ELEC 374: GPU Machine Problem #2

Device Query

Due Date: 2022-03-27

Isabel Frolick – 20155540

To become familiarized with CUDA API, a matrix addition routine where each thread produces one or more output matrix elements by taking in two input matrices, A and B and produces an output matrix, C. The problem has been simplified under the assumption that only square matrices will be entered.

In this problem, a parameter 'SIZE' was used as a global variable to define the size of thread blocks. Two functions were written to create a matrix adding routine: a host function 'matrix_addition' and a global kernel function, 'kernel_addition'.

The host function, 'matrix_addition', created four 'auto' type variable, two input matrices (matrix1, matrix2) and two output matrices: one to be passed to the kernel and GPU (matrix_out) and one to use as a comparison on the CPU (CPU_matrix_out).

```
cudaError_t matrix_addition(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE], int num) {

    auto matrix1 = new int[SIZE][SIZE];
    auto matrix2 = new int[SIZE][SIZE];
    auto matrix_out = new int[SIZE][SIZE];
    auto CPU_matrix_out = new int[SIZE][SIZE];
    bool fail = false;

    int square_matrix_size = num*num;

    for (int x = 0; x < num; x++) {
        for (int y = 0; y < num; y++) {
            CPU_matrix_out[x][y] = 0;
        }
    }

    //each thread producing one output matrix element
    //dim3 threads(SIZE, SIZE, 1);
    //dim3 blocks(num / SIZE, num / SIZE, 1);

    //each thread producing one output matrix row
    dim3 threads = dim3(num/SIZE, 1, 1);
    dim3 blocks = dim3(SIZE, 1, 1);

    //each thread producing one output matrix column
    //dim3 threads(1, num/SIZE, 1);
    //dim3 blocks(1, SIZE, 1);
```

The performance of the GPU must be found when each thread produces one output matrix element, one output matrix row, one output matrix column. As such, the dim3 integer vector type was used to declare the number of threads per output, row, or column. Only one of the three functionalities was run

at one time, while the others were commented out.

```
//checking if we can use the first device
cudaError_t cudaStatus;
cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
    goto Error;
}

// Allocate GPU buffers for three matrices (two input, one output)    .
cudaStatus = cudaMalloc((void**)&matrix1, square_matrix_size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&matrix2, square_matrix_size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&matrix_out, square_matrix_size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}
```

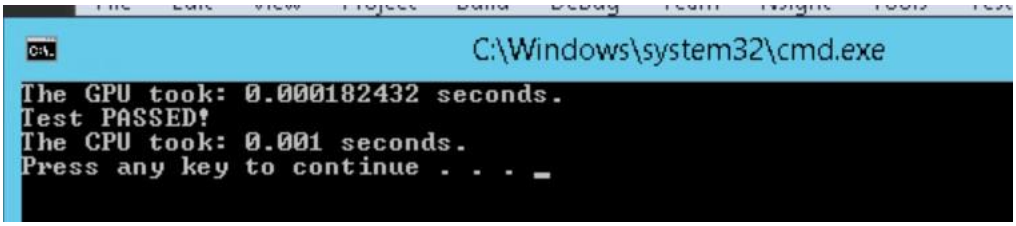
The host function was responsible for allocating memory for the input and output matrices sent to the GPU, launching the kernel by invoking the 'kernel_addition' function, transferring the output data to host, and freeing the devices' memory. The host function also contained timing logic to monitor the start and stop timing of the GPU compared to the CPU time.

```
kernel_addition << <threads, blocks >> >(*matrix1, *matrix2, *matrix_out, num);


cudaStatus = cudaMemcpy(C, matrix_out, square_matrix_size * sizeof(int), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "CudaMemcpy failed for Output Matrix returning from device to host!");
    goto Error;
}

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&timing_GPU, start, stop);

cout << "The GPU took: " << timing_GPU / 1000 << " seconds." << endl;



clock_t cpuStart = clock();
for (int i = 0; i < num; i++) {
    for (int j = 0; j < num; j++) {
        CPU_matrix_out[i][j] = A[i][j] + B[i][j];
        if (CPU_matrix_out[i][j] != C[i][j]) {
            cout << "Test failed.";
            fail = true;
            break;
        }

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

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

cout << "The CPU took: " << finish << " seconds." << endl;

return cudaStatus;
```

The kernel_addition function used dimensions of the blockID, blockDim, and threadID CUDA function iterate through the matrix addition, dependent on the output matrix produced by each thread. Below is

the instantiation of the kernel_additon function when each thread produces one output matrix column when the matrix row and column size is 16 blocks, respectively.

The output to the console when completing the column production test with a 16x16 matrix size is shown below.




```
#define SIZE 16

__global__ void kernel_addition(int *A, int *B, int *C, int num)
{
    //Each thread produces one output matrix element
    /*
    int i = (blockIdx.x *blockDim.x + threadIdx.x) * num + (blockIdx.y *blockDim.y + threadIdx.y);
    C[i] = A[i]+B[i];
    */

    /*
    //Each thread produces one output matrix row
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < num) {
        for (int k = 0; k < num; k++) {
            C[i*num + k] = B[i*num + k] + A[i*num + k];
        }
    }
    */

    //Each thread produces one output matrix column
    int i = blockIdx.y*blockDim.y + threadIdx.y;

    if (i<num){
        for ( int k = 0; k<num; k++){
            C[i*num+k] = A[i*num+k] + B[i*num+k];
    }
    }
```

Thread:Matrix Ratio	Matrix Size	GPU Performance (s)	CPU Performance (s)
Thread:Matrix Element	16 x 16	0.00527037	0.001
	256 x 256	0.00164928	0.002
	4096 x 4096	0.782772	0.698
Thread:Matrix Row	16 x 16	0.00021344	0.00001
	256 x 256	0.002093334	0.002
	4096 x 4096	0.229789	0.227
Thread:Matrix Column	16 x 16	0.000182432	0.001

	256 x 256	0.00197709	0.001
	4096 x 4096	0.156608	0.153

From the table above, it is evident that the GPU Performance is better than the CPU, up to a certain point on this device. The time needed to complete the GPU computation of the smaller matrices (16x16, 256x256) is much faster than the time needed to complete the CPU computation. This is most likely because GPU functionality permits more parallel computations than CPU because they have more blocks and, in each block, there exist multiple threads, resulting in faster computations. However, for the larger size matrix (4096x4096), the CPU performance is better than the GPU. This is most likely because, for matrix addition, transferring large quantities of data to the GPU takes a long time compared to executing a simple operation like addition on the CPU.

When comparing the three methods of thread production for the matrix output, per element- per row, and per column- performance is best when each thread produces one output matrix column, and worse when each thread produces one output matrix element. It is clear why the element-wise thread production would have the worst performance, as there is no grouping of elements to parse down the required data. The column-wise thread production is the fastest due to the manner with which CUDA uses the memory subsystem. When each thread reads by column, adjacent threads are optimally loading adjacent data. Comparatively, when each thread reads by row, adjacent threads are loading data separated by a row width, which is less efficient as there is a larger space to traverse.