Machine Problem 2

Joel Coghlin – 20228087

Full Coda Program – With Updated Kernel

Note that TILE_WIDTH and the matrix width were modified for each experiment

```
//Joel Coghlin - 20228087
//Machine Problem 2
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <cstdlib>   // Include for rand() function
#include <random>

#define TILE_WIDTH 5

/*/ Old Kernel function to perform matrix multiplication on GPU
__global__ void matrixMul(float* A, float* B, float* C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < width && col < width) {
        float sum = 0.0;
        for (int k = 0; k < width; k++) {
            sum += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = sum;
    }
}
*/



__global__ void matrixMul(float* A, float* B, float* C, int width) {
    __shared__ float share_A[TILE_WIDTH * TILE_WIDTH];
    __shared__ float share_B[TILE_WIDTH * TILE_WIDTH];



    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int bx = blockIdx.x;
    int by = blockIdx.y;

    int row = by * blockDim.y + ty; //indices
    int col = bx * blockDim.y +tx;

    float sum = 0.0;

    if (row >= width || col >= width) return;
    for (int m = 0; m < (width-1)/TILE_WIDTH + 1; ++m) {
        int A_index = row * width + m * TILE_WIDTH + tx;   //load the matrices into
shared memory arrays.
        int B_index = (m * TILE_WIDTH + ty) * width + col;

        share_A[ty * TILE_WIDTH + tx] = A[A_index];
        share_B[tx * TILE_WIDTH + ty] = B[B_index];


        //check if the shared indexing of A tile is out of bounds (row matrix)
        if (m * TILE_WIDTH + tx < width) {
            share_A[ty * TILE_WIDTH + tx] = A[A_index];
        }
        else {
            share_A[ty * TILE_WIDTH + tx] = 0.0;
        }

        //check the shared indexing of the B tile as well (column matrix)
        if (m * TILE_WIDTH + ty < width) {
            share_B[ty * TILE_WIDTH + tx] = B[B_index];
        }
        else {
            share_B[ty * TILE_WIDTH + tx] = 0.0;
        }



        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k) {
            if(m*TILE_WIDTH +k < width)
                sum += share_A[ty*TILE_WIDTH + k]*share_B[k*TILE_WIDTH + tx];
//do mat mul with these new arrays
        }
        __syncthreads();
    }


    C[row * width + col] = sum;

}

// Function to perform matrix multiplication on CPU
void matrixMulCPU(float* A, float* B, float* C, int width) {
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < width; j++) {
            float sum = 0.0;
            for (int k = 0; k < width; k++) {
                sum += A[i * width + k] * B[k * width + j];
            }
            C[i * width + j] = sum;
        }
    }
}

int main() {

    int numDevices;
    cudaGetDeviceCount(&numDevices);
    printf("There are/is %d device(s) available\n", numDevices);

    cudaDeviceProp deviceProperties;
    cudaGetDeviceProperties(&deviceProperties, 0);

    for (int i = 0; i < numDevices; i++) {
        printf("Device Number (on system): %d\n", i);
        printf("Device Clock Rate: %d kHz\n", deviceProperties.clockRate);
        printf("Number of SMs: %d\n", deviceProperties.multiProcessorCount);
        printf("Number of Cores: \n");
        printf("Warp Size: %d\n", deviceProperties.warpSize);
        printf("Total Global Memory: %zu Bytes\n", deviceProperties.totalGlobalMem);
        printf("Total Constant Memory: %zu Bytes\n",
deviceProperties.totalConstMem);
        printf("Shared Memory Per Block: %zu Bytes\n",
deviceProperties.sharedMemPerBlock);
        printf("Registers Available Per Block: %d\n",
deviceProperties.regsPerBlock);
        printf("Max Threads Per Block: %d\n", deviceProperties.maxThreadsPerBlock);
        printf("Max Size of dimBlock: %d, %d, %d\n",
deviceProperties.maxThreadsDim[0], deviceProperties.maxThreadsDim[1],
deviceProperties.maxThreadsDim[2]);
        printf("Max Size of dimGrid: %d, %d, %d\n", deviceProperties.maxGridSize[0],
deviceProperties.maxGridSize[1], deviceProperties.maxGridSize[2]);
    }

    cudaSetDevice(0);

    // Matrix size
    int width = 10;
    int size = width * width * sizeof(float);

    // Allocate memory for host matrices
    float* h_A = 0;
    float* h_B = 0;
    float* h_C = 0;
    float* h_C_CPU = 0; // For CPU computation

    cudaMallocHost((void**)&h_A, size);
    cudaMallocHost((void**)&h_B, size);
    cudaMallocHost((void**)&h_C, size);
    cudaMallocHost((void**)&h_C_CPU, size);

    /// Seed the random number generator
    srand(time(nullptr));

    /*// Initialize host matrices with random numbers from 1 to 10
    for (int i = 0; i < width * width; i++) {
        h_A[i] = static_cast<float>(rand() % 10 + 1);  // Random number from 1 to 10
        h_B[i] = static_cast<float>(rand() % 10 + 1);  // Random number from 1 to 10
    }
    */

    for (int i = 0; i < width * width; i++) {
        h_A[i] = (float)rand() / RAND_MAX;
    }

    for (int i = 0; i < width * width; i++) {
        h_B[i] = (float)rand() / RAND_MAX;
    }

    for (int i = 0; i < width * width; i++) {
        h_C[i] = 0;
        h_C_CPU[i] = 0;
    }

    // Allocate memory for device matrices
    float* d_A, * d_B, * d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Copy host matrices to device
    cudaEventRecord(start, 0);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float milliseconds = 0;   //to Hold the time elapsed


    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("copying host to device took: %f\n", milliseconds);

    //Define grid and block dimensions
    dim3 dimGrid = dim3((width + TILE_WIDTH - 1) / TILE_WIDTH, (width + TILE_WIDTH -
1) / TILE_WIDTH, 1);
    dim3 dimBlock = dim3(TILE_WIDTH, TILE_WIDTH, 1);

    //Launch kernel
    cudaEventRecord(start, 0);
    matrixMul << <dimGrid, dimBlock >> > (d_A, d_B, d_C, width);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("GPU Kernel took: %f\n", milliseconds);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    /* Print GPU result (used in debugging)
        printf("\nGPU Result:\n");
        for (int i = 0; i < width; ++i) {
            for (int j = 0; j < width; ++j) {
                printf("%.3f ", h_C[i * width + j]);
            }
            printf("\n");
        }*/
        // Perform matrix multiplication on CPU for comparison
        //cudaEvent_t start, stop;

    cudaEventRecord(start, 0);
    matrixMulCPU(h_A, h_B, h_C_CPU, width);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("The CPU took: %f\n", milliseconds);

    //Verify
    for (int i = 0; i < width * width; i++) {
        if (fabs(h_C[i] - h_C_CPU[i]) > 1) {
            printf("Test FAILED\n");
            //break;
        }
    }

    for (int e = 0; e < width * width; e++) {
        printf(" %.3f\n", h_C[e]);
    }

    for (int w = 0; w < width * width; w++) {
        printf(" %.3f\n", h_C_CPU[w]);
    }


    /* Print CPU result (also used in debugging)
    printf("CPU Result:\n");
    for (int i = 0; i < width; ++i) {
        for (int j = 0; j < width; ++j) {
            printf("%.3f ", h_C_CPU[i * width + j]);
        }
        printf("\n");
    }*/

    cudaEventRecord(start, 0);
    cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_B, d_B, size, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("copying device to host took: %f\n", milliseconds);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);


    //Copy result matrix from device to host
    //cudaEvent_t start, stop;
    //Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    //Free host memory
    cudaFreeHost(h_A);
    cudaFreeHost(h_B);
    cudaFreeHost(h_C);
    cudaFreeHost(h_C_CPU);

    return 0;
}
```

Results

I tested, recorded then plotted the results for my new tiling kernel. I expected the use of shared memory to decrease the duration of computing each matrix multiplication. The results are outlined below in Table 1. The baseline matrix multiplication kernel is identified as *Machine Problem 1* and the new shared memory kernel is identified as *Machine Problem 2*. Results for *Machine Problem 1* kernel were pulled from my results from my submission. *Machine Problem 2* data was retrieved by testing the program and altering the TILE_WIDTH and matrix size. Note that computation time does not include the time required for transferring data to and from the host. However, it **does** include the time taken to place results in shared memory.

Table 1: Tiling Kernel Matrix Multiplication Results

Kernel	**Specifics**	**Duration with Respect to Matrix Size (ms)**				
		100	**250**	**500**	**1000**	**1500**
Machine Problem 1	Standard Block Width of 32	0.0123	0.459	2.836	27.731	86.605
Machine Problem 2	TILE_WIDTH = 2	3.571	36.199	275.45	1514.448	4839.934
	TILE_WIDTH = 5	1.143	5.471	29.063	194.272	581.776
	TILE_WIDTH = 10	0.212	1.395	9.018	74.067	242.475
	TILE_WIDTH = 25	0.183	0.891	7.51	55.014	175.307

This table is be visualized in Figure 1 and Figure 2. In Figure 1, the computation speed for each TILE_WIDTH is compared with the matrix size. Increasing the tile width showed an immense improvement in computation speed. This is because the kernel spends more time computing the result entries rather than transferring data to shared memory and global memory. In the case that the TILE_WIDTH is 2, the kernel is spending an unnecessary amount of time transferring data instead of computing the result.

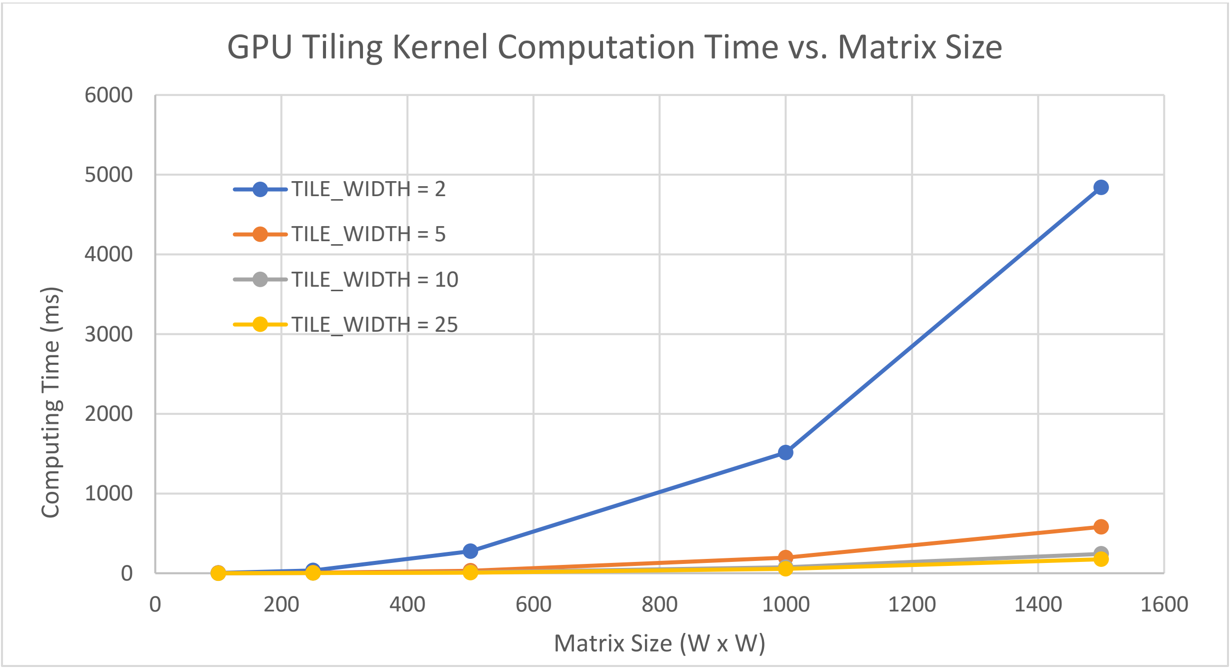

Figure 1: GPU Tiling Kernel Computation Time vs. Matrix Size

Figure 2 shows the baseline kernel's computation time against each matrix size. It appears to follow the same trend as the trendlines in Figure 1, however, it is important to note that the scale of the computation time axis in Figure 2 only displays up to 100ms. This shows that the baseline matrix multiplication is substantially better than the tiling kernel at TILE_WIDTH = 2.

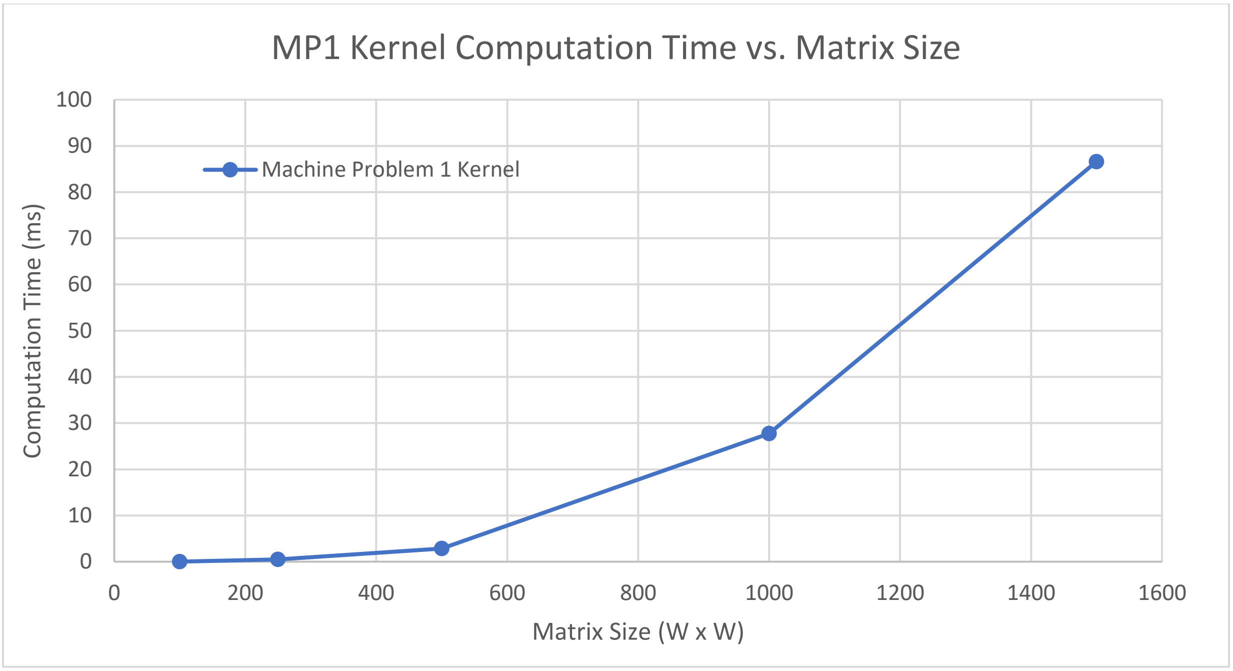

Figure 2: Baseline Matrix Multiplication Kernel from MP1

Comparing With Machine Problem 1 and Discussion Questions

The differences in results for each kernel were – in my opinion – unexpected. I was expecting to achieve greater results with the tiling kernel since it is trying to save time in memory allocation. However, the tiling algorithm does take slightly longer than the baseline kernel from MP1. I thought that this may be caused by the frequent allocation of memory compared to the actual computations. In the baseline kernel, the block width is set and each thread computes an element of the result matrix. This is then written to global memory in one step. In the tiling kernel, while it is true that transferring data from shared memory to global memory is faster, when the TILE_WIDTH is small, it is required to happen more often. Even if the computation themselves are just as fast, the excessive memory transfers make the timestamps appear longer in this case.