Joshua Caudill

CS6068

Assignment #5

Table of Contents

[Introduction 4](#_Toc178517374)

[Tools 4](#_Toc178517375)

[Code 4](#_Toc178517376)

[Results 9](#_Toc178517377)

[Conclusion 10](#_Toc178517378)

Table of Figures

[Figure 1: Speedups for Various Transpose Operations 9](#_Toc178517287)

# Introduction

The goal of this assignment was to write two tiled versions of the matrix transpose operation in CUDA. The first version of the matrix transpose operation used global memory. The second version of the matrix transpose operation used shared memory. The matrix transpose operations were launched with one thread per element in K by K blocks. The two tiled versions of the matrix transpose operation were compared against a serial matrix transpose operation and a parallel per row matrix transpose operation.

# Tools

* gputimer.h
* transpose.cu
* CUDA
* Pitzer Desktop (1 GPU, 48 Cores, 1 Visualization Node)

# Code

#include <stdio.h>

#include "gputimer.h"

//#include "utils.h"

const int N= 1024;  // matrix size will be NxN

const int K= 16;

int compare\_matrices(float \*gpu, float \*ref, int N)

{

        int result = 0;

        for(int j=0; j < N; j++)

        for(int i=0; i < N; i++)

                if (ref[i + j\*N] != gpu[i + j\*N])

                   {result = 1;}

 return result;

}

// fill a matrix with sequential numbers in the range 0..N-1

void fill\_matrix(float \*mat, int N)

{

        for(int j=0; j < N \* N; j++)

                mat[j] = (float) j;

}

// The following functions and kernels are for your references

void

transpose\_CPU(float in[], float out[])

{

    for(int j=0; j < N; j++)

        for(int i=0; i < N; i++)

            out[j + i\*N] = in[i + j\*N]; // out(j,i) = in(i,j)

}

// to be launched on a single thread

\_\_global\_\_ void

transpose\_serial(float in[], float out[])

{

    for(int j=0; j < N; j++)

        for(int i=0; i < N; i++)

            out[j + i\*N] = in[i + j\*N]; // out(j,i) = in(i,j)

}

// to be launched with one thread per row of output matrix

\_\_global\_\_ void

transpose\_parallel\_per\_row(float in[], float out[])

{

    int i = threadIdx.x + blockDim.x \* blockIdx.x;

    for(int j=0; j < N; j++)

        out[j + i\*N] = in[i + j\*N]; // out(j,i) = in(i,j)

}

// Write two tiled versions of transpose -- One using shared memory.

// To be launched with one thread per element, in KxK threadblocks.

// You will determine for each thread (x,y) in tile the element (i,j) of global output matrix.

\_\_global\_\_ void

transpose\_parallel\_per\_element\_tiled(float in[], float out[])

{

    // Replace blockDim.x and blockDim.y with K

    int i = threadIdx.x + blockIdx.x \* K;

    int j = threadIdx.y + blockIdx.y \* K;

    // Loop unnecessary when parallelizing per element

    out[j + i\*N] = in[i + j\*N];

}

\_\_global\_\_ void

transpose\_parallel\_per\_element\_tiled\_shared(float in[], float out[])

{

    \_\_shared\_\_ int tile[K][K]; // Create shared tile

    // Grab thread indices

    int x = threadIdx.x;

    int y = threadIdx.y;

    // Replace blockDim.x and blockDim.y with K

    int i = blockIdx.x \* K;

    int j = blockIdx.y \* K;

    tile[y][x] = in[(i+x) + (j+y)\*N]; // Transpose matrix stored shared tile

    \_\_syncthreads(); // Synchronize threads before writing to global memory

    out[(j+x) + (i+y)\*N] = tile[x][y]; // Write to global memory

}

int main(int argc, char \*\*argv)

{

    int numbytes = N \* N \* sizeof(float);

    float \*in = (float \*) malloc(numbytes);

    float \*out = (float \*) malloc(numbytes);

    float \*gold = (float \*) malloc(numbytes);

    fill\_matrix(in, N);

    transpose\_CPU(in, gold);

    float \*d\_in, \*d\_out;

    cudaMalloc(&d\_in, numbytes);

    cudaMalloc(&d\_out, numbytes);

    cudaMemcpy(d\_in, in, numbytes, cudaMemcpyHostToDevice);

    GpuTimer timer;

    timer.Start();

    transpose\_serial<<<1,1>>>(d\_in, d\_out);

    timer.Stop();

    float serial\_time = timer.Elapsed();

    for (int i=0; i < N\*N; ++i){out[i] = 0.0;}

    cudaMemcpy(out, d\_out, numbytes, cudaMemcpyDeviceToHost);

    printf("transpose\_serial: %g ms.\nVerifying ...%s\n",

           serial\_time, compare\_matrices(out, gold, N) ? "Failed" : "Success");

    cudaMemcpy(d\_out, d\_in, numbytes, cudaMemcpyDeviceToDevice); //clean d\_out

    timer.Start();

    transpose\_parallel\_per\_row<<<1,N>>>(d\_in, d\_out);

    timer.Stop();

    float parallel\_time = timer.Elapsed();

    for (int i=0; i < N\*N; ++i){out[i] = 0.0;}  //clean out

    cudaMemcpy(out, d\_out, numbytes, cudaMemcpyDeviceToHost);

    printf("transpose\_parallel\_per\_row: %g ms.\nVerifying ...%s\n",

            parallel\_time, compare\_matrices(out, gold, N) ? "Failed" : "Success");

    printf("speedup: %0.3f\n", (serial\_time/parallel\_time));

    cudaMemcpy(d\_out, d\_in, numbytes, cudaMemcpyDeviceToDevice); //clean d\_out

    // Tiled versions

    dim3 blocks\_tiled(N/K,N/K);

    dim3 threads\_tiled(K,K);

    timer.Start();

    transpose\_parallel\_per\_element\_tiled<<<blocks\_tiled,threads\_tiled>>>(d\_in, d\_out);

    timer.Stop();

    parallel\_time = timer.Elapsed();

    for (int i=0; i < N\*N; ++i){out[i] = 0.0;}

    cudaMemcpy(out, d\_out, numbytes, cudaMemcpyDeviceToHost);

    printf("transpose\_parallel\_per\_element\_tiled %dx%d: %g ms.\nVerifying ...%s\n",

           K, K, parallel\_time, compare\_matrices(out, gold, N) ? "Failed" : "Success");

    printf("speedup: %0.3f\n", (serial\_time/parallel\_time));

    cudaMemcpy(d\_out, d\_in, numbytes, cudaMemcpyDeviceToDevice); //clean d\_out

    dim3 blocks\_tiled\_sh(N/K,N/K);

    dim3 threads\_tiled\_sh(K,K);

    timer.Start();

    transpose\_parallel\_per\_element\_tiled\_shared<<<blocks\_tiled\_sh,threads\_tiled\_sh>>>(d\_in, d\_out);

    timer.Stop();

    parallel\_time = timer.Elapsed();

    for (int i=0; i < N\*N; ++i){out[i] = 0.0;}

    cudaMemcpy(out, d\_out, numbytes, cudaMemcpyDeviceToHost);

    printf("transpose\_parallel\_per\_element\_tiled\_shared %dx%d: %g ms.\nVerifying ...%s\n",

           K, K, timer.Elapsed(), compare\_matrices(out, gold, N) ? "Failed" : "Success");

    printf("speedup: %0.3f\n", (serial\_time/parallel\_time));

    cudaFree(d\_in);

    cudaFree(d\_out);

}

# Results

First, transpose.cu was executed on the Pitzer Desktop. The results are shown in Figure 1. The parallel per row matrix transpose operation achieved a speedup of 30.133 compared to the serial matrix transpose operation. The tiled matrix transpose operation that used global memory achieved a speedup of 1753.729 compared to the serial matrix transpose operation. The tiled matrix transpose operation that used shared memory achieved a speedup of 3162.181 compared to the serial matrix transpose operation.

A screenshot of a computer

Description automatically generated

Figure 1: Speedups for Various Transpose Operations

# Conclusion

The goal of this assignment was to write two tiled versions of the matrix transpose operation in CUDA. The first version of the matrix transpose operation used global memory. The second version of the matrix transpose operation used shared memory. The two tiled versions of the matrix transpose operation were compared against a serial matrix transpose operation and a parallel per row matrix transpose operation. It was demonstrated that the tiled matrix transpose operation that used shared memory achieved the fastest speedup.