Machine Problem 2

Jacob Badali ~ 20290739 ~ March 29th, 2024

“I do hereby verify that this machine problem submission is my own work and contains my own original ideas, concepts, and designs. No portion of this report or code has been copied in whole or in part from another source, with the possible exception of properly referenced material".

Part 1

Code was written to implement call the GPU and implement tiled matrix multiplication. The code can be found in

The block width (number of tiles) was changed, and a sample of the following results were found for each requested matrix size.

Time, 100x100					
GPU Tiles	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	1.0608	1.1448	1.055008	1.163456	1.106016
5	0.229088	0.292608	0.336	0.324448	0.295536
10	0.201536	0.198688	0.227744	0.227616	0.213896
25	0.198208	0.27312	0.18624	0.268	0.231392

Figure 1 100x100 tiled matrix multiplication

You can see full tables for both tiled and threaded matrix multiplication in the Appendix. The results were plotted against the Threaded results seen below in Figure 2.

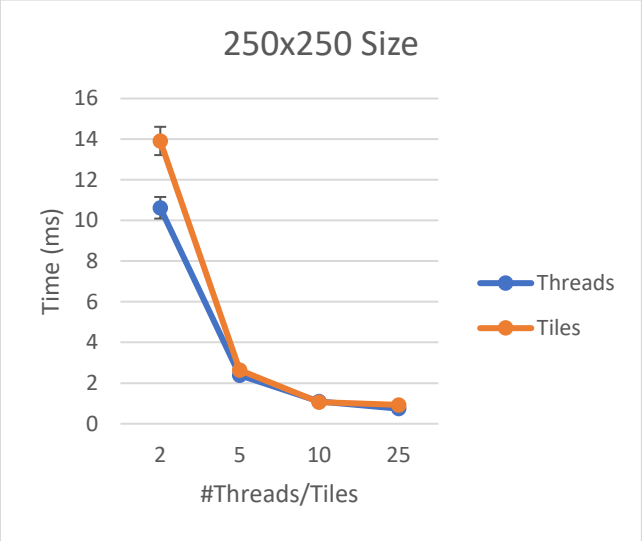
Time, 100x100					
GPU Threa	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	0.673952	0.810688	0.809248	0.672992	0.74172
5	0.269376	0.143168	0.284352	0.268352	0.241312
10	0.261056	0.255904	0.264896	0.133216	0.228768
25	0.259808	0.14176	0.136384	0.143232	0.170296

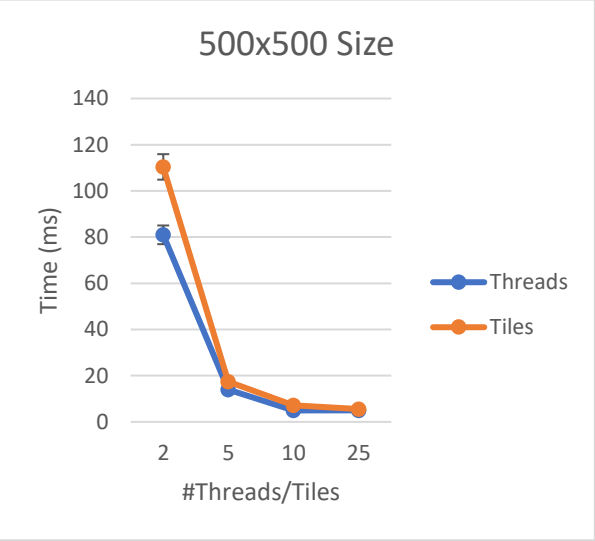
Figure 2 100x100 threaded matrix multiplication

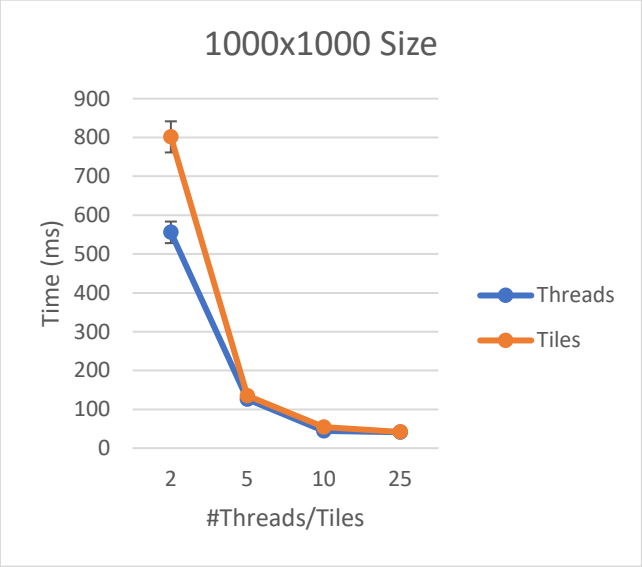
The plot of their averages can be seen in Figure 3.

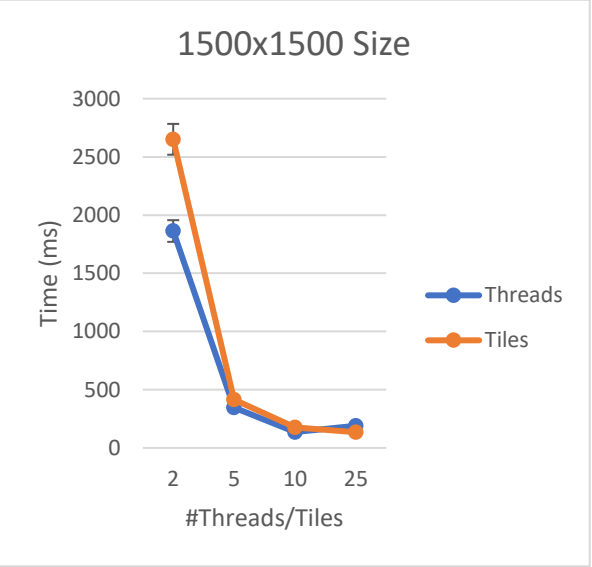
Figure 3 Threads vs Tiles 100x100 size

As can be seen in the graph, the tiled matrix multiplication is almost exclusively slightly longer than the threaded matrix multiplication. This was repeated for all sizes, and the results can be seen below.







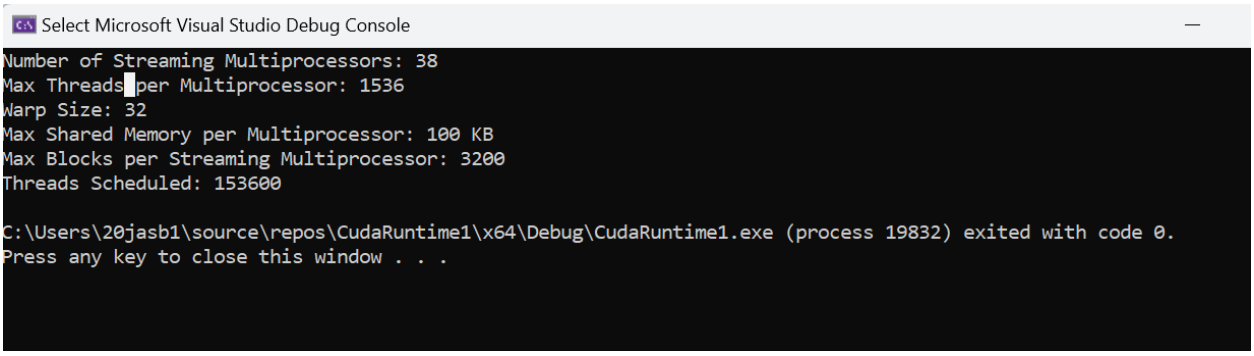


Again, the tiled matrix multiplication takes longer than the threaded multiplication. Tiled matrix multiplication should outperform threaded, because it reduces global memory accesses, and maximizes the utilization of shared memory, which is faster compared to global. There are multiple possible reasons why this may not be the case, however. First, tiled matrix multiplication introduces more overhead because of its memory sharing. Second, if the tile dimensions are note aligned well with the tile size, it can lead to inefficient execution.

Question 1

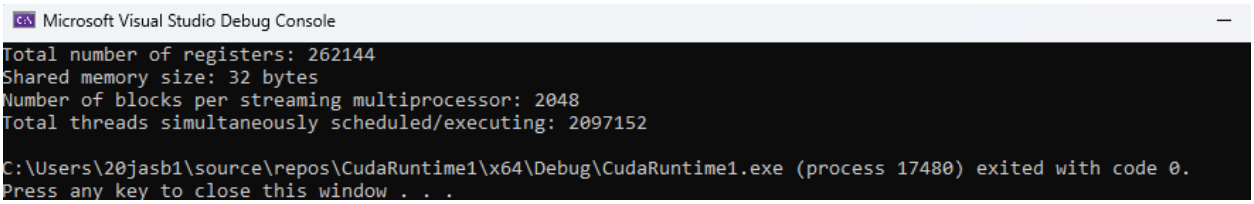
The number of threads that can be simultaneously scheduled on a CUDA device depends on its own hardware limitations. This can be found by accessing its id, and its specifications. The multiProcessorCount, maxThreadsPerMultiProcessor, warpSize, and then the sharedMemPerMultiprocessor were accessed. The shared memory needed to be considered per the

lecture slides, with a chosen tile width of 2. This accounts for the shared memory. A sample output can be seen below.



Question 2

To access the resources of your GPU, you again need to look to the hardware specifications. This time, we are accessing the regsPerBlock, and multiplying it by the tile width, to calculate the total registers. The shared memory is the number of tiles, multiplied two ways by the tile width (2 dimensions) and then multiplied by the size of a float, to encapsulate the size. The total threads are the max number are just the max number of blocks multiplied by the number of threads per block. A sample output can be seen below.



Appendix

Part 1 Code

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>

#include <stdio.h>
#include <math.h>
#include <stdlib.h>


#include <device_functions.h>

#define WIDTH (1500) //CHANGE THIS!!!

#define TILE_WIDTH 2 //CHANGE THIS!!! [2,5,10,25]

//Tiled Multiplication Kernel
__global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
	__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
```

```
	__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

	int bx = blockIdx.x; int by = blockIdx.y;
	int tx = threadIdx.x; int ty = threadIdx.y;

	int Row = by * TILE_WIDTH + ty;
	int Col = bx * TILE_WIDTH + tx;

	float Pvalue = 0;

	for (int ph = 0; ph < Width / TILE_WIDTH; ++ph) {
		Mds[ty][tx] = M[Row * Width + ph * TILE_WIDTH + tx];
		Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * Width + Col];
		__syncthreads();
		for (int k = 0; k < TILE_WIDTH; ++k) {
			Pvalue += Mds[ty][k] * Nds[k][tx];
		}
		__syncthreads();
	}
	P[Row * Width + Col] = Pvalue;
}

void matrixMulCPU(float* M, float* N, float* P, int Width) {
	for (int i = 0; i < Width; ++i) {
		for (int j = 0; j < Width; ++j) {
			float sum = 0;
			for (int k = 0; k < Width; ++k) {
				sum += M[i * Width + k] * N[k * Width + j];
			}
			P[i * Width + j] = sum;
		}
	}
}

int main() {
	float* d_M = 0;
	float* d_N = 0;
	float* d_P = 0;

	float* h_M;
	float* h_N;
	float* h_P;
	float* h_Pcheck;


	int size = WIDTH * WIDTH * sizeof(float);

	cudaMallocHost((void**)&h_M, size);
	cudaMallocHost((void**)&h_N, size);
	cudaMallocHost((void**)&h_P, size);
	cudaMallocHost((void**)&h_Pcheck, size);

	int NumBlocks = WIDTH / TILE_WIDTH;
	if (WIDTH % TILE_WIDTH) NumBlocks++;

	dim3 dimGrid(NumBlocks, NumBlocks);
	dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
```

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

	float elapsedTime_MatrixMulTiled;

	cudaMalloc((void**)&d_M, size);
	cudaMalloc((void**)&d_N, size);
	cudaMalloc((void**)&d_P, size);

	//fill host matrices
	for (int k = 0; k < WIDTH; k++) {
		for (int j = 0; j < WIDTH; j++) {
			h_M[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0f; //
fill with rand values from 0-100
			h_N[k * WIDTH + j] = ((float)rand() / RAND_MAX) * 100.0f;
			h_P[k * WIDTH + j] = 0.0;
			h_Pcheck[k * WIDTH + j] = 0.0;
		}
	}

	cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);
	cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);

	cudaEventRecord(start, 0);

	matrixMulKernel << <dimGrid, dimBlock, 0, 0 >> > (d_M, d_N, d_P, WIDTH);

	cudaEventRecord(stop, 0);
	cudaEventSynchronize(stop);

	cudaEventElapsedTime(&elapsedTime_MatrixMulTiled, start, stop);
	printf("Device Matrix Mul Time, size[%d]: %f ms\n", WIDTH,
elapsedTime_MatrixMulTiled);

	cudaMemcpy(h_M, d_M, size, cudaMemcpyDeviceToHost);
	cudaMemcpy(h_N, d_N, size, cudaMemcpyDeviceToHost);

	cudaMemcpy(h_P, d_P, size, cudaMemcpyDeviceToHost);

	//Do CPU matrix multiplication to refer to
	matrixMulCPU(h_M, h_N, h_Pcheck, WIDTH);

	int check = 0;
	for (int i = 0; i < WIDTH; i++) {
		for (int j = 0; j < WIDTH; j++) {
			if (abs(h_P[i * WIDTH + j] - h_Pcheck[i * WIDTH + j]) > 1) {
				check = 1;
			}
		}
	}


	if (check == 0) {
		printf("TEST PASSED\n");
	}
	else {
		printf("TEST FAILED\n");
```

```
        }

        cudaFree(d_M);
        cudaFree(d_N);
        cudaFree(d_P);
        cudaFree(h_M);
        cudaFree(h_N);
        cudaFree(h_P);
        cudaFree(h_Pcheck);
}
```

Part 1 Tiled and Threaded Tables to Accompany Graphs

MP2 Tiles					
Time, 100x100					
GPU Tiles	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	1.0608	1.1448	1.055008	1.163456	1.106016
5	0.229088	0.292608	0.336	0.324448	0.295536
10	0.201536	0.198688	0.227744	0.227616	0.213896
25	0.198208	0.27312	0.18624	0.268	0.231392

MP1 Threads					
Time, 100x100					
GPU Threa	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	0.673952	0.810688	0.809248	0.672992	0.74172
5	0.269376	0.143168	0.284352	0.268352	0.241312
10	0.261056	0.255904	0.264896	0.133216	0.228768
25	0.259808	0.14176	0.136384	0.143232	0.170296

Time, 250x250					
GPU Tiles	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	13.69453	13.8312	13.94592	14.17875	13.9126
5	2.641664	2.514304	2.885952	2.51552	2.63936
10	0.975136	1.113792	1.196768	0.975232	1.065232
25	0.945664	0.951872	0.949056	0.862144	0.927184

Time, 250x250					
GPU Threa	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	11.0863	10.75923	9.906176	10.74902	10.62518
5	2.025504	2.923424	2.559392	2.040992	2.387328
10	0.960256	0.964096	1.50528	0.961824	1.097864
25	0.733792	0.739936	0.739424	0.754752	0.741976

Time, 500x500					
GPU Tiles	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	110.0036	110.2702	110.2576	111.0126	110.386
5	17.38419	17.69619	17.29917	17.42707	17.45166
10	6.955296	6.850592	7.197312	7.670272	7.168368
25	5.666976	5.40288	5.822592	5.3192	5.552912

Time, 500x500					
GPU Threa	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	81.72301	80.31344	80.74394	81.16781	80.98705
5	14.10966	14.23398	14.00234	13.67123	14.0043
10	4.955936	4.997568	4.990336	5.004512	4.987088
25	4.699136	5.460512	4.631296	5.145408	4.984088

Time, 1000x1000					
GPU Tiles	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	805.9958	804.2344	795.6401	800.8304	801.6752
5	136.2868	132.791	136.5395	136.3284	135.4864
10	53.46941	53.46941	53.46941	55.47379	53.9705
25	43.26717	41.36742	41.12192	41.35411	41.77766

Time, 1000x1000					
GPU Threa	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	586.8982	530.8599	566.4705	539.0612	555.8224
5	127.2612	128.7012	125.0887	124.5066	126.3894
10	43.92358	45.28125	43.44951	44.33088	44.2463
25	41.75322	40.19728	41.18055	41.1024	41.05836

Time, 1500x1500					
GPU Tiles	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	2661.523	2651.135	2646.437	2645.05	2651.036
5	418.9518	413.3042	421.0344	410.7944	416.0212
10	179.1131	175.038	175.1741	179.3036	177.1572
25	135.6812	135.5078	135.0199	132.45	134.6647

Time, 1500x1500					
GPU Threa	Trial 1	Trial 2	Trial 3	Trial 4	Average
2	1900.626	1861.793	1845.312	1845.587	1863.33
5	340.6214	358.3709	340.5679	346.5012	346.5153
10	144.9095	130.8658	132.0595	137.0765	136.2278
25	335.7727	137.9879	145.0468	137.8224	189.1575

Question 1 Code

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

// Helper function to convert compute capability to the number of cores
int ConvertSMVer2Cores(int major, int minor) {
```

```
    // Refer to NVIDIA CUDA Programming Guide for the compute capability to cores
conversion
    // This is a simplified version and may not cover all cases
    int cores;

    switch ((major << 4) + minor) {
    case 0x10:
        cores = 8;
        break;
    case 0x11:
    case 0x12:
        cores = 8;
        break;
    case 0x13:
        cores = 32;
        break;
    case 0x20:
        cores = 32;
        break;
    default:
        cores = 0;
        break;
    }

    return cores;
}

#define TILE_WIDTH 2

int main() {
    int device_id = 0; // Device ID (you can change it if you have multiple devices)
    cudaSetDevice(device_id); // Set the device to use

    cudaDeviceProp device_prop;
    cudaGetDeviceProperties(&device_prop, device_id);

    int num_SM = device_prop.multiProcessorCount;
    int max_threads_per_SM = device_prop.maxThreadsPerMultiProcessor;
    int warp_size = device_prop.warpSize;
    int max_shared_memory_per_SM = device_prop.sharedMemPerMultiprocessor;

    // Calculate the maximum number of blocks per SM based on shared memory size
constraint
    // Assuming each block uses shared memory of size 2 * TILE_WIDTH * TILE_WIDTH *
sizeof(float)
    int max_blocks_per_SM = max_shared_memory_per_SM / (2 * TILE_WIDTH * TILE_WIDTH
* sizeof(float));

    // Calculate the maximum number of threads that can be scheduled per SM
    int max_threads_per_block = max_threads_per_SM / warp_size;
    int threads_scheduled = max_blocks_per_SM * max_threads_per_block;

    printf("Number of Streaming Multiprocessors: %d\n", num_SM);
    printf("Max Threads per Multiprocessor: %d\n", max_threads_per_SM);
    printf("Warp Size: %d\n", warp_size);
    printf("Max Shared Memory per Multiprocessor: %d KB\n", max_shared_memory_per_SM
/ 1024);
    printf("Max Blocks per Streaming Multiprocessor: %d\n", max_blocks_per_SM);
```

```
    printf("Threads Scheduled: %d\n", threads_scheduled);

    return 0;
}
```

Question 2 Code

```
#include <cuda_runtime.h>
#include <stdio.h>

#define TILE_WIDTH 2 //

__global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;

    for (int ph = 0; ph < Width / TILE_WIDTH; ++ph) {
        Mds[ty][tx] = M[Row * Width + ph * TILE_WIDTH + tx];
        Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * Width + Col];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
    }
    P[Row * Width + Col] = Pvalue;
}

int main() {
    cudaDeviceProp device_prop;
    cudaGetDeviceProperties(&device_prop, 0); // Assuming device 0

    int max_shared_memory_per_SM = device_prop.sharedMemPerMultiprocessor;

    int max_blocks_per_SM = max_shared_memory_per_SM / (2 * TILE_WIDTH * TILE_WIDTH
* sizeof(float));
    int threads_per_block = device_prop.maxThreadsPerBlock;
    int total_threads = max_blocks_per_SM * threads_per_block;

    printf("Total number of registers: %d\n", device_prop.regsPerBlock * TILE_WIDTH
* TILE_WIDTH);
    printf("Shared memory size: %d bytes\n", 2 * TILE_WIDTH * TILE_WIDTH *
sizeof(float));
    printf("Number of blocks per streaming multiprocessor: %d\n",
max_blocks_per_SM);
    printf("Total threads simultaneously scheduled/executing: %d\n", total_threads);
```

```
    return 0;
}
```