CS 426 Parallel Computing Project 4 Report

Bartu Atabek - 21602229 29/05/2020

1 Answers to The Questions

1.1 What is control flow divergence?

Flow divergence also known as the warp divergence, is meant for a performance issue while in kernel functions, if we meet a conditional branch like if, different threads of the current warp can enter different sections of the branch which damages the parallelism since it is wanted for warps to act parallel. If different threads go in the same branch, it is very efficient otherwise, performance decreases significantly.

1.2 How can we create a dynamic sized shared memory?

Creating dynamic sized memory is similar to creating static sized memory but it has little differences. While creating dynamic sized memory, we put at the start of the initialization line **extern** keyword, then **__shared__**, then the type of the memory and then name of it. Below line holds an example declaration.

Example: **extern __shared__ float shared_data[]**

We do not specify the size of it, as it is dynamic We can use as much as space that the caller passed as parameter to global function. It is passed as the third argument of the **"< < < > > >"** block

1.3 How can we use shared memory to accelerate our code?

The shared memory is much faster than the global memory as it is just on the same chip GPU resides. It is almost 100 times faster than the global memory. Shared memory is spared for all the threads in current function. They all can read and write to it so we should be careful while using it. As it is faster, we should use it but we should be using it in a safe way. For example, two threads should not update the same location at the same time or one should not read the location while other one is writing at the same time. These kinds of issues can be handles in some ways in CUDA. One of them is to use atomic operations like atomicadd and atomicsub.

1.4 Which CUDA operations give us device properties? To answer this question you should write a simple program and query the device properties of the machine you are working with.

In CUDA, there is a data structure which is called, cudaDeviceProp, it can be initialized with the function cudaGetDeviceProperties. After then, we can easily obtain the information about our CUDA device as we want. For example, after initialization of cudaDeviceProp, we can obtain our devices total global memory as prop.totalGlobalMem or we can learn maximum threads per block as pro.maxThreadsPerBlock

1.5 What are the necessary compiler options in order to use atomic operations?

In CUDA, to use atomic operations, our compiled binary architecture should match the device architecture. To check this, we should look at our device architecture code from NVDIA website or from device properties. This architecture type should match the one in the Project properties > C/C++ > Device > Code Generation > compute.....

2 Results & Discussion

2.1 Implementation Details

In my implementation, I firstly do the basic things of every CUDA program. Allocated memory for host variables and kernel variables, memcpy my host variables to kernel ones. Then I call the **__global__**function to execute my kernel code. I passed 4 parameters to it, the first two ones are the 2 vector arrays. I create them in the function arrayGenerator before calling the **__global__** func. I pass N to my array generator function and it returns me a 2 arrays which have size of N and every member of them has a random value between 0 and 100. In my **__global__** function, I basically applied the very similar approach to find firstly, the doc product of the 2 vectors, then, their individual, length by summing up their values to the power of 2 and then square rooting at the end. I output 3 values from this function, dot product,sum of power 2 of each vector. This was enough. Then, I divide the dot product into the multiplication of sqrt of each vector's length. In my CPU function, I again find firstly the dot product of two vectors and their individual power 2 and sqrt of them and their division.

2.2 Results as Plots

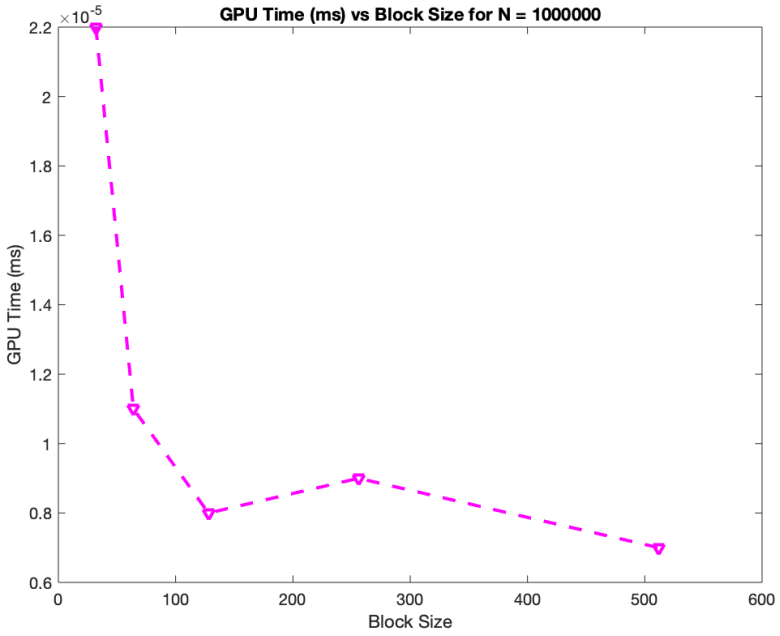

Figure 1: When N is constant as 1000000 and blocksize is 32, 64, 128, 256, 512. Time in nanoseconds

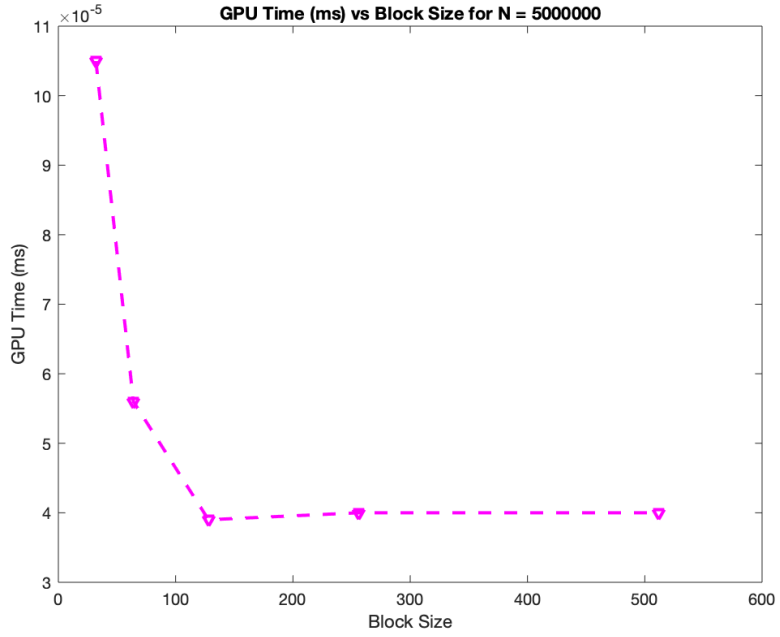

Figure 2: When N is constant as 5000000 and blocksize is 32, 64, 128, 256, 512. Time in nanoseconds

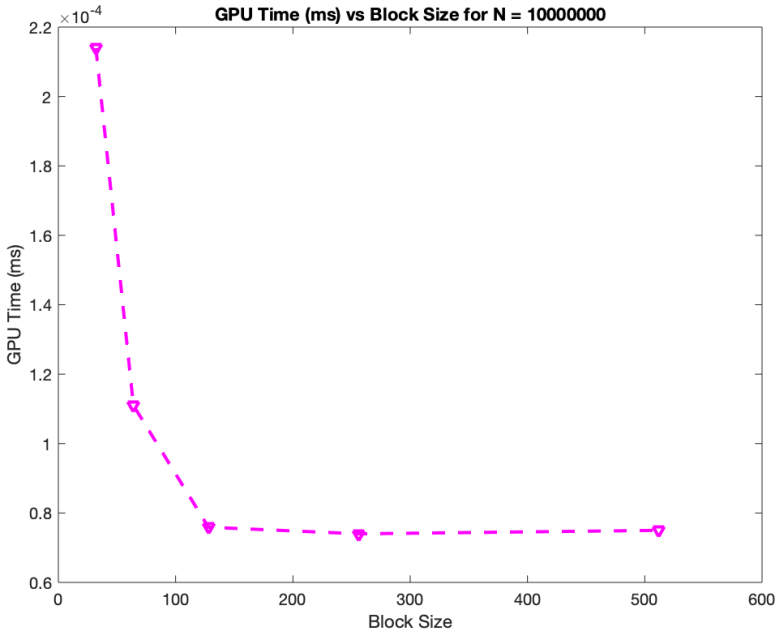

Figure 3: When N is constant as 10000000 and blocksize is 32, 64, 128, 256, 512. Time in nanoseconds

2.3 Discussion of Results

In the project, as it can be seen in graphs, the elapsed time for GPU is decreased when the block size increased. However; after some point, after 128, increase in number of blocks did not affect the elapsed time much, it may be because of the fact that we reached an almost upperlimit for increase because while going from 32 to 64 and 64 to 128, decrease was very high but then it almost stopped. Also, when the size of the vector is increase, the time is also increasing.

As a disclosure I can say that there may be a small index difference between the CPU results and the GPU results. I am not entirely sure the reason behind this because I tried different algorithms and still experienced this problem. I also tried the kernel code in the official Nvidia documentation. I could not solve this as I go over the code and it seems correct. There is probably a tiny index error or something like this in the code.