ELEC 374: GPU Machine Problem #2

Matrix Multiplication

Due Date: 2022-03-27

Isabel Frolick – 20155540

To analyze the impact of data transfer time on performance, a dense matrix multiplication routine was created. This program was useable with different number of blocks and threads per block and contained two main functions: the host function and the kernel function.

The routine sent three matrices to the GPU- two input matrices, N and M, and one output matrix, P. The input matrices, N and M, were initialized with random integers, using the CPU. The host function, 'multiply_matrices' was responsible for the allocation, data transfer, kernel launch, and freeing of memory when sending the input matrices to the GPU, very similar to the previous Machine Problem. The largest difference in the host function is the utilization of the CPU for matrix multiplication, as seen below.

```
clock_t start_CPU_time = clock();
for (int i = 0; i < nums; i++) {
    for (int j = 0; j < nums; j++) {
        for (int k = 0; k < nums; k++) {
            CPU_output_matrix[i][j] += N[i][k] * M[k][j];
        }
        if (CPU_output_matrix[i][j] != P[i][j]) {
            cout << "Test failed.";
            cout << CPU_output_matrix[i][j] << ", ";
            cout << i << ", " << j << endl;
            fail = true;
            break;
        }

    }
}
if (!fail) cout << "Test PASSED" << endl;

float finish = (float)(clock() - start_CPU_time) / CLOCKS_PER_SEC;

cout << "The CPU took " << finish << " seconds";
```

The kernel function, 'kernel_mutliplication', the values of the rows and columns was found using the x and y dimensions of the blockIDx, blockDim, and threadIdx CUDA functions. If the matrix size has not yet been exceeded, then the kernel function keeps a running product of the multiplication of the input matrices N and M and placed in the output matrix, P.

```
clock_t start_CPU_time = clock();
for (int i = 0; i < nums; i++) {
    for (int j = 0; j < nums; j++) {
        for (int k = 0; k < nums; k++) {
            CPU_output_matrix[i][j] += N[i][k] * M[k][j];
        }
        if (CPU_output_matrix[i][j] != P[i][j]) {
            cout << "Test failed.";
            cout << CPU_output_matrix[i][j] << ", ";
            cout << i << ", " << j << endl;
            fail = true;
            break;
        }

    }
}
if (!fail) cout << "Test PASSED" << endl;

float finish = (float)(clock() - start_CPU_time) / CLOCKS_PER_SEC;

cout << "The CPU took " << finish << " seconds";
```

Within the host function of the routine, the number of required blocks based on the number of elements in the input matrices is found, as seen below.

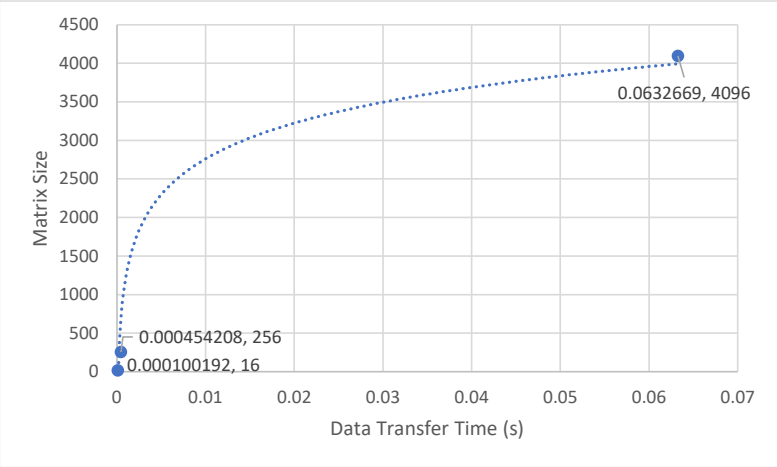
```
int blocksPerThread = SIZE / BLOCK_WIDTH;
if (SIZE % BLOCK_WIDTH) blocksPerThread++;

dim3 threads(blocksPerThread, blocksPerThread);
dim3 blocks(BLOCK_WIDTH, BLOCK_WIDTH);
```

The time to transfer two input matrices at different matrix sizes from the host to the device is below. Note that the block size was 16 for all computations.

Matrix Size	Transfer Time (s)
16x16	0.000100192
256x256	0.000454208
4096x4096	0.0632669

This table has been plotted on the graph below.

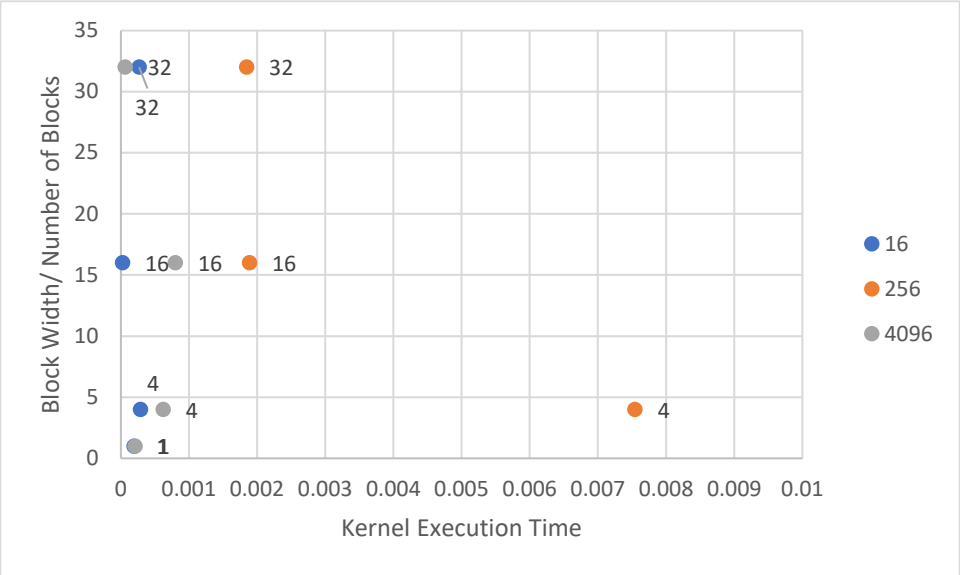


To compare the performance of the GPU computation vs the CPU computation for matrix multiplication, matrix sizes 16x16, 256x256, 4096x4096 were selected with block widths: 1, 4, 16, and 32.

Matrix Size	Thread Block Width	GPU Performance (s)	CPU Performance (s)
16x16	1	0.000191872	0.00000001
	4	0.000287232	0.001
	16	0.000024576	0.004
	32	0.000270464	0.001
256x256	1	0.0893525	0.163
	4	0.00754339	0.16
	16	0.00188659	0.141
	32	0.0018457	0.162
4096x4096	1	0.000209408	0.002
	4	479.448	420.22
	16	33.3839	21.33
	32	4.7532	2.34

The performance of matrix multiplication is limited by the memory throughput, which is why GPU computation is typically faster than CPU computation, as it has a higher memory throughput. However, when there are very few operations sent to the GPU, or the data is segmented into small blocks, such as when the block width is 1, offloading to the device is a slowdown due to the high start up time and large overhead associated with the device. When the block width is very small, or the number of operations is minimal, it is better to perform the computation on the host (CPU) system.

By increasing the block width, or the number of threads within one block allowing them to communicate with shared memory, run in parallel, and execute atomic operations, is an improvement to the routine's efficiency. By increasing the block width, more threads can execute operations simultaneously, thus completing operations faster. In this routine, the maximum block width is 64, as there are a maximum 1024-bit dimension size for blocks on this system, as per Machine Problem 1.



When threads exist in the same block, they have shared memory for easy data transfer. All threads in a block load a portion of the global memory into their collective shared memory, such that each thread in the block loads one element from each input matrix into the shared memory. The threads 'remember' the running sum of the previous computations, and the next segment of data can be loaded from the global memory, and the running sum is maintained. Resultantly, the shared memory can allow each element in the input matrix to be loaded once, increasing speed without decreasing storage.

The number of memory accesses necessary for an integer computation is proportional to the width of the block and the number of operations completed on that element. Each thread computes one element in the output matrix but must access each element in a row as many times as is equal to the width of the block. For example, a thread with threadID(x,y) computes the output matrix elements of row x and column y, and therefore must access each element in row x to compute column y, column y+1, etc. Therefore, the ratio of integer computations to memory accesses in each thread is 1:WIDTH where WIDTH is the block width. For each computation needed, it will need to be accessed block width times.

However, as the kernel accesses each memory location WIDTH times, the kernel is a faster memory system that reduces the time for the data to load. Resultantly, the number of accesses is not decreased when computing on the device compared to the host, but the access speed is faster.