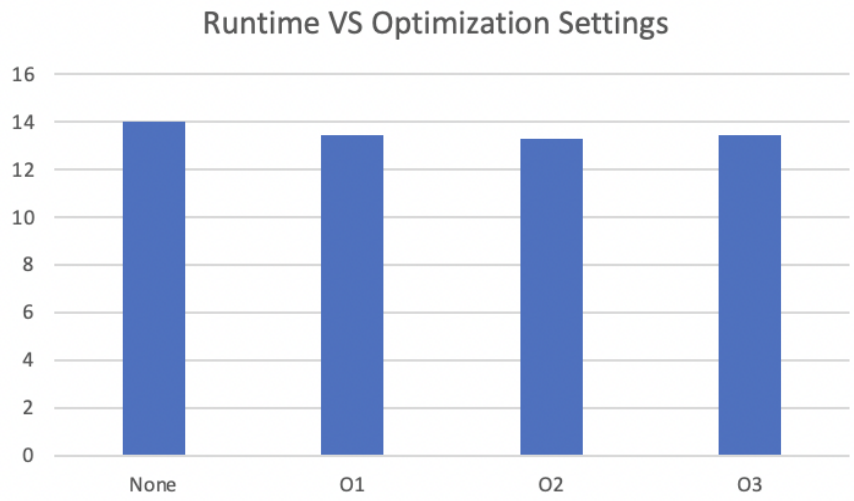

Figure 3. Runtime (second) vs. Optimization Settings

4. Compile and Run

Compile: *nvcc -O2 assign3_cuda.cu -o a3*
Run: *./a3 4096 32*
Best result: 13.75782 second.