

ELEC 374

Machine Problem 2

Presented To: Ahmad Afsahi
Shayan Rahman, 20282946

We do hereby verify that this written lab report is our work and contains our own original ideas, concepts, and designs. No portion of this lab report has been copied in whole or in part from another source, with the possible exception of properly referenced material.

Contents

Part 1	3
Checking Tiled Matrix Multiplication Output	3
Computation Comparison Tiled vs non-Tiled	3
Questions	6

Part 1

Checking Tiled Matrix Multiplication Output

Initially using the file named “MP2_Part1.1”, the outputs generated from the host matrix multiplication code and the device tiled matrix multiplication were compared with a tolerance of 10^{-6} units. It was determined that each comparison passed for matrix sizes of 100, 250, 500, 1000, and 1500. Therefore, the device tiled matrix multiplication implementation can be considered to be generating the same output as the host matrix multiplication implementation.

```
//GPU Tiled Matrix Multiplication Implementation
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
    int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    __shared__ float M_tile[TILE_WIDTH][TILE_WIDTH];
    __shared__ float N_tile[TILE_WIDTH][TILE_WIDTH];

    float Pvalue = 0;

    int numTiles = (Width + TILE_WIDTH - 1) / TILE_WIDTH;

    for (int t = 0; t < numTiles; ++t) {
        int mRow = row;
        int mCol = t * TILE_WIDTH + threadIdx.x;
        int nRow = t * TILE_WIDTH + threadIdx.y;
        int nCol = col;

        if (mRow < Width && mCol < Width)
            M_tile[threadIdx.y][threadIdx.x] = M[mRow * Width + mCol];
        else
            M_tile[threadIdx.y][threadIdx.x] = 0.0;

        if (nRow < Width && nCol < Width)
            N_tile[threadIdx.y][threadIdx.x] = N[nRow * Width + nCol];
        else
            N_tile[threadIdx.y][threadIdx.x] = 0.0;

        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += M_tile[threadIdx.y][k] * N_tile[k][threadIdx.x];

        __syncthreads();
    }

    if (row < Width && col < Width)
        P[row * Width + col] = Pvalue;
}
```

```
// CPU Matrix Multiply Implementation
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;
        }
    }
}
```

Figure 1: Implementations of the CPU and tiled GPU matrix multiplication

Computation Comparison Tiled vs non-Tiled

Below are images comparing the average device tiled matrix multiplication vs regular matrix multiplication execution times for differing matrix sizes for a certain tile width. These times do not include data transfer time or memory allocation/free time on the device. The regular gpu matrix multiplication times are all the same as they are not affected by changing tile widths, however the tile gpu matrix multiplication times change.

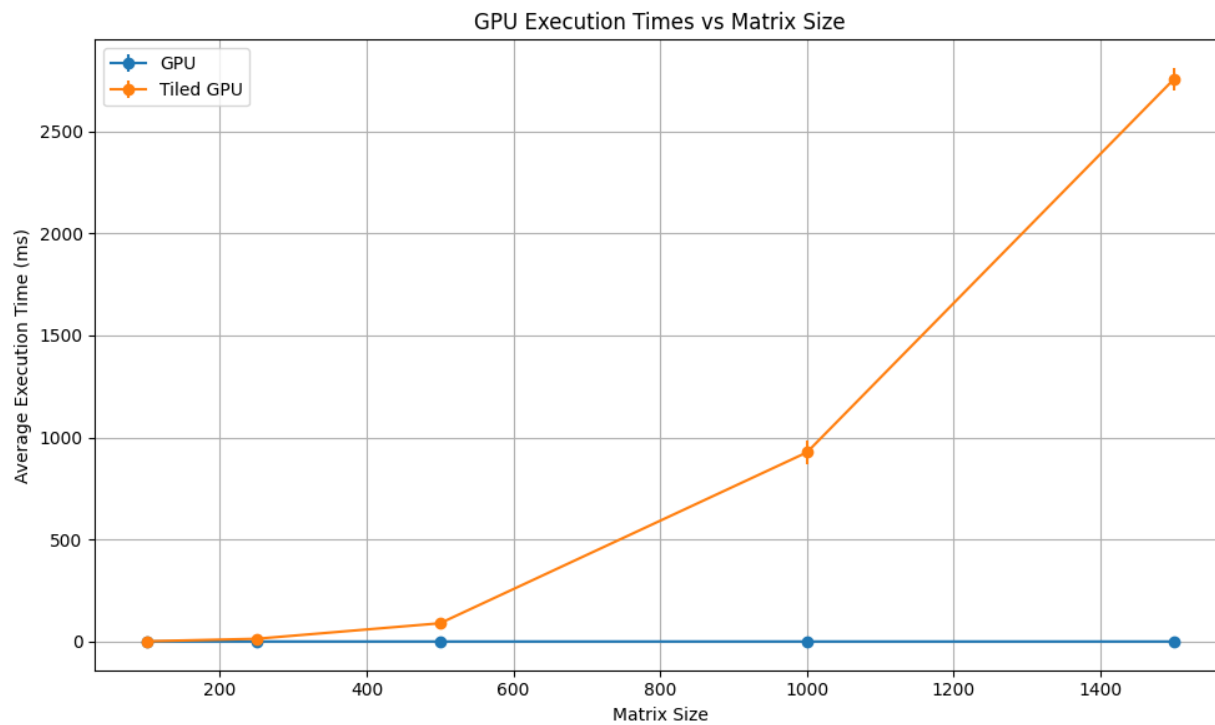


Figure 2: Execution Comparison with Tile Size 2

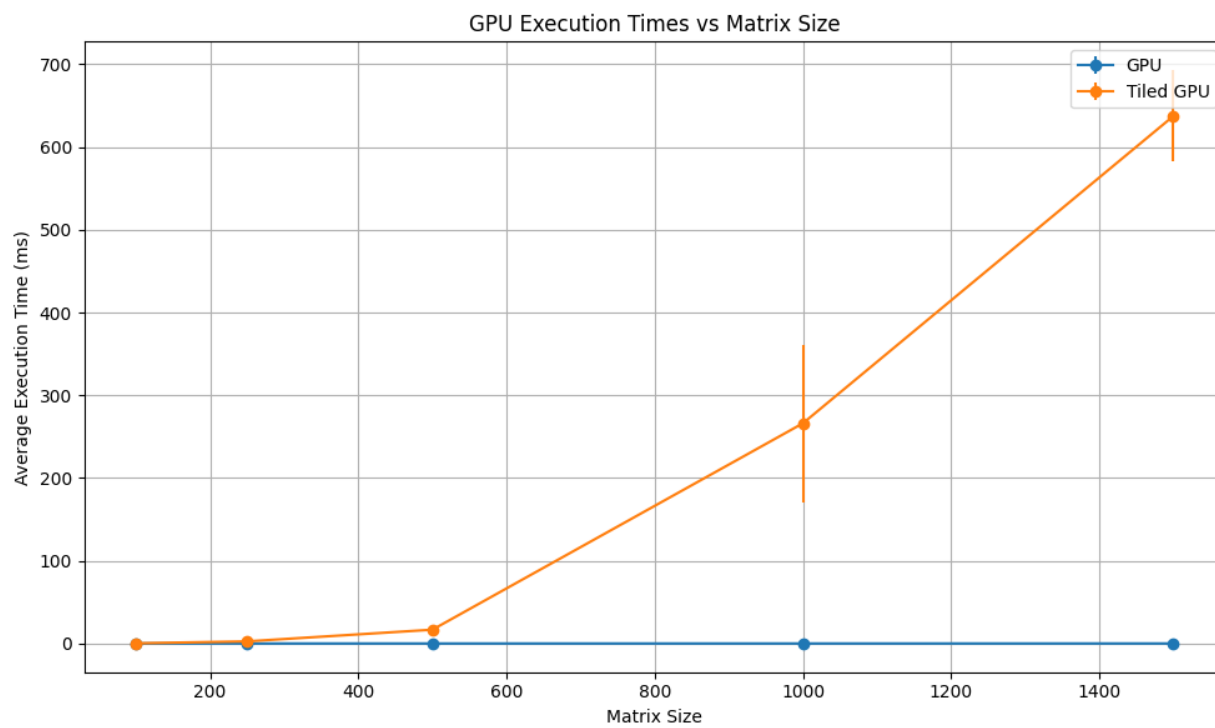


Figure 3: Execution Comparison with Tile Size 5

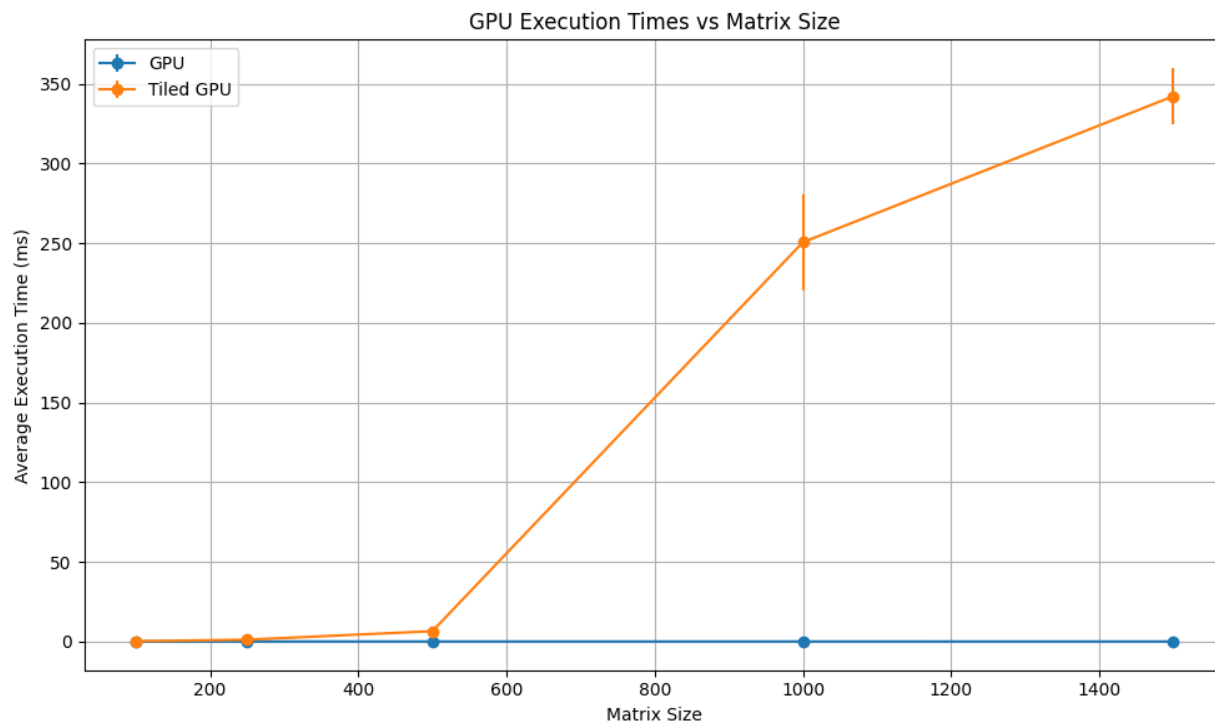


Figure 4: Execution Comparison with Tile Size 10

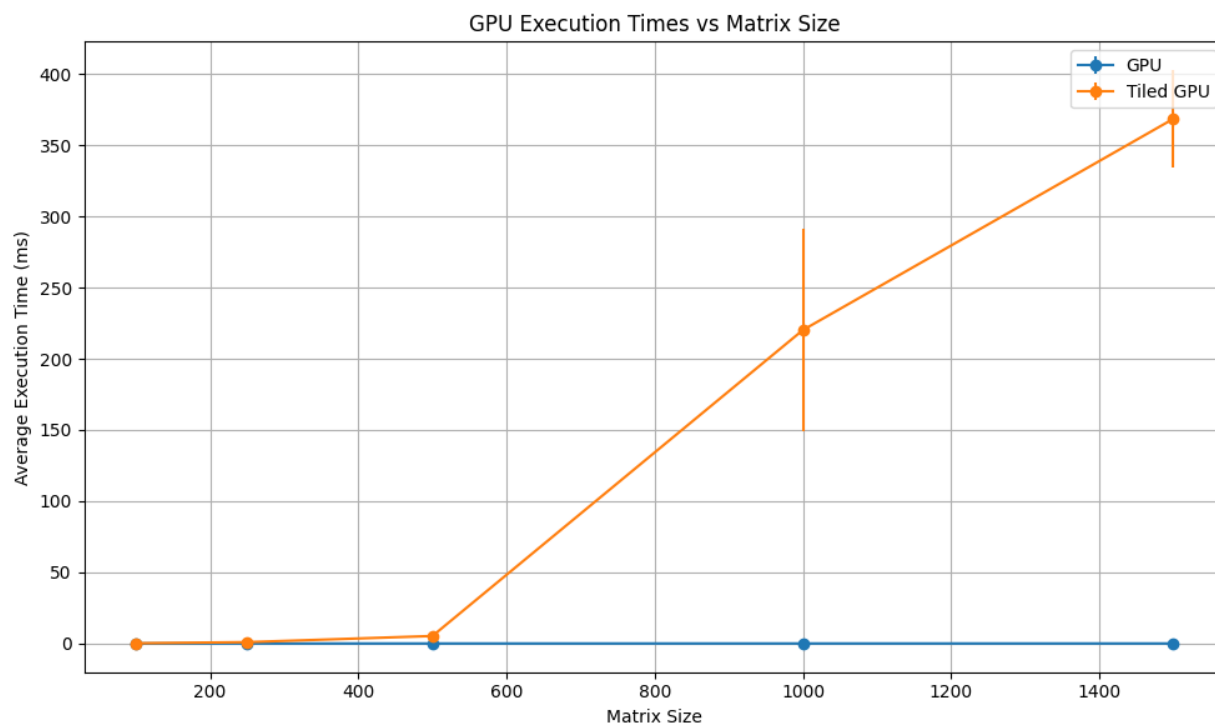


Figure 5: Execution Comparison with Tile Size 25

Overall from the graphs we can see that for matrix sizes up to 500, the tiled and regular device matrix multiplication execution times are almost equal. However, for matrix sizes larger than 500 the execution times the tiled device matrix multiplication times become larger than the regular device matrix multiplication times. Tiled matrix multiplication is typically expected to perform better for larger matrix sizes compared to regular matrix multiplication, however there are 2 factors that could explain these observations:

Shared Memory Utilization: Tiled matrix multiplication takes advantage of shared memory to reduce the number of global memory accesses by loading data into the faster shared memory. However, shared memory is limited in size and can become a limiting factor for larger matrix sizes. When the matrix size exceeds the capacity of shared memory, the tiled approach may suffer from increased memory contention and synchronization overhead, leading to longer execution times compared to regular matrix multiplication.

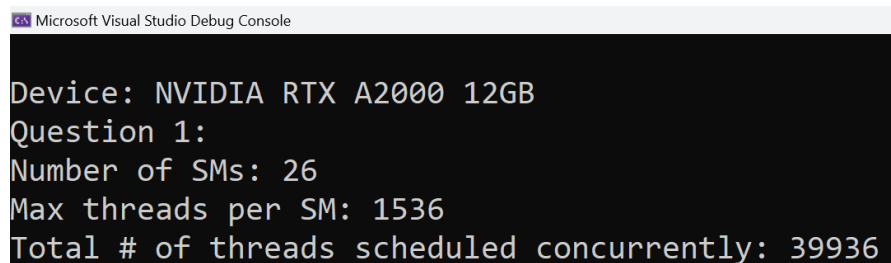
Thread-Level Parallelism: Regular matrix multiplication distributes the workload evenly across threads, but for larger matrices, the computational workload per thread increases. This can lead to better utilization of GPU resources and improved parallelism. In contrast, tiled matrix multiplication may introduce additional overhead due to thread synchronization and shared memory management, which can impact the overall performance, especially for larger matrices.

Overall, I can conclude that there are limitations and overhead associated with shared memory management and thread synchronization that can impact performance, particularly for matrix sizes larger than a certain threshold.

Questions

1. In your kernel implementation, how many threads can be simultaneously scheduled on your CUDA device, considering the number of streaming multiprocessors?

Assuming each SM can handle a certain maximum number of threads, you can calculate the maximum number of threads that can be simultaneously scheduled by multiplying the number of SMs by the maximum number of threads per SM. After outputting properties of the kernel we found the number of SMs to be 26 and the max threads to be 1536.

A screenshot of the Microsoft Visual Studio Debug Console. The window title is "Microsoft Visual Studio Debug Console". The text inside the console is as follows:

```
Device: NVIDIA RTX A2000 12GB
Question 1:
Number of SMs: 26
Max threads per SM: 1536
Total # of threads scheduled concurrently: 39936
```

Figure 6: Kernel Properties

Total Threads = Number of SMs × Maximum Threads per SM

Maximum threads simultaneously scheduled = $1536 * 26 = 39936$

So, with 38 SMs, and if each SM can handle 1024 threads, a maximum of 39,936 threads can be simultaneously scheduled on your CUDA device.

2. Find the resource usage of your kernel, including the number of registers, shared memory size, number of blocks per streaming multiprocessor, and maximum total threads simultaneously scheduled/executing.

Below are screenshots from running the file named “MP2_Part1.3” to analyze the resource usage of the device tiled matrix multiplication kernel when launched. In the images below we can see that the number of threads and number of blocks per SM remain the same for varying tile widths. We also notice that the shared memory size per block increases as the tile width increases. Also, the number of registers per thread is 29 for a tile width of 2 and 32 for tile widths of 5, 10, and 25. These results all make sense as greater tile widths require greater shared memory as the size of the M and N matrixes are relative to the tile widths and the shared memory is relative to the M and N matrixes.

```
Question 2:
Number of registers per thread: 29
Shared memory size per block: 32
Number of Blocks per SM: 1
Maximum threads per block: 1024
```

Figure 7: Resource Usage of Kernel with Tile Width 2

```
Question 2:
Number of registers per thread: 32
Shared memory size per block: 200
Number of Blocks per SM: 1
Maximum threads per block: 1024
```

Figure 8 Resource Usage of Kernel with Tile Width 5

```
Question 2:
Number of registers per thread: 32
Shared memory size per block: 800
Number of Blocks per SM: 1
Maximum threads per block: 1024
```

Figure 9: Resource Usage of Kernel with Tile Width 10

```
Question 2:
Number of registers per thread: 32
Shared memory size per block: 5000
Number of Blocks per SM: 1
Maximum threads per block: 1024
```

Figure 10: Resource Usage of Kernel with Tile Width 25