# Machine Problem 2

Jacob Badali ~ 20290739 ~ March 29th, 2024

# 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.



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.



Figure 2 100x100 threaded matrix multiplication

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

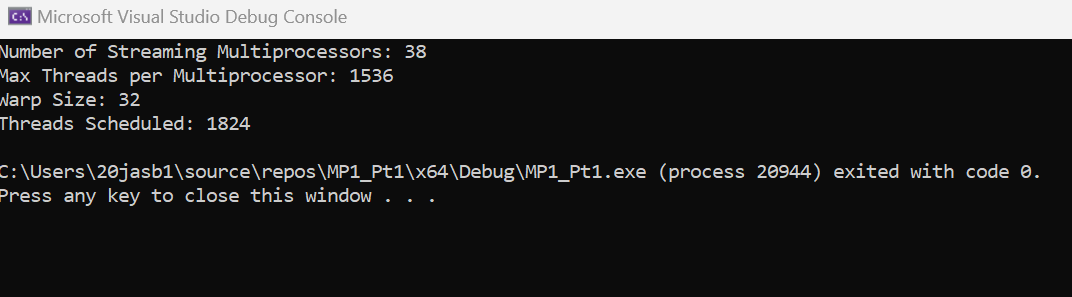
Figure 3 Threads vs Tiles 100x100 size

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 using the equation



## Question 2

# 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









