Bilkent University

Computer Engineering

CS 425 – Parallel Computing

**Project 4 Report**

**Due Date: 26 December 2017**

**Name: Gülsüm Güdükbay (21401148)**

**Answers To Questions**

1. What is control flow divergence?

In CUDA, threads work in warps of 32 and all of the threads in a warp execute the same task (instruction) concurrently. If threads in the same warp have to do different tasks, it is called control flow (warp) divergence. In NVIDIA graphical processing units, there are instructions that are executed if a conditional flag is true. All threads in CUDA execute all conditional branches so the cost doubles, leading to a loss of performance. Because of this, the NVIDIA compiler checks if all threads in a warp take the same conditional branch, and does warp voting. Every warp chooses the most efficient branch and in some cases, at compilation, all of the threads in a warp may go to the same branch. Different threads executing different warps and all threads going to the same branch causes a huge performance decrease.

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

Dynamically allocated shared memory is used when the amount of shared memory I not known a priori. To allocate dynamic sized shared memory, an optional third configuration parameter is specified when calling a kernel function that uses dynamically sized shared memory:

someFunction<<<1, n, n\*sizeof(float)>>>(arr, n);

Dynamic sized shared memory is allocated as in the below example:

extern \_\_shared\_\_ int s[];

Here, extern specifies that the array is shared and the size of the array comes from the third configuration parameter, which is a size input that is calculated during compile time.

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

Using shared memory is much faster than the global memory that is not cached. This is due to the fact that shared memory is allocated per thread block, so all of the threads in one block utilize the same memory, causing increased locality and controlled data caches, high performance cooperative parallel algorithms such as parallel reductions.

1. 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.

**Code:**

**int** **main**()

{

**int** device\_no;

//get device number

cudaGetDeviceCount(&device\_no);

//for each device find the props

**int** i, driverVersion, runtimeVersion;

**for**(i = 0; i < device\_no; i++)

{

cudaDeviceProp properties;

cudaGetDeviceProperties(&properties, i);

printf("Name of device %d: %s\n", i, properties.name);

cudaDriverGetVersion(&driverVersion);

cudaRuntimeGetVersion(&runtimeVersion);

printf("\tCUDA driver version: %d.%d\n", driverVersion/1000, (driverVersion%100)/10);

printf("\tCUDA runtime Version: %d.%d\n", runtimeVersion/1000, (runtimeVersion%100)/10);

printf("\tCUDA capability version number: %d.%d\n", properties.major, properties.minor);

printf("\tMemory clock rate (KHz): %.0f Mhz\n", properties.memoryClockRate \* 1e-3f);

printf("\tMemory bus width (bits): %d\n", properties.memoryBusWidth);

printf("\tPeak memory bandwidth: (GB/s): %f\n", 2.0\*properties.memoryClockRate\*(properties.memoryBusWidth/8)/1.0e6);

printf("\tTotal constant memory (bytes): %lu\n", properties.totalGlobalMem);

printf("\tTotal global memory: %.0f MBytes (%llu bytes)\n", (**float**)properties.totalGlobalMem/1048576.0f, (**unsigned** **long** **long**) properties.totalGlobalMem);

printf("\tMaximum shared memory available on a thread block (bytes): %lu\n", properties.sharedMemPerBlock);

printf("\tMaximum number of 32-bit registers on a thread block: %d\n", properties.regsPerBlock);

printf("\tWarp size: %d\n", properties.warpSize);

printf("\tMaximum number of threads per block: %d\n", properties.maxThreadsPerBlock);

printf("\tMaximum size of each dimension of a block: %d, %d, %d\n", properties.maxThreadsDim[0], properties.maxThreadsDim[1], properties.maxThreadsDim[2]);

printf("\tMaximum size of each dimension of a grid: %d, %d, %d\n", properties.maxGridSize[0], properties.maxGridSize[1], properties.maxGridSize[2]);

printf("\tClock Rate (KHz): %d\n\n", properties.clockRate);

} }

**Code Output:**

[gulsum@HPZ820 src]$ /usr/local/cuda-9.1/bin/nvcc deviceProps.cu -o deviceProps

[gulsum@HPZ820 src]$ ./deviceProps

Name of device 0: Tesla K20c

CUDA driver version: 9.1

CUDA runtime Version: 9.1

CUDA capability version number: 3.5

Memory clock rate (KHz): 2600 Mhz

Memory bus width (bits): 320

Peak memory bandwidth: (GB/s): 208.000000

Total constant memory (bytes): 4972937216

Total global memory: 4743 MBytes (4972937216 bytes)

Maximum shared memory available on a thread block (bytes): 49152

Maximum number of 32-bit registers on a thread block: 65536

Warp size: 32

Maximum number of threads per block: 1024

Maximum size of each dimension of a block: 1024, 1024, 64

Maximum size of each dimension of a grid: 2147483647, 65535, 65535

Clock Rate (KHz): 705500

Name of device 1: GeForce GTX 480

CUDA driver version: 9.1

CUDA runtime Version: 9.1

CUDA capability version number: 2.0

Memory clock rate (KHz): 1848 Mhz

Memory bus width (bits): 384

Peak memory bandwidth: (GB/s): 177.408000

Total constant memory (bytes): 1545469952

Total global memory: 1474 MBytes (1545469952 bytes)

Maximum shared memory available on a thread block (bytes): 49152

Maximum number of 32-bit registers on a thread block: 32768

Warp size: 32

Maximum number of threads per block: 1024

Maximum size of each dimension of a block: 1024, 1024, 64

Maximum size of each dimension of a grid: 65535, 65535, 65535

Clock Rate (KHz): 1401000

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

There are several functions that allow atomic operations such as “atomicAdd()”, “atomicSub()”, “atomicMax()” and so on. These methods ensure that during the operation, there is no interference from other threads (so no race conditions happen). However, they do not ensure memory fencing and synchronization. The atomic functions can only be used in device functions. There are memory fence functions provided by CUDA that allow enforcement for the ordering of memory accesses. These functions are independent of the type of the memory (shared, global, peer device). There are also synchronization functions that ensure waiting until all the threads in a thread block or all of the threads have reached the point and makes memory accesses visible to all threads.

**Implementation Details**

**CPU Code For Finding Angle Between Two Vectors**

**double** **findAngleCPU**(**const** **double** \*A, **const** **double** \*B, **int** numElements)

{

**double** res = 0.0;

**double** dot\_prod = 0.0;

**double** mag1 = 0.0;

**double** mag2 = 0.0;

**for**(**int** i = 0; i < numElements; i++)

{

dot\_prod += A[i] \* B[i];

mag1 += pow(A[i], 2);

mag2 += pow(B[i], 2);

}

mag1 = sqrt(mag1);

mag2 = sqrt(mag2);

res = acos(dot\_prod/(mag1\*mag2));

**return** res;

}

The main idea is to find the dot product of two vectors by element-wise multiplication and then summation of those products. Also, finding the magnitudes of two vectors by summing up the squares of elements of each array and then taking the square root of each summation. The formula for the angle is therefore:

**GPU Code For Finding Angle Between Two Vectors**

// double precision atomic add function

// taken from https://devtalk.nvidia.com/default/topic/763119/atomic-add-operation/

**\_\_device\_\_** **double** **atomicAdd2**(**double**\* address, **double** val)

{

**unsigned** **long** **long** **int**\* address\_as\_ull = (**unsigned** **long** **long** **int**\*)address;

**unsigned** **long** **long** **int** old = \*address\_as\_ull, assumed;

**do**{ assumed = old;

old = atomicCAS(address\_as\_ull, assumed,\_\_double\_as\_longlong(val +\_\_longlong\_as\_double(assumed)));

} **while** (assumed != old);

**return** \_\_longlong\_as\_double(old);

}

I used the above function to add the sums of each blocks’ threads while reducing the dot products and magnitudes. The reasons of using a double precision atomic add function is to get rid of race conditions and the fact that all of the arrays are double precision.

**\_\_device\_\_** **void** **dot\_product**(**const** **double** \*A, **const** **double** \*B, **int** numElements, **int** blockSize, **int** width\_thread, **double** \*result)

{

**int** start = width\_thread\*(blockIdx.x \* blockSize + threadIdx.x);

**double** sum = 0.0;

**for**(**int** i = start; i < start+width\_thread; i++)

{

**if**(i < numElements)

sum += A[i] \* B[i];

}

atomicAdd2(&result[blockIdx.x], sum);

}

The above function is finding the dot product of two vectors. It does element-wise multiplication for each element of a thread, then sums up the products and outputs it to its own block atomically.

**\_\_device\_\_** **void** **mag\_squared**(**const** **double** \*A, **int** numElements, **int** blockSize, **int** width\_thread, **double** \*result)

{

**int** start =width\_thread\*(blockIdx.x \* blockSize + threadIdx.x);

**double** sum = 0.0;

//sum all elements squared in the block

**for**(**int** i = start; i < start+width\_thread; i++)

{

**if**(i < numElements)

sum+= pow(A[i],2);

}

atomicAdd2(&result[blockIdx.x], sum);

}

The above function is finding the magnitude-squared of a vector. It does element-wise power operation for each element of a thread, then sums up the squares and outputs it to its own block atomically.

**\_\_global\_\_** **void** **find\_angle**(**const** **double** \*A, **const** **double** \*B, **int** numElements, **int** blockSize, **int** width\_thread, **double** \*mag1, **double** \*mag2, **double** \*dot\_prod)

{

**mag\_squared**(A, numElements, blockSize, width\_thread+1, mag1);

**mag\_squared**(B, numElements, blockSize, width\_thread+1, mag2);

**dot\_product**(A, B, numElements, blockSize, width\_thread+1, dot\_prod);

\_\_syncthreads();

}

The above function calls the mag\_squared function for two arrays and the dot\_product function for finding the two arrays’ dot products.

**int** **main**(**int** argc, **char** \*argv[])

{

…

threadElts = 256;

**int** no\_of\_blocks = (**int**)ceil( N / blockSize / threadElts)+1;

**find\_angle**<<<no\_of\_blocks, blockSize>>>(d\_A, d\_B, N, blockSize, threadElts, mag1, mag2, dot\_prod);

…

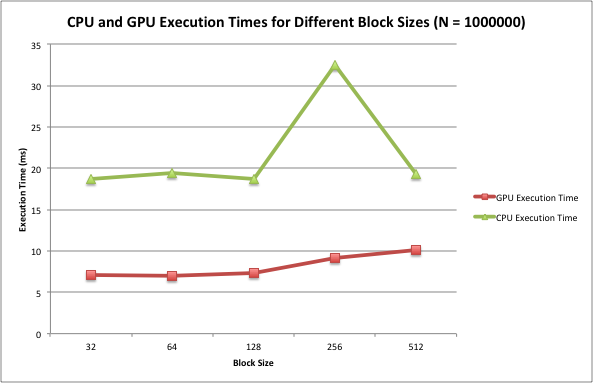
}

The above main code segment shows that 256 elements per thread was chosen to be processed. The number of blocks is therefore number of elements divided by block size and number of thread elements. No shared memory was used in the GPU code, because as also stated in the NVIDIA developer website, using shared memory with atomic operations slows down the execution significantly (source: https://devtalk.nvidia.com/default/topic/514085/atomicadd-in-shared-memory-is-measured-slower-than-in-global-memory-timing-shared-memory-atomic-o/).

**Plots for GPU Execution Times**

**N = 1000000**

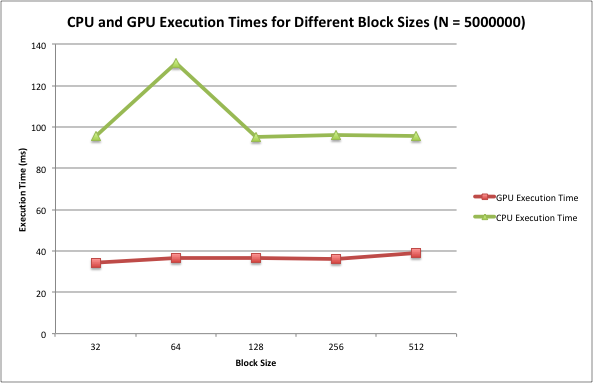
|  |  |  |
| --- | --- | --- |
| Block Size | GPU Execution Time | CPU Execution Time |
| 32 | 7.148128 | 18.701759 |
| 64 | 6.996064 | 19.452415 |
| 128 | 7.356416 | 18.717089 |
| 256 | 9.106592 | 32.450462 |
| 512 | 10.138144 | 19.299744 |



The CPU execution times were always greater than the GPU execution times (approximately 2 times) but the benefit was not as good as in 5000000 elements. The execution times of GPU runs increased after 128 threads per block, which indicates the overhead of creating more blocks than needed. However, the benefit is still a lot.

**N = 5000000**

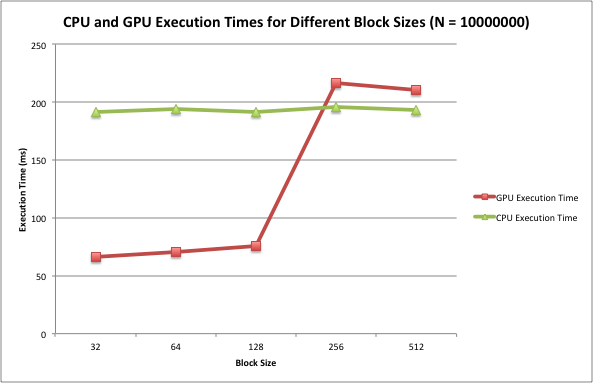
|  |  |  |
| --- | --- | --- |
| Block Size | GPU Execution Time | CPU Execution Time |
| 32 | 34.349442 | 95.813057 |
| 64 | 36.412193 | 131.063873 |
| 128 | 36.689278 | 95.144638 |
| 256 | 36.077438 | 96.117699 |
| 512 | 39.071106 | 95.576576 |



The GPU execution times increased after 256 block size and this is due to the fact that there’s a huge overhead of creating more blocks than needed and granularity became very fine. The CPU execution time was much higher (almost 3 times), so using CUDA and parallelizing the code worked as desired.

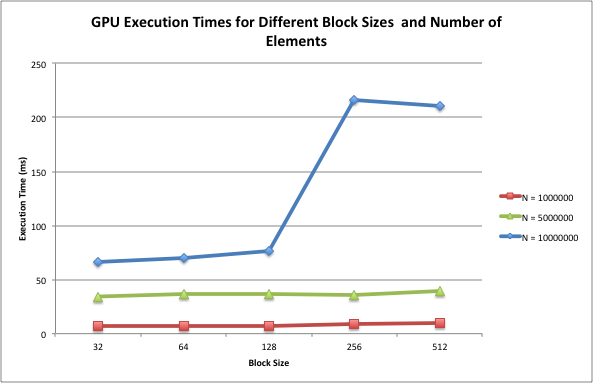
**N = 10000000**

|  |  |  |
| --- | --- | --- |
| Block Size | GPU Execution Time | CPU Execution Time |
| 32 | 66.304321 | 191.047577 |
| 64 | 70.174561 | 193.628036 |
| 128 | 76.032547 | 191.765503 |
| 256 | 216.12381 | 195.686844 |
| 512 | 210.443909 | 193.548416 |



As seen in the above plot, CPU execution times were significantly greater than the GPU execution times (more than 3 times). However, after 128 threads per block, the GPU execution times became more than the CPU execution times due to the overhead of having more blocks than needed.

**GPU Execution Times for Different Block Sizes and Number of Elements**



As seen in the above plot, as the number of elements increase, the execution time increases significantly after 10000000 elements. Also increasing the block size affects the execution time negatively. After 128 threads per block, the overhead became intolerable.