CS 426 Project 4 Report

Eren Aytüre - 21200559 - Section 1

1)Questions

A) What is control flow divergence?

Blocks of threads are forced to follow different execution paths which causes existing hardware mechanisms to reconverge threads duplicate execution of code for unstructured control flow graphs . Non divergent efficiency can be achieved by branch efficiency in CUDA.

B) How can we create a dynamic sized shared memory?

If the size of the shared memory is known at the run time then size of the memory is decided at the run time.

```
extern __shared__  DataType memory[];
```

Size input taken from kernel<<GRID_SIZE, BLOCK_SIZE, BLOCK_SIZE *sizeof(data_type)>>();

C) How can we use shared memory to accelerate our code?

Since each block has their own shared memory, threads in a block can access shared memory without accessing global memory. Shared memory needs a load from global memory. To run a kernel without conflicts, threads need to be synchronized by “__syncthreads()” instruction. This instruction is a barrier. After this instruction, the kernel is executed according to pc.

D) Which CUDA operations give us device properties?

```
for (int i = 0; i < nDevices; i++) {
   cudaDeviceProp prop;
   cudaGetDeviceProperties(&prop, i);
   printf("Device Number: %d\n", i);
   printf("  Device name: %s\n", prop.name);
   printf("  Memory Clock Rate (KHz): %d\n",
          prop.memoryClockRate);
   printf("  Memory Bus Width (bits): %d\n",
          prop.memoryBusWidth);
   printf("  Peak Memory Bandwidth (GB/s): %f\n\n",

2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
 }
```

In my machine output is:

Device Number: 0
  Device name: GeForce GTX 960M
  Memory Clock Rate (KHz): 2505000
  Memory Bus Width (bits): 128
  Peak Memory Bandwidth (GB/s): 80.160000

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

Atomic operations lead to access to a memory location from exactly one thread at a time without race condition.

On GPU architectures with compute capability lower than 6.x, atomics operations done from the GPU are atomic only with respect to that GPU. If the GPU attempts an atomic operation to a peer GPU's memory, the operation appears as a regular read followed by a write to the peer GPU, and the two operations are not done as one single atomic operation.
Similarly, atomic operations from the GPU to CPU memory will not be atomic with respect to CPU initiated atomic operations[1].

Note: System wide atomics are not supported on Tegra devices with compute capability less than 7.2. The new scoped versions of atomics are available for all atomics listed below only for compute capabilities 6.x and later[2].

1) Arithmetic Functions

- int atomicAdd(int* address, int val);
  - ❖ The 32-bit floating-point version of atomicAdd() is only supported by devices of compute capability 2.x and higher.
  - ❖ The 64-bit floating-point version of atomicAdd() is only supported by devices of compute capability 6.x and higher.
  - ❖ The 16-bit __half floating-point version of atomicAdd() is only supported by devices of compute capability 7.x and higher.
- int atomicSub(int* address, int val);
- int atomicExch(int* address, int val);
- int atomicMin(int* address, int val);
  - ○ The 64-bit version of atomicMin() is only supported by devices of compute capability 3.5 and higher.
- int atomicMax(int* address, int val);
  - ○ The 64-bit version of atomicMin() is only supported by devices of compute capability 3.5 and higher.
- unsigned int atomicInc(unsigned int* address, unsigned int val);
- unsigned int atomicDec(unsigned int* address, unsigned int val);
- int atomicCAS(int* address, int compare, int val);

2) Bitwise Functions

The 64-bit version of atomicXor() is only supported by devices of compute capability 3.5 and higher.

- int atomicAnd(int* address, int val);
- int atomicOr(int* address, int val);
- int atomicXor(int* address, int val);

2) Implementation details

First I created x and y arrays that each represents a vector. Their size equals to n. If there is no input file numbers are generated between 0 and 100 randomly, else read from .txt file.

As math $cos\theta = \frac{U*V}{|U||V|}$

U is X array
V is Y array

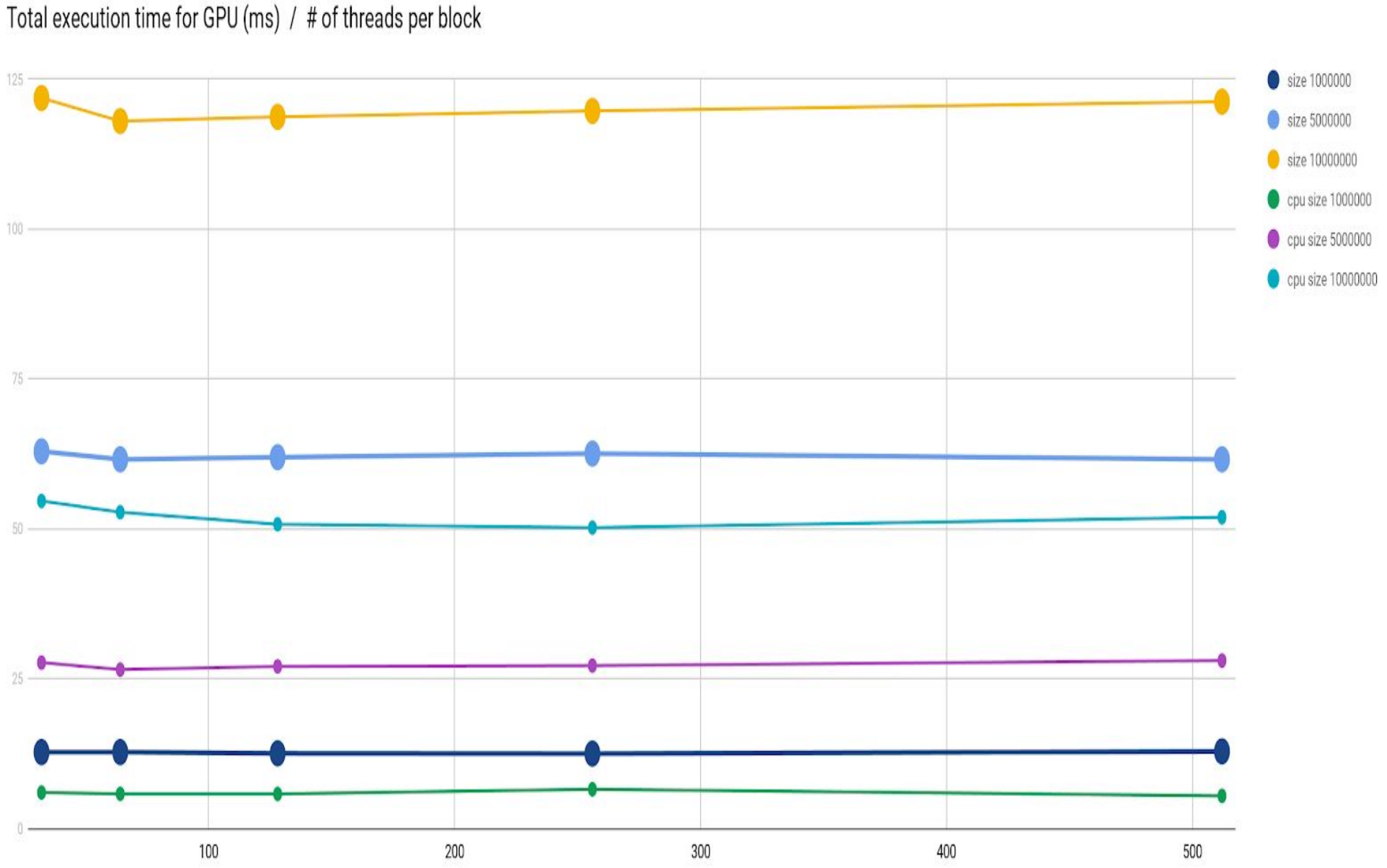
U*V = X[0]*Y[0]+X[1]*Y[1]+X[2]*Y[2]+...X[N-1]*Y[N-1]

|U| = $\sqrt{x[0]^2 + x[1]^2 + x[2]^2 + x[3]^2 + x[4]^2 + ... + x[N-1]^2}$

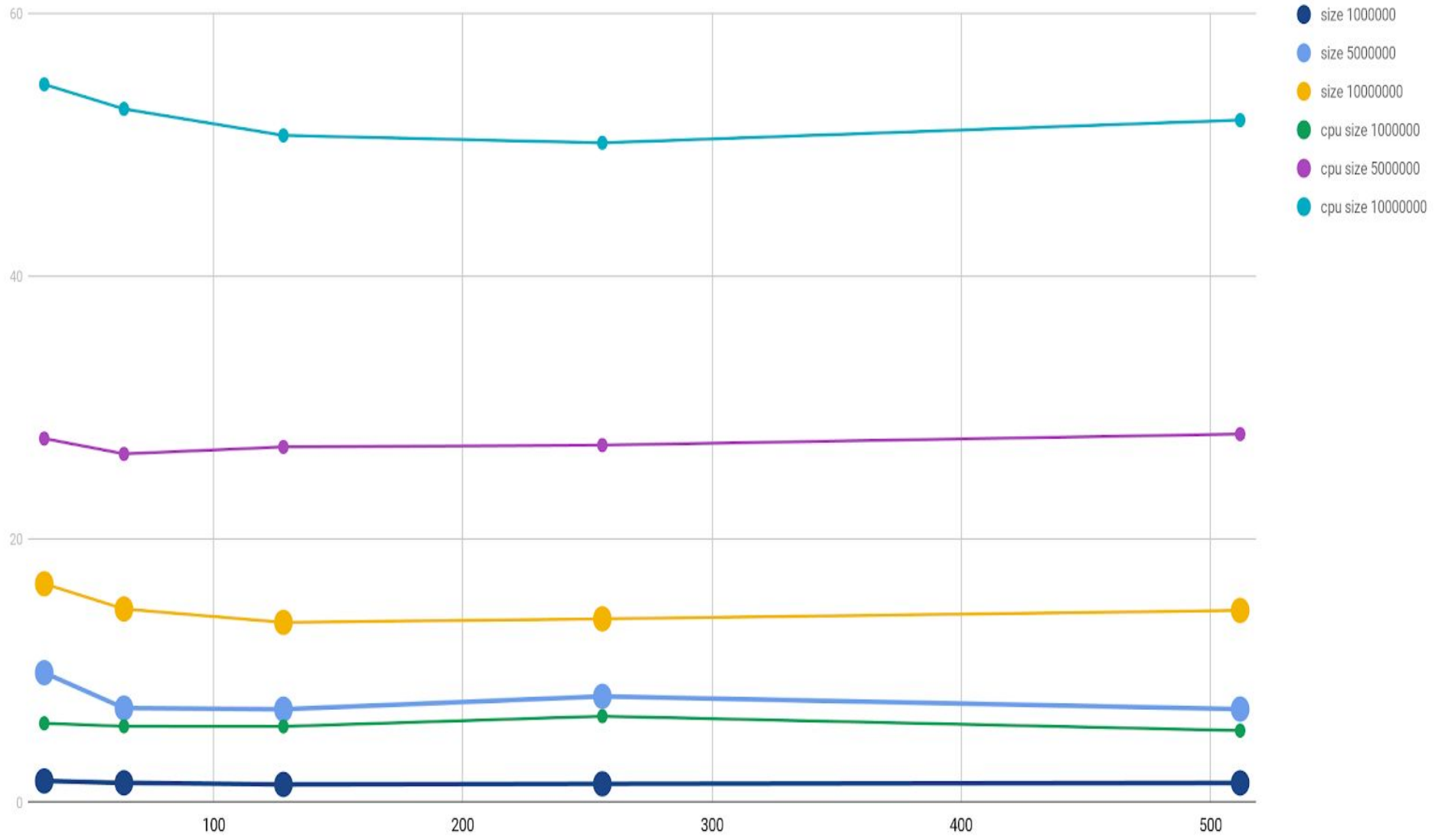
|V| = $\sqrt{y[0]^2 + y[1]^2 + y[2]^2 + y[3]^2 + y[4]^2 + ... + y[N-1]^2}$

To do this operation I used mult kernel that i implemented. This kernel basically computes Z[i] = x[i] * Y[i]. To gain the best performance of it, I used grid stride loop. This kernel computes $x[i]^2, y[i]^2, x[i] * y[i]$. Their each result is stored in gpu global memory with three different arrays to calculate U*V, |U|, |V| after. to calculate U*V, |U|, |V| . Those three arrays are reduced by sum operation. Reduction follows the strategy of sequential addressing, and stride loop is changed with reverse loop. Results are stored and sent to the host. Now, we have, $U * V$; $|U|^2; |V|^2$ separately. Left part is math.

3) Plot for GPU execution times


Total execution time for GPU (ms) / # of threads per block
125
100
75
50
25
0
100
200
300
400
500
size 1000000
size 5000000
size 10000000
cpu size 1000000
cpu size 5000000
cpu size 10000000

kernel execution time for GPU (ms) / # of threads per block and serial execution



4) Discussion of Results

Unfortunately while finding Gpu results, there is a small error due to parallel regions bound in reduction. However, results are similar with cpu results.

Before examining plots, ı would like to clarify that the number of blocks is equal to the size of an array divided by the number of threads per block.

In the first plot we see that total execution of gpu is much slower than cpu execution and in the second plot kernel is much faster than cpu execution. This is because sending and receiving data between host and device is costly. Therefore, Gpu execution must be chosen for a small number of sizer or complex algorithms in terms of time complexity. On the other hand, the number of threads per block does not affect execution too much because the total number of threads is determined by the size of the array(# of blocks).

5) References

[1]CUDA C++ Programming Guide. (n.d.). Retrieved May 25, 2020, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2]CUDA C++ Programming Guide. (n.d.). Retrieved May 25, 2020, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html