CS426 Fall 2017

Project 4

Alba Mustafaj

21500009

Table of Contents

1. Questions

a. What is control flow divergence?

Divergence, as it is defined, means trend to be different or develop in different directions. In terms of CUDA, it indicates a performance issue of the control flow of instructions. The threads are supposed to run concurrently on GPU for a maximal parallel work. Thus, to have a high performance, threads must follow a particular execution trajectory by executing same instruction at the same time. However, when an if clause is present, the threads fulfilling the if clause will do something different from the thread fulfilling the else clause. Thus, branching for conditionals is a case of divergence and executing in a divergent control flow might result in the lack of efficiency. This lack of efficiency is attributed to the fact that SIMD implementations cannot perform diverse instructions in same time. CUDA platform has decided the workaround for divergence, so the "if" part first is executed firstly, and after that the else part. But while the first part is executed, the else part threads are deactivated. so they are not executed in parallel, but in serial. Thus, in control flow divergence case, threads lose performance by threads not executing concurrently.

b. How can we create a dynamic sized shared memory?

Dynamic sized shared memory is necessary when the amount of shared memory unknown at compile time. The way to allocate dynamically in shared memory is by declaring a variable in the shared memory as external. For instance: extern _shared_ float sharedArr[];

So, the size of array is determined at launch time and all variables declared like this start at the same memory address. Therefore, the layouts of array variables should be managed with offsets. Also, pointers should be aligned to the type they point For instance the following can be done:

float * arr1 = (float*) sharedArr;

float * arr2 = (float*) arr1[64];

Furthermore, to call a global function in this case a third configuration parameter must be used to specify the shared memory allocation size per thread block. For example : someKernel <<< 2, 10, sh>>> (d_f);

where sh has type size_t (for instance sizeof(float)). So the external array uses memory allocated in this way.

c. How can we use shared memory to accelerate our code?

For a better performance, it is reasonable to increase bandwidth. To achieve this, the shared memory is divided into memory modules called banks which have equal sizes and can be simultaneously accessed. Thus, if there is a certain number of distinct banks n, the bandwidth obtained would be n times greater than that of a single bank. This advantage also has a disadvantage that is repaired. If addresses requested by many threads map to the same bank, the access is serialized. Therefore, conflicting memory requests are split by the hardware into as many conflict-free requests as needed, thus resulting in a decrement of the bandwidth. This decrement is equal to the number of colliding requests done to memory.

d. 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.

There are 2 main functions used for getting device properties:-
cudaGetDeviceProperties(cudaDeviceProp prp; int, device); //puts properties of a device in prp- cudaGetDeviceCount(int nrOfDevices); //get total number of devices.

Firstly, we can obtain the number of devices and then for each of them we can call cudaGetDeviceProperties function and print them accordingly. For instance:

```
#include <stdio.h>

int main() {

  int nrOfDevices;

  cudaGetDeviceCount(&nrOfDevices);

  for (int device = 0; device < nrOfDevices; device++) {

    cudaDeviceProp prp;

    cudaGetDeviceProperties(&prp, device);

    printf("Device Number is: %d\n", device);

    printf("Device name is: %s\n", prp.name);

    printf("The memory Bus Width is %d  bits\n", prp.memoryBusWidth);

    printf("Warp size: %d\n",  prp.warpSize);

    printf("Maximum memory pitch: %u\n",  prp.memPitch);

    printf("Memory Clock Rate is  %d KHz \n", prp.memoryClockRate);

    printf("Maximum threads per block is  %d  \n", prp.maxThreadsPerBlock);

    printf("Peak Memory Bandwidth is %f  GB/s \n",
       2.0*prp.memoryClockRate*(prp.memoryBusWidth/8)/1.0e6);

    printf("Number of multiprocessors is  %d  \n", prp.multiProcessorCount);

  }

    return 0;

}
```

This was the output I got:

```
Device Number is: 0
Device name is: Tesla K20c

The memory Bus Width is 320  bits

Warp size: 32

Maximum memory pitch: 2147483647

Memory Clock Rate is  2600000 KHz

Maximum threads per block is  1024

Peak Memory Bandwidth is 208.000000  GB/s

Number of multiprocessors is  13

Device Number is: 1

Device name is: GeForce GTX 480

The memory Bus Width is 384  bits

Warp size: 32

Maximum memory pitch: 2147483647

Memory Clock Rate is  1848000 KHz

Maximum threads per block is  1024

Peak Memory Bandwidth is 177.408000  GB/s

Number of multiprocessors is  15
```

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

There is a certain way to compile the code in order to use atomic operations in cuda. The following is used:

-*arch compute_xx.*

The virtual architecture for which the compilation will occur is illustrated by the xx part which is separated from compute with a _. Those xx are substituted by numbers in compile time. In order to imply a real architecture sm_xx can be used instead. Furthermore, the system will define the constants for atomic usage.

2. Programming Assignment

2.1 Implementation Details

In the main function which executes in CPU I take the necessary information from argv. To separate with which of the two cases the program will proceed, I check the number of arguments with argc. If there are 3 arguments it means the text file is not given (1 arguments is always there, 2nd argument is size, 3rd argument is bloc ksize), thus I continue generating random numbers in range 0- 10000 for both arrays. If there are 4 arguments (including text file) I take the arrays' data by reading from the text file with the read function which takes data line by line. I also declare additional arrays to hold the dot product and lengths of both vectors, since the formula of finding the angle is based on finding its cosine.

$$\cos \alpha = \frac{a \cdot b}{|a| \cdot |b|}$$

I also declare the arrays which will be passed to the global function to make the necessary computations. I allocate the main arrays dynamically by making use of the size obtained from argv[1] and also allocate the above mentioned dot products and vector lengths arrays with size equal to the product of block size and block number, because each of the threads will make use of them and store the computations done in the kernel. Since both CPU and GPU versions of the code are required, I firstly perform the CPU version, by using 3 additional doubles (instead of the arrays used in GPU version) which inside of a for loop bounded by size, add up on each iteration as following:

dott (dot product storing double): keeps the dot product of each index : dott += (arr1[c] * arr2[c]);

sqr1 (length of first vector): firstly data in each index is squared then summed and in the end the square root is found (based on the length formula).

sqr2(length of the second vector) same computations as the first vector.

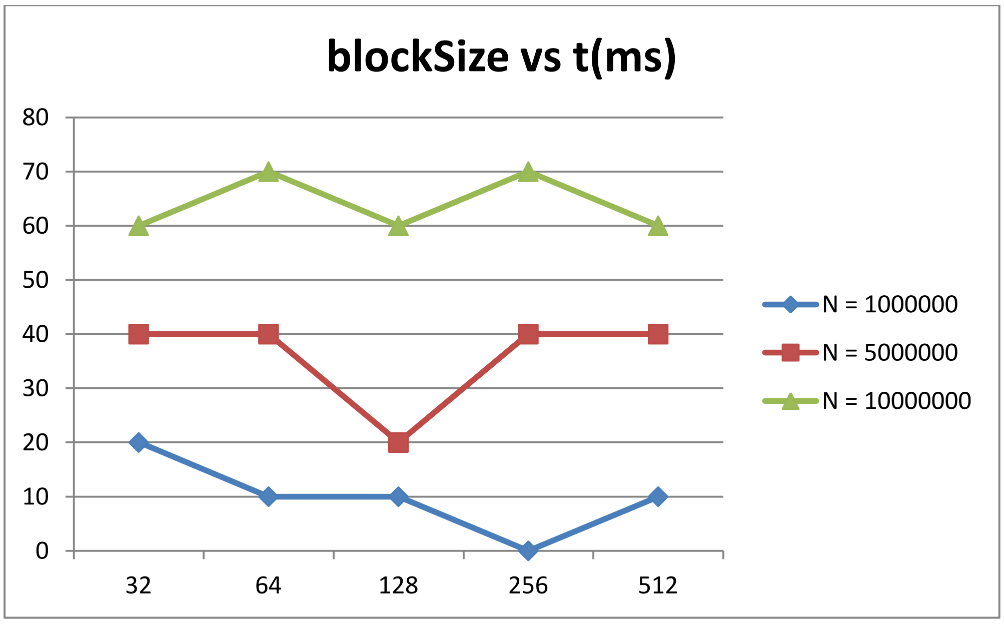
After these computations I find the ratio of dot product with the product of lengths of both vectors which gives the cosine of the angle we are requiring. To find the angle we take arccosine of the value then multiply it by 180 and divide by pi (3.14) to obtain the approximate value in degrees. This is the end of competing this exercise in CPU only.

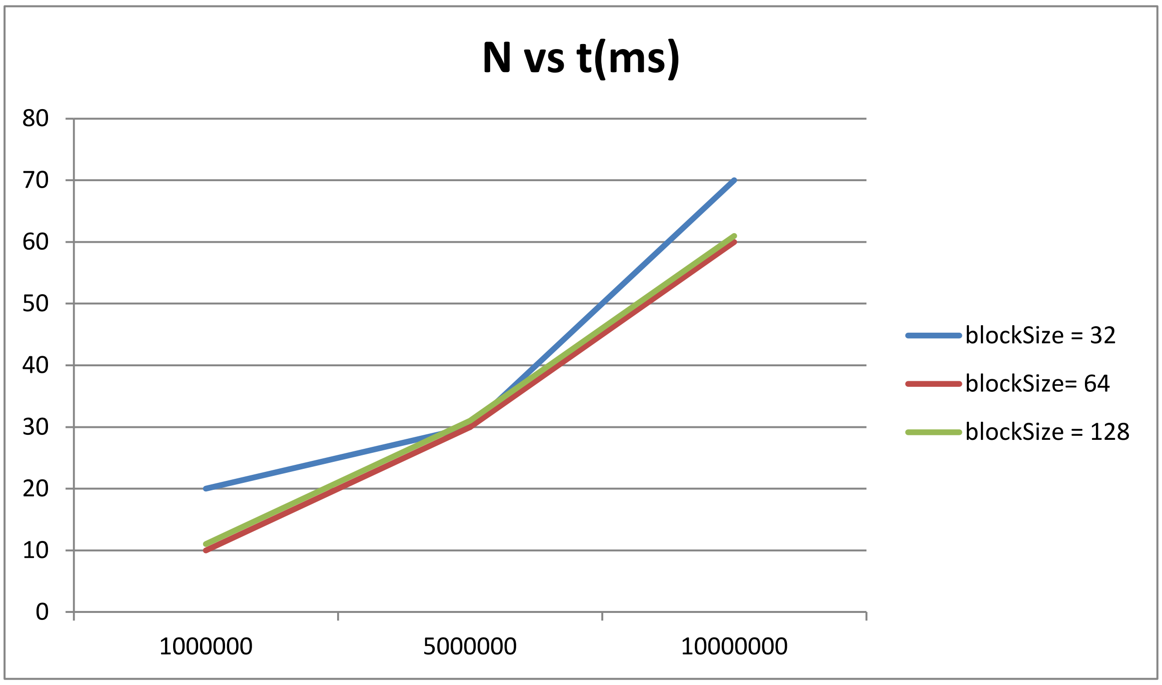
As mentioned above, I do the corresponding memory allocations with cudaMalloc for device arrays and additional arrays helping to compute dot product and length, then copy the 2 vectors from host to device. The data taken by each thread is also calculated by dividing the array size with the product of block number and block size. I pass this value to the kernel parameters along with the 2 main arrays (vectors) and 3 additional ones (dotProduct, length1, length2) and the block size. In the compute function (kernel) the particular index to be modified by each thread is calculated (recalling storing data in row major) so to assign threads accordingly with the array indexes the threadIdx.x is added to the product of block size with blockIdx.x. Furthermore, in a for loop bounded by data to each thread (clarified above) the dot products and lengths are calculated by each thread. After completing this task, and before copying from device to host, cudaThreadSynchronize() is called to make sure the threads have completed their work and there is no inconsistency.

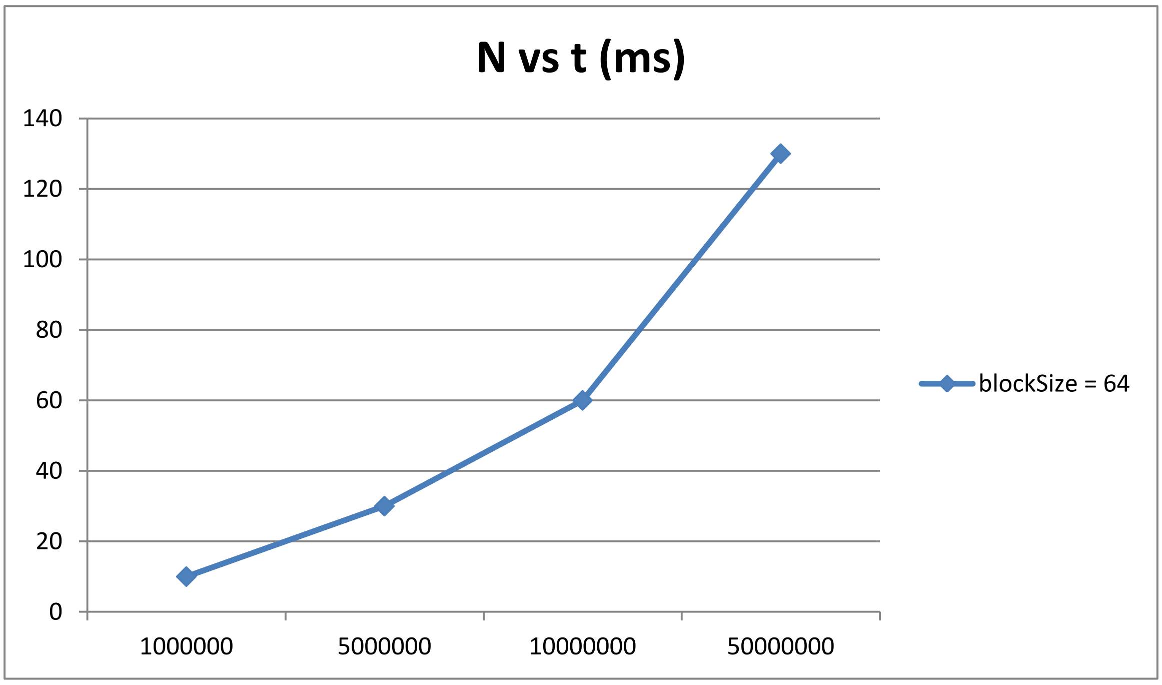
The dot product and lengths are retrieved by host and the cosine of the angle is found as explained above. Applying arccosine and radian to degree formulas gives the correct angle in degrees. While all this process occurs, times required in the project are also computed and displayed.

Note: I have also considered the case when the size is not divisible by blocksize * blockNumber, so I assign the remaining computations to master.

2.2 GPU timing graphs




N vs t(ms)
80
70
60
50
40
30
20
10
0
1000000
5000000
10000000
blockSize = 32
blockSize= 64
blockSize = 128


N vs t (ms)
140
120
100
80
60
40
20
0
1000000
5000000
10000000
50000000
blockSize = 64

2.3 Discussions & conclusions

As observed firstly in the outputted results after running the script, the function on CPU takes significantly more time than performing it on GPU. This is due to sharing work among threads and each computing concurrently without depending on each others' data. When finding angle is done in CPU all the computations were done serially thus causing a long time elapsed, while in presence of a kernel, the tasks are parallelised and take fundamentally less time to compute. Transferring data between host and device does not take long and it is much less than time elapsed for the CPU function. Ideally as well as proven concretely in this project, the involvement of GPU is a high advantage to the performance.

Regarding the GPU implementation part and also considering the 3 graphs depicted above, it can be deduced that time elapsed for host to device transfer + kernel execution, which make the total execution time for GPU are relatively small even when calculated for big values of N and with a precision of milliseconds. In the first two graphs, the time elapsed for different block sizes (as N is kept constant) is observed. The time elapsed is so small that we may even conclude that it is not changing. The slight changes of 10-20 milliseconds can be attributed to internal errors in rounding and computation and it can be said that they are within the error margin. As can also be deduced from graph 1 where time elapsed is computed for different N, being that the values are relatively small and close when N is kept constant, it can be deduced that the times elapsed for different block sizes are very similar varying with 10-20 milliseconds. However, as N increases, so does the elapsed time. For instance, when N = 10000000, it is obvious that time elapsed for it is larger as it's graph is also depicted above those of smaller N.

I plotted the second graph to give a better insight on how time elapsed changes for different block sizes when N is kept constant. As observed by the results the lines are very close to each other and two of them are even overlapping each other, which means that for a certain value of N, even if the block size changes, the time elapsed is not altered significantly.

The third graph gives an insight on how time elapsed changes while block size is kept constant and N is increased. Regarding the depicted results of graph 3, I observed a parabola-like shape of graph as N increases. I intentionally tried with a very large value of N to have a clear insight on the results as the times until 10000000 were not larger than 30 milliseconds. Thus, it can be concluded that as the size of vector increases and the block size is kept constant, the time elapsed increases as well. Of course, also changing the number of blocks for different N will give varying results, but the third graph was made to illustrate the concrete relation between the bloc ksize and N (although it can be logically deduced).

All in all, GPU enhances the performance significantly. It is important to make the proper allocations and assign the work that can be parallelised to the global function (kernel) as threads will execute concurrently what would be executed serially in the host (CPU). Execution time for GPU is relatively and fundamentally smaller than CPU even when calculated for big values of N in a precision of milliseconds, making it favourable for large and complicated computations. I observed in my implementation that as N is kept constant, the time elapsed for different block sizes is relatively close and very small. As the total time elapsed from GPU usage is relatively small in this project, it supports the above arguments as well as implies that a great precision is needed to calculate the real benefit or loss in performance while N and block size is changing.