CS426 Project 4

Barış Can 21501886

1 Questions

1. GPU hardware includes many processors which use SIMD. Therefore, when threads run on processors, they operate on independent data and use Single Instruction Multiple Threads (SIMT). However, the efficiency of the data-parallel application may be affected if the threads in the same warp take different control paths while executing the same instruction. This is called control flow divergence. For example, the execution of the branch instruction may cause the control flow to diverge since the statement of the if clause may not be true for all of them and some threads may go to the else part. It comes from a design decision to devote more resources to compute units than control flow handling. Both hardware and software solutions for the reduction of control flow divergence exist.

2. Dynamically allocated shared memory allows us to change the amount of shared memory per kernel for each launch. It can be obtained in two ways. You can pass a third parameter in the kernel function-calling aside from the number of grids and block size for dynamically allocate the memory such as $<<<\ 1,\ 512,\ N * sizeof(int)\ >>>$. Here, the third parameter represents N integer bytes shared memory per block which triggers $extern\_\_\_shared\_\_\_$. It can be also declared explicitly as above. If there are different types of data in the dynamically shared memory such as some parts of it double and and the other parts int, then pointers should be used while accessing and changing the data.

3. Because it is on-chip, the communication speed and latency of the shared memory is much faster than the local or global memory. The shared memory is allocated per block, so only the thread within the same block use the same shared memory. The shared memory can either be declared like $\_\_\_shared\_\_\_$ to created statically shared memory or $extern\_\_\_shared\_\_\_$ to create dynamically shared memory (or as expressed in Q2). Then, the threads within the same block may use the same shared memory. However, there could be a race condition since the threads in the same block may also be in the different warps so $\_\_syncthreads()$ should be used to synchronize the threads at required locations.

4. The device properties can be learnt by the function cudaGetDeviceProperties(&devProp, 0); and my device properties can be seen below.
   Name: GeForce GTX 850M
   Total global memory: 4294967296
   Total shared memory per block: 49152
   Total registers per block: 65536
   The warp size: 32
   The maximum memory pitch: 2147483647
   The maximum threads per block: 1024
   The maximum dimension of block 0: 1024
   The maximum dimension of block 1: 1024
   The maximum dimension of block 2: 64
   The maximum dimension of grid 0 2147483647
   The maximum dimension of grid 1: 65535
   The maximum dimension of grid 2: 65535
   Clock rate: 901500
   Total constant memory: 65536
   Major revision number: 5
   Minor revision number: 0
   Texture alignment: 512
   Concurrent copy and execution: Yes
   Number of multiprocessors: 5
   Kernel execution timeout: Yes

5. If you are working with Visual Studio, there should not be a problem with atomic intrinsics. However, in Linux or macOS, $-arch\ sm\_11$ flag should be added next to nvcc. For Tesla GPU's $-arch\ sm\_20$ also works and in some cases you can also specify compute capability as well like $-arch\ compute\_20\ -code\ sm\_20$. Also, the compute capability should be higher than 6.x.

2 Implementation Strategy

I started by reading the inputs and creating the arrays. I created two input arrays for the vectors A and B, and three output arrays for the dot product of A and B, the magnitude of A and magnitude for B which will be explained later. Then, I either generate random inputs or read from a text file according to the arguments of the application. I did not understand why we use the size of the vector at the start of the text file since we already know the size of the vector from the command prompt argument so I omitted it considering it should be the same with the argument. After initializing the inputs, I calculated the angle between two vectors with a serial implementation. I was not sure whether it will be the same if I give 1 as the number of threads so I implemented a serial code in the main function considering it will run on the CPU. The implementation was done by simply diving the dot product of A and B with the magnitude of A multiplied with the magnitude of B. Then, I moved on to the parallel implementation. I send my input, output arrays, N and block size to the host function, considering the outputs will be filled later, I created a for loop to combine each output coming from each block. The reason why I used two arrays for the magnitudes of A and B was since individual blocks calculate independent data, I was not sure where to combine them. They should be combined outside of the device and after the sync function so I decided to do this on the main function. Maybe there could be a better implementation where they are combined on the device which increases the speed of combining and reduces the number of output arrays to two which speeds up the implementation by sending less data for each communication but I could not find such way. After that, I calculated the angles and printed the results. The other disadvantage of using my algorithm is the angles are not exactly the same since I calculated the magnitudes of the vectors in the device function by

taking the square of them, adding them and taking the square root of them. However, the angles of CPU and GPU execution is slightly different. The reason I can think for this is since in the CPU, I calculated the values at the same place and in GPU, I send the part of the sum to the main function and combine them there which causes some floating-point error. I looked for many places which I thought the error may be there, but I could not find the part causing the error. I also divided all the data equally on the blocks since I thought there might be some remaining data while partitioning, but I could not find an error there as well. This is not a problem for small N such as 10000 or 100000, but the angle starts to differ by its third digit after the decimal point after 1000000. I also measured time for all of these steps which will be discussed in the following section.

In the host function, I created the device input arrays with the size N and output arrays with the size block size. I used Cuda's own time function which includes cudaEvent_t so I am more confident for GPU's timing results since I could not find a proper timing function for CPU in Windows. After creating the device memories, I passed them to the global function which is the interface between the host and the device using cudaMemcpy. I send the output arrays to the Host function and free the device arrays. In the global function, I calculated each part of the data using i which increases until all the data is calculated, and threadIdx.x since I have one dimension of blocks. Here, I find each value in the vector A and B according to the i and threadIdx.x, and calculated their dot product and magnitudes. While calculating the dot products, I called the device function to do the calculation. After finishing the calculations, the output arrays automatically send to the host and main function which gives me the results. While printing the results, I followed the guideline in the assignment.

3 Discussion

The serial implementation for finding the angle between two vectors is generally faster than the parallel implementation. I think the reason behind this is the size of the input arrays are still not enough to be profitable from parallel implementation and the number of thread is still not enough to calculate faster which does not compensate the communication between the threads. I also tried with 1024 number of threads and it gave better results for every experiment. However, you can see that the closeness of the serial and parallel implementation decreases while increasing the number of elements and the number of threads. Below, you can see the experiment done with 1000000 elements and different number of threads (Figure 1). It can be seen that the running time of the serial implementation is the same with the parallel implementation running with 512 threads which also shows there is no speed up between the serial and parallel implementation. Therefore, using parallel implementation for this case does not get any benefit.

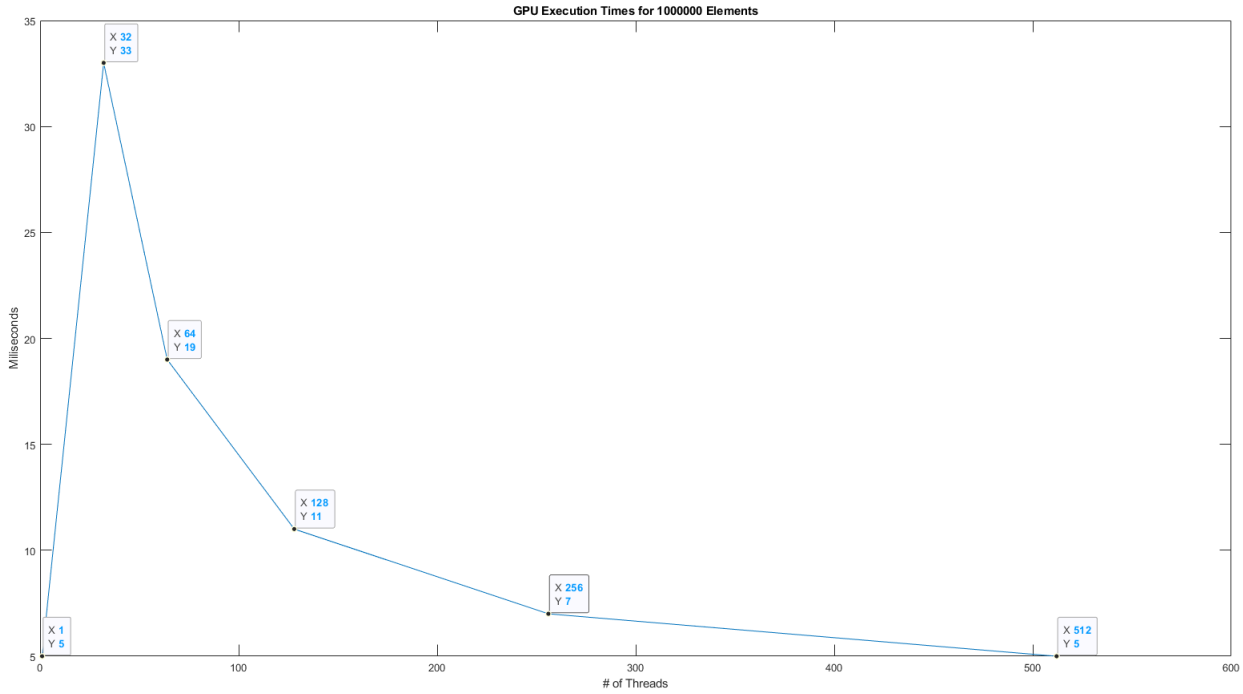

Figure 1: GPU Execution Times for 1000000 Elements

Below, it can also be seen that the parallel implementation with 512 threads is slightly faster than the serial implementation by two milliseconds for 5000000 elements (Figure 2), and 6 milliseconds for 10000000 elements (Figure 3)which offers about 0.875 speed up. The other reason which slows down the parallel implementation is the communication cost for sending the input and the output arrays from device to host and from host to device. Considering the device function time is faster when there are more than 128 threads, but because of sending inputs and getting outputs, the parallel execution faster than the serial execution only when there are more than 512 threads. While the device execution gets faster with the increasing number of threads, since the number of elements is increasing, overall GPU execution is also increasing.

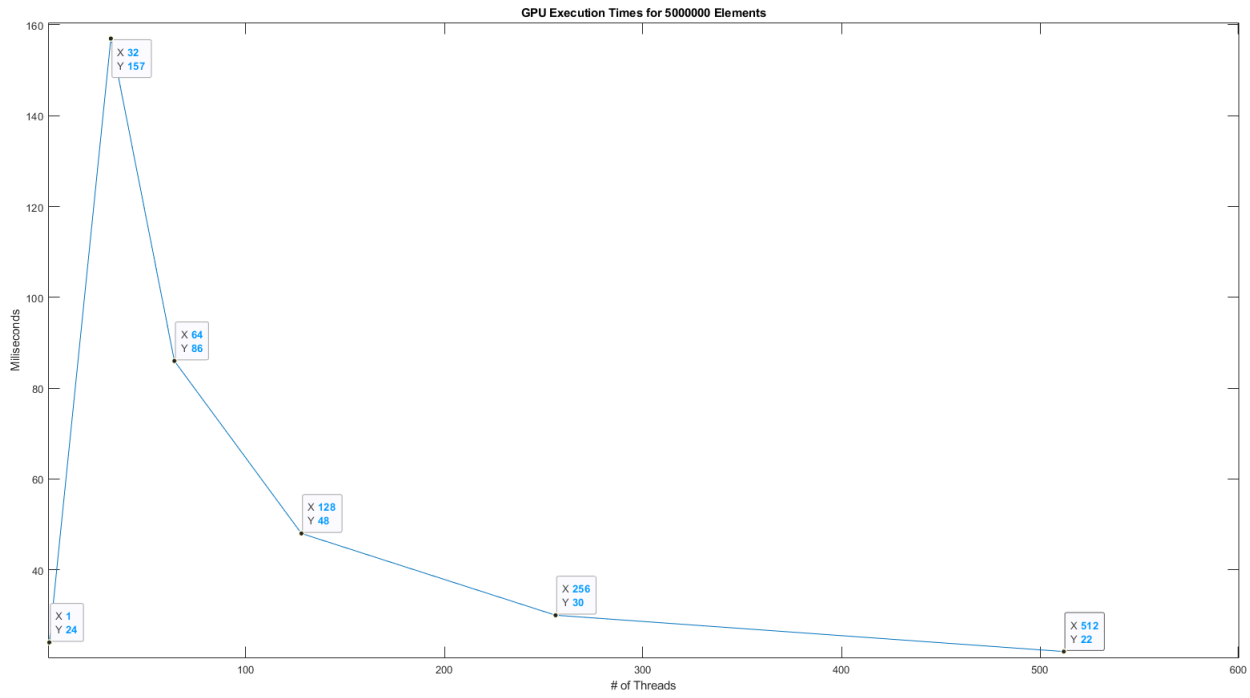

Figure 2: GPU Execution Times for 5000000 Elements

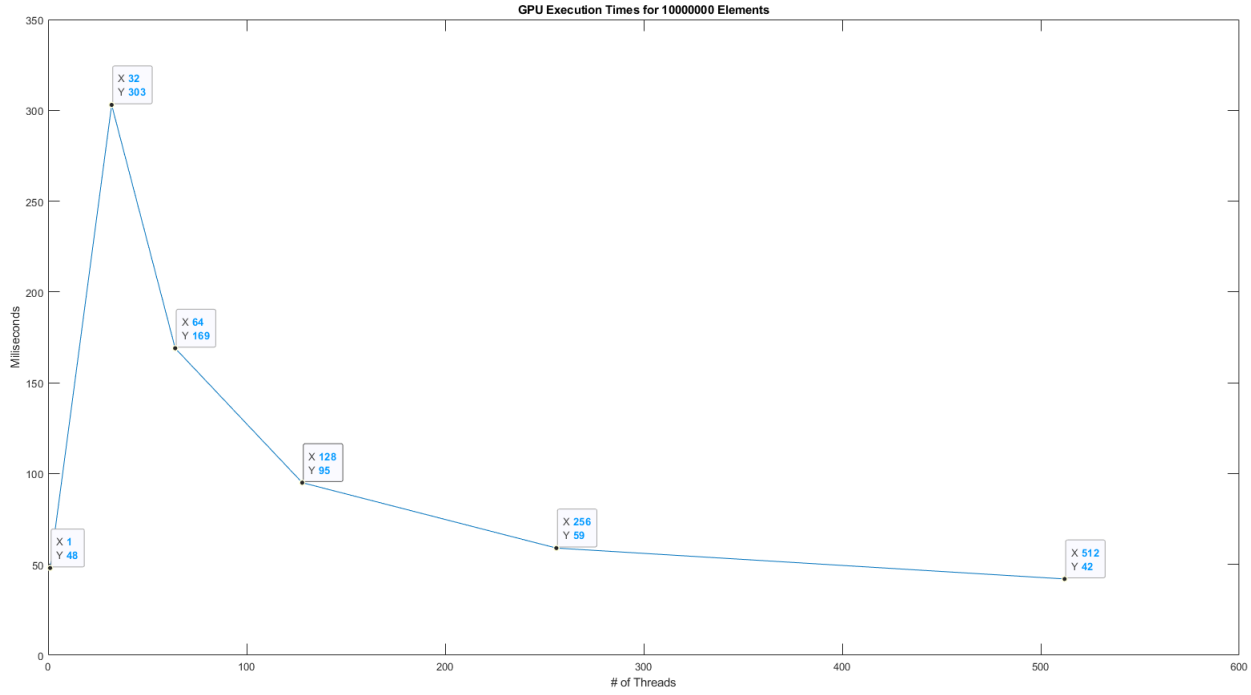

Figure 3: GPU Execution Times for 10000000 Elements