Angle between two N-Dimensional vectors

Alemdar Salmoor

May 2020

1 Questions

1.1 What is control flow divergence?

GPU runs multiple thread blocks simultaneously on the same multiprocessor. To run hundreds of multiprocessors concurrently it employs SIMT(Single Instruction, Multiple Thread) architecture. The instructions are pipelined. Unlike in CPU cores, in GPU there is no branch prediction or speculative execution [2]. The multiprocessor within GPU manages threads in groups of 32, that are called *warps*. Even though each thread within a warp start at the same program address they have their own program counter and register state and free to branch independently. Since each thread within a warp execute concurrently, maximum efficiency is reached when all of the threads within a warp agree on the path taken. When conditional branch occur, threads that took one branch path are executed while the threads that took another branch path are essentially put on hold until the threads from the first path are executed. This essentially serializes the execution.

1.2 How can we create a dynamic sized shared memory?

Shared Memory is allocated by the $\_\_shared\_\_$ memory specifier in the kernel function. When calling the kernel function from the host additional argument is passed in the execution configuration of the kernel function which signifies the size of the shared memory in bytes per block.

As an example, host function:

```
int main(){
...
kernelFoo<<<nBlocks, nThreads, size>>>();
...
}
```

where the *size* argument is the size of the shared memory in bytes. In the following example *myArr[]* is of the size which was set during the call from the host function. We also declare shared memory with the keyword *extern* to signal that the size of it is determined during the launch time, i.e. dynamically.

```
__global__ void kernelFoo(){
...
extern shared int myArr[];
...
}
```

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

First of all, since shared memory is on chip its faster than the global memory (100x lower latency by the estimates)[4]. Because all of the threads within a block have access to the the same shared memory, threads can benefit from using data in the shared memory that was loaded from the global memory by the other threads in the block. Also, shred memory can be a fast communication mechanism for threads within a block.

1.4 Which CUDA operations give us device properties?

We can access device properties using *cudaGetDeviceProperties()* function. The following lines of code query some of the device properties:

```
int main(){
...
cudaDeviceProp p;
```

```
cudaGetDeviceProperties(&p, 0);
printf("Max shared Memory per block: d\n", p.sharedMemPerBlock);
printf("Max Threads per block: d\n", p.maxThreadsPerBlock);
...
}
```

In total *cudaDeviceProp* structure has 25 data members that can be queried in a similar way as above.

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

The minimal needed compute capability to use atomic operations is 1.1. Therefore we should include *-arch sm_11* when compiling. As it appears that Nvidia gradually ceases the support of devices with Compute Capabilities lower than 3.0 which their CUDA Documentation page reflect, I used *-arch sm_30*.

2 Implementation details

Two vectors $\mathbf{a}$ and $\mathbf{b}$ and the angle θ between them are related by:

$$\cos\theta = \frac{\mathbf{a} \bullet \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

Rearranging for the angle and using the definitions of the length of the vector, the dot product, and product property of the square root, we can rewrite the above as:

$$\theta = \arccos\left(\frac{\mathbf{a} \bullet \mathbf{b}}{\sqrt{(\mathbf{a} \bullet \mathbf{a})(\mathbf{b} \bullet \mathbf{b})}}\right)$$

This representation helps us to see that in order to find the angle we need to compute three dot products. This description is enough to implement the serial version.

For the parallel CUDA implementation we would like the threads to perform the dot products. At the end of the parallel CUDA code we would like to have three values that correspond to the dot products. We then use them to find the angle as defined the by the formula above.

Each thread will work on the set number of integers, let us call it ***ints_per_thread***. The total number of integers in the array may not be an exact multiple of the total number of threads. As we learned in the item 1 in Part 1, we would like to avoid branching of the threads, also we would like the workload among the threads to be as equal as possible. Therefore we will have an array sizes that is a multiple of the total number of threads, whose computation will be shown shortly. We will pad the actual arrays with 0's in the end in the positions created by the extension. Please note that $\mathbf{x} = \langle x_1, x_2, x_3, 0\rangle = \langle x_1, x_2, x_3\rangle$. Therefore the correctness of our computation is unaffected. The extended size of the array is then:

$$\textbf{arraySize} = \textbf{number\_of\_blocks} \times \textbf{blockSize} \times \textbf{ints\_per\_thread}$$

where ***blockSize*** is given by the arguments during prorgam call. In our parallel code each thread works on the portion of the both vectors and contributes equal amounts of work for the final dot products. As we can generalize our algorithm to parallel reduction, we can apply Brent's theorem [3]. If the original size of each array is N, then

$$\textbf{ints\_per\_thread} = log_2 M, \textit{ with smallest integer } i \textit{ such that } M = 2^i \geq N$$

The number of blocks is found by

$$\mathbf{number\_of\_blocks} = \mathbf{base} + \mathbf{\Delta blocks}$$

base is the minimum number of blocks to keep the GPU occupied.

$$\mathbf{base} = \mathbf{number\_of\_SM's} \times \mathbf{number\_of\_active\_blocks}$$

$$\mathbf{number\_of\_active\_blocks} = min(\mathbf{max\_resident\_blocks\_per\_SM}, \frac{\mathbf{max\_resident\_threads\_per\_SM}}{\mathbf{blockSize}})$$

Number of SM's, maximum number of resident threads can be found using device properties. Maximum resident blocks can be determined from major and minor revisions queried from the device properties. We find additional number of blocks by

$$\mathbf{\Delta blocks} = \lceil \frac{N}{\mathbf{blockSize} \times \mathbf{ints\_per\_thread}} - \mathbf{base} \rceil$$

where N is the original vector size. It can be seen that $\mathbf{\Delta blocks}$ can obtain negative values. This occurs for rather smaller array sizes. However, due to the ceiling function, **base** is at least **1** for any two real vectors. After this point we have all the values needed to compute the new size for our arrays that will be padded with extra zeros. In the end of this tedious calculations to find the number of integers per each thread, number of blocks and array sizes we accomplish 3 goals: 1. eliminate one of the sources of control flow divergence, 2. make sure that the GPU is occupied, 3. comply with Brent's Theorem so that our algorithm is scalable.

Let us define $U = \mathbf{a} \bullet \mathbf{b}$, $D_1 = \mathbf{a} \bullet \mathbf{a}$, $D_2 = \mathbf{b} \bullet \mathbf{b}$. Each thread t_i will compute its partial $U(t_i)$, $D_1(t_i)$, $D_2(t_i)$ values and store them in the shared memory within its thread block. Then all the threads in each thread block reduce their values to common locations (i.e. reduce to threads with ID 0). That is, after this step every thread block b_i will have computed its $U(b_i)$, $D_1(b_i)$, $D_2(b_i)$. Then every thread block contributes its partial values to the final global memory locations for each of the U, D_1, D_2 with the help of atomic operations. Then finally, we just plug the obtained U, D_1, D_2 to the formula presented in the beginning of this section to find the angle.

3 Timing

Figure 1: Total GPU Execution time

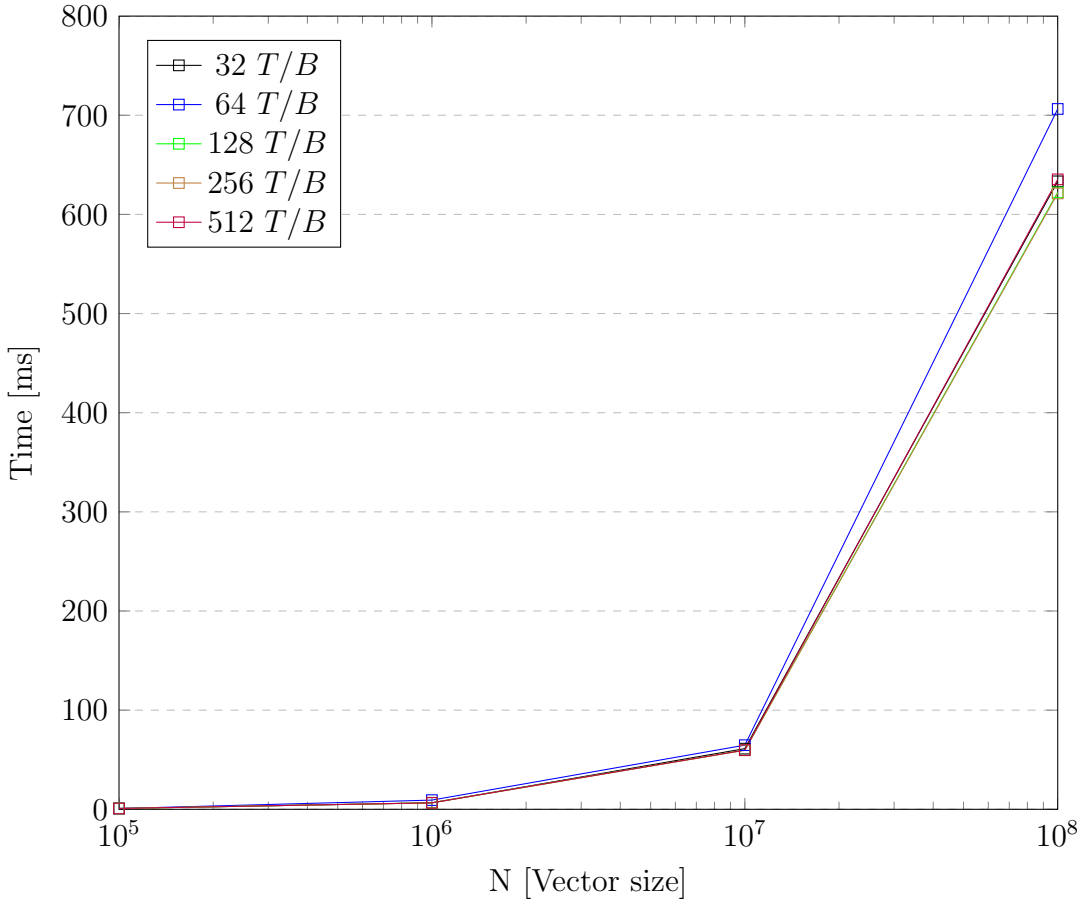


Figure 2: Kernel Execution time

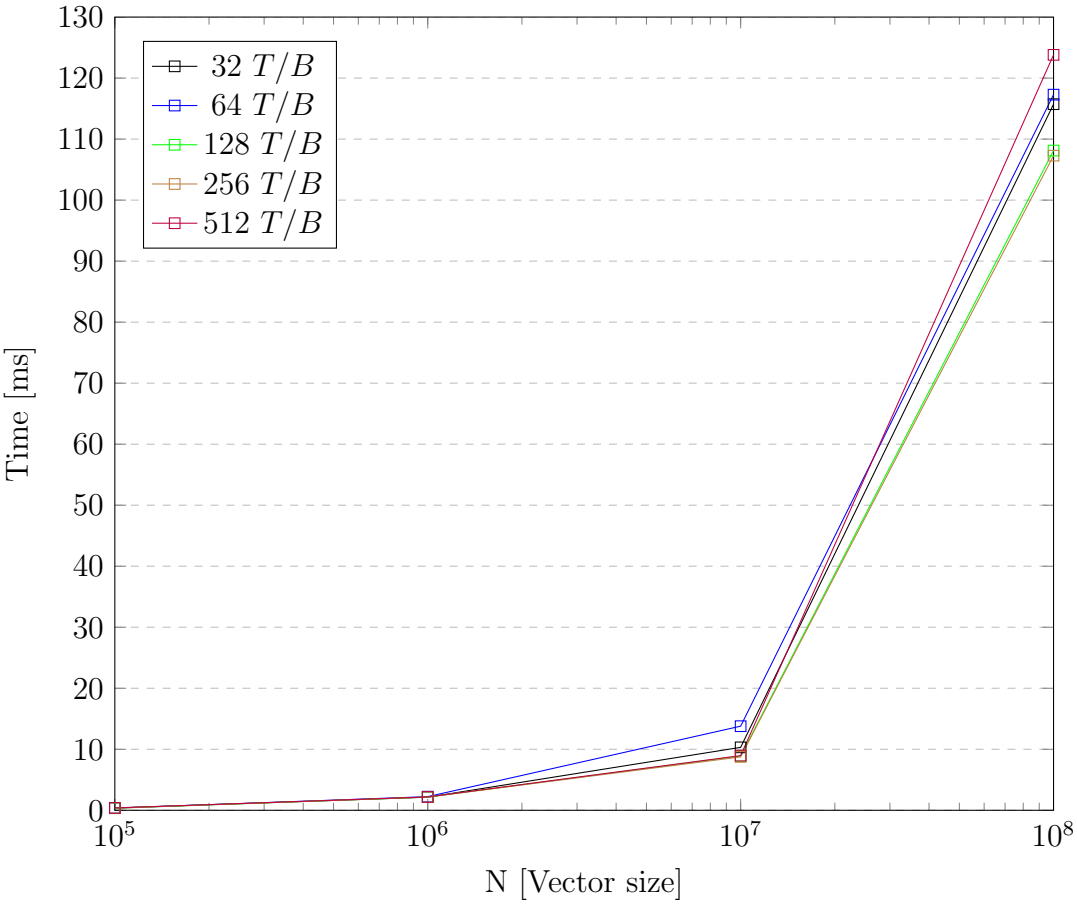
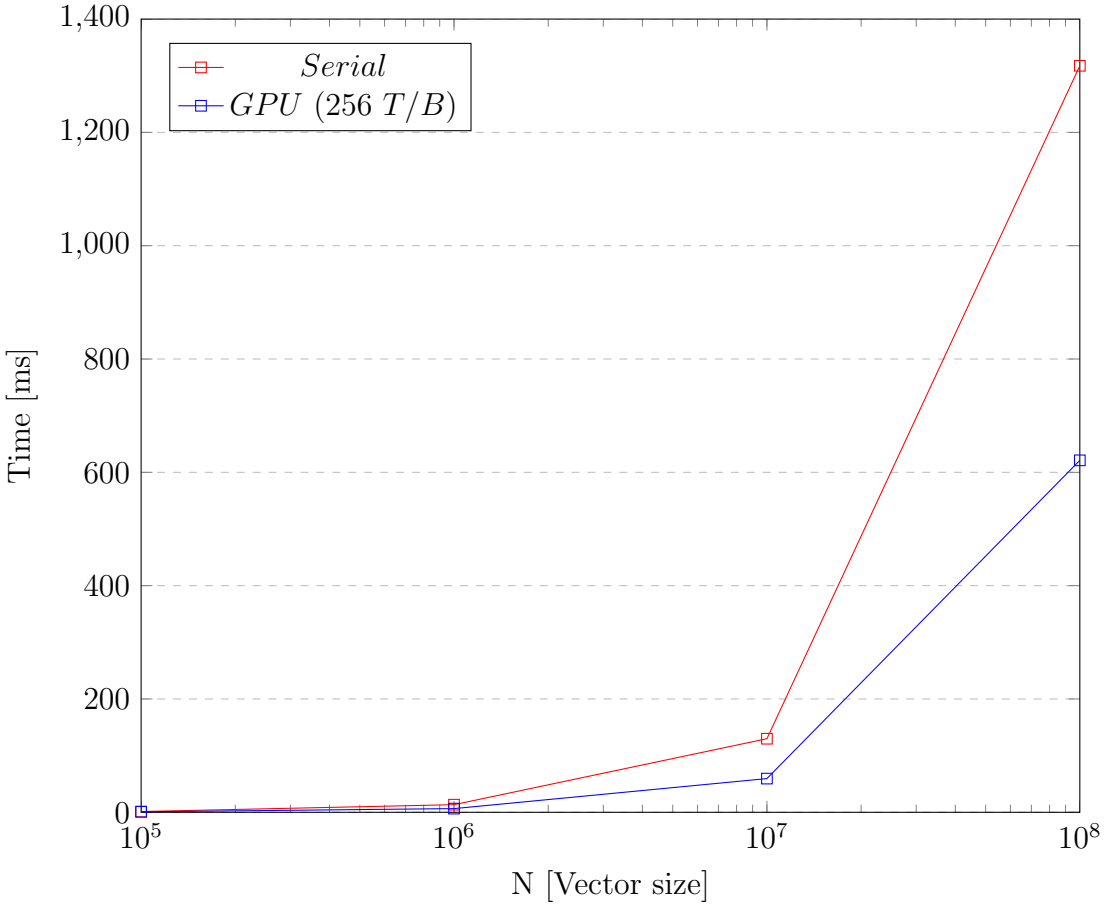


Figure 3: Serial Execution Time vs GPU Execution time



	$N = 100000$	$N = 1000000$	$N = 10000000$	$N = 100000000$
Speedup	1.6247	2.06373	2.18288	2.12112

4 Discussion

By looking at the Fig 1 and Fig 2 we observe slight improvements in terms of execution time as we increase the block size. With *256 Threads/Block* performing the best. This is due to the fact that more threads within the block can benefit from fast shared memory. However, execution time with *512 Threads/Block* seems to perform worst for $N = 10^8$. We can try to justify it by saying that number of created blocks for it is less than the other configurations so that memory latency can not be properly hid away with other thread blocks when threads try to access global memory. However, we should note that the discrepancy with other executions is not drastic and well within the acceptable bounds. These small differences in execution times between different configurations show the differences between theoretical Parallel Random Access Machine (PRAM) and real life parallel machine. In PRAM all of the configurations for the same N would have the same run time, however in real life parallel machines, in this case CUDA, we have to consider many variables such as Control Flow Divergence, number of active blocks, large global memory access latency, atomic operations on the same locations which serialize the execution, thread synchronization and many more. Looking at the Fig 1 and Fig 2 we can also note that the large portion of the GPU time is the time it takes to transfer data between host and device. From Fig 3 we see that execution time of the GPU is roughly the half of the serial CPU so that *Speedup* appear to converge at value 2 which we would expect as our algorithm is cost efficient.

Even though PRAM does not consider physical limitations of the real life parallel machine. I understand and value that, algorithms developed for it and theorems proven/shown on it produce close to optimal results when transferred to real life machines and that it is a job of a developer to fine tune according to the physical constraints of any particular machine.

To finalize we can consider this Nvidia document that points at 4 major reasons for *Low Achieved Occupancy* [1]. Since Occupancy of the GPU is important metric to achieve the best possible results we will now consider whether we have addressed those issues. *1. Unbalanced workload within blocks, 2. Unbalanced workload across blocks, 3. Too few blocks launched, 4. Partial last wave (Active Warps/Max Warps).* We addressed **1** by introducing the extra 0's in the end of the arrays so that every thread has equal

amount of work to perform however we have unavoidable asymmetry among the threads within a block in terms of work when it comes to reduction of the partial values to one thread and atomic operations. **2** holds as the amount of the total work per each thread block is the same. **3** may occur for smaller values of the vector sizes. **4** holds since we always started with maximum possible number of blocks that can reside on the multiprocessors and all of our configurations included at least 32 (Warp Size) number of threads per block.

References

[1] *Achieved Occupancy*. Nvidia. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[2] *Cuda Toolkit Documentation*. Nvidia.

[3] John L. Gustafson. *Brent's Theorem*. DOI: https://doi.org/10.1007/978-0-387-09766-4_80.

[4] *Using Shared Memory in CUDA C/C++*. Nvidia. URL: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/.