Erdem Ege Maraşlı
21602156

CS426 - Project 4 - Report

1. Questions

A)

Branch instruction execution may cause control flow divergence [1]. Control divergence occurs when threads in a warp take different control flow paths by making different control decisions such as branch decisions. If you are running multiple work items in SIMD mode, work items executing different code paths must be partly stalled, which results in reduced use of the SIMD modules and likely longer running time of that kernel [2].

B)

“extern __shared__ int s[];” instruction in a global function creates dynamic size shared memory for GPU. Its size must be specified when calling the global function.

C)

Shared memory is much faster than local and global memory since it is on-chip. Since shared memory in a thread block is shared by threads, it provides a cooperative mechanism for the threads so that we can accelerate our code. For exemplify, for a reduce sum operation normally we need an atomic add operation to avoid race conditions, however, if we are using shared memory we can do a reduction in the same block without an atomic operation because it makes the threads in the same block cooperative. Note we still need to use atomic add when we are reducing blocks but we don’t need when we are reducing threads in the same block.

D)

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);
```

These calls get the device properties of the first device attached to the system and put it in the prop variable. You can access its name by “prop.name” Memory Clock Rate (KHz) by

“prop.memoryClockRate” Memory Bus Width (bits) by “prop.memoryBusWidth” and Peak Memory Bandwidth (GB/s) by “2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6” calls. Example execution is given below.

Device name: Tesla P4
Memory Clock Rate (KHz): 3003000
Memory Bus Width (bits): 256
Peak Memory Bandwidth (GB/s): 192.192000

Note: I use Google Colab for testing.

E)

Atomic instructions become available when the appropriate architecture is specified to nvcc via --gpu- architecture [3].

2. Implementation Details

GPU uses the SIMD paradigm, which means the same operation executed in parallel for multiple data. The real purpose of GPU programming is to find and parallelize programs that perform the same operation on a huge independent data sets so that every operation can run parallel. I figure out that we need to calculate 3 dot products when we are calculating the angle between two vectors. Vec1.Vec2, Vec1.Vec1 and Vec2.Vec2 needed to calculated and remaining operations just calculated with O(1) time. Because of this, I calculated these dot products at kernel. I sent the data of Vec1 and Vec2 to the kernel.

First, I needed to calculate multiplication and then needed to perform reduce the sum to calculate the dot products. I use shared dynamic memory so that I do not need to use atomic add for the reduction in the same block. Multiplication is really straight forward and after multiplication, I reduce the whole threads in the same blocks. Straight forward implementation of this reduction causes highly divergent warps which is really inefficient. You can see different implementations for the reduction [4] I choose to use reversed loop and threadID-based indexing with sequential addressing since its one of the most efficient technique and easy to understand.

After this reduction operation, other blocks still need to perform reduction which each other and since they are not cooperative made reduction with atomic add to avoid race conditions.

3. Benchmark Results

GPU: Tesla P4
Note: I use Gooogle Colab to test my implementation.

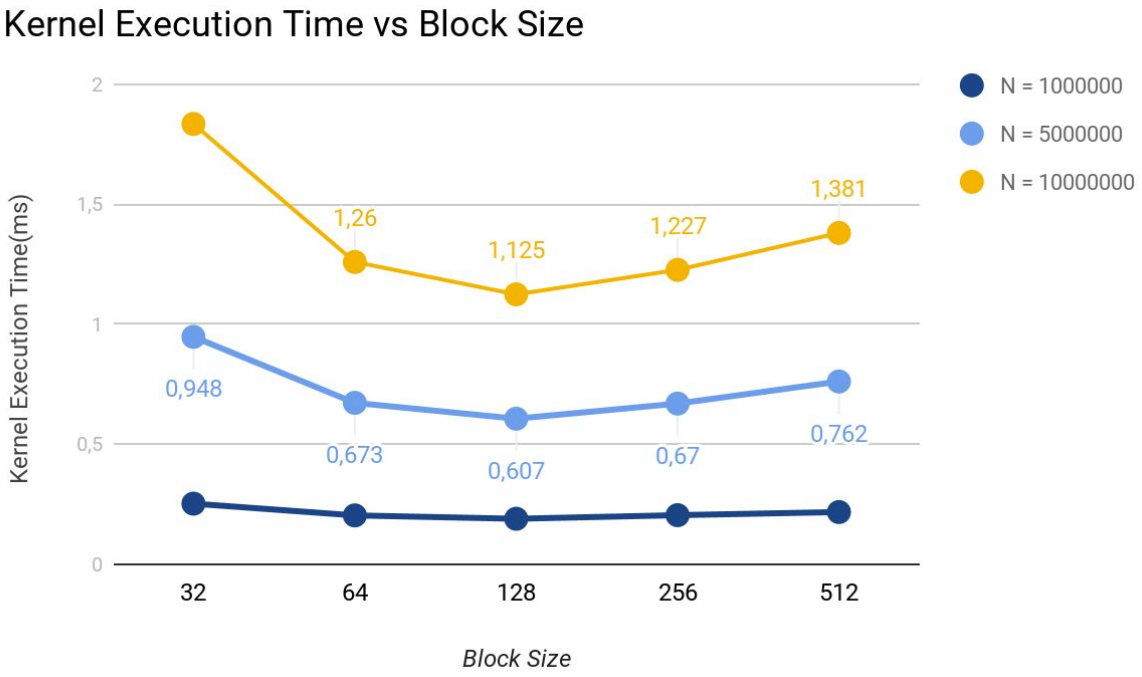

Figure 1: Kernel Execution Time vs Block Size

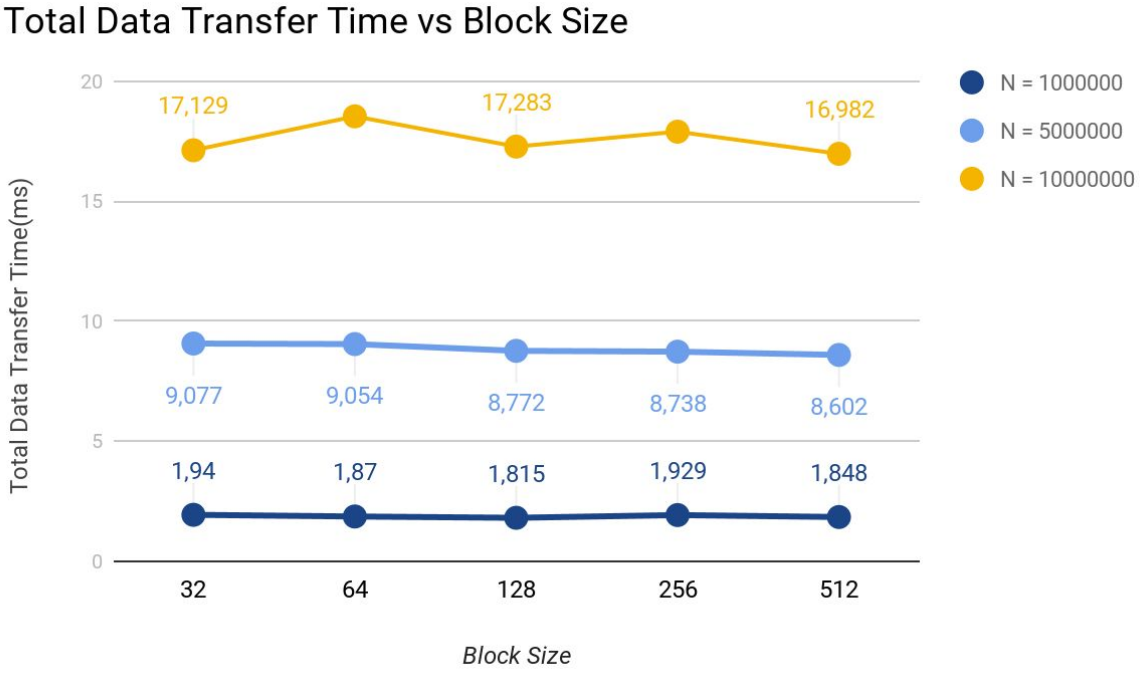

Figure 2: Total Data Transfer Time vs Block Size

4. Discussion of Experiments

When we investigate the test results we can see that total data transfer times are really similar at each execution of the same N which is number of elements in a vector. The reason of this is pretty straight forward since we always transfer same size of data between host and device. We can see the relationship between N = 10 000 000, N = 5 000 000 and N = 1 000 000 that their data size and execution time is consistent respectively.

When we investigate the kernel execution times, they are again really similar to each execution and because of these 2 results, we can again say that the total GPU calculation time is really similar to each execution. I did not plot the CPU execution time because it was really straight forward.

References

[1] Cui, Zheng et al. "An Accurate GPU Performance Model For Effective Control Flow Divergence Optimization". *2012 IEEE 26Th International Parallel And Distributed Processing Symposium*, 2012. *IEEE*, doi:10.1109/ipdps.2012.18. Accessed 29 May 2020.

[2]Schaub, Thomas et al. "The Impact Of The SIMD Width On Control-Flow And Memory Divergence". *ACM Transactions On Architecture And Code Optimization*, vol 11, no. 4, 2015, pp. 1-25. *Association For Computing Machinery (ACM)*, doi:10.1145/2687355. Accessed 29 May 2020.

[3] Wilt, Nicholas. *The CUDA Handbook*. Addison-Wesley, 2013, p. 236.

[4] *Developer.Download.Nvidia.Com*, 2020, https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. Accessed 29 May 2020.