Joshua Caudill

CS6068

Assignment #5

Joshua Caudill - CS6068 - Assignment #5

Table of Contents

Table of Figures

Joshua Caudill - CS6068 - Assignment #5

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
```

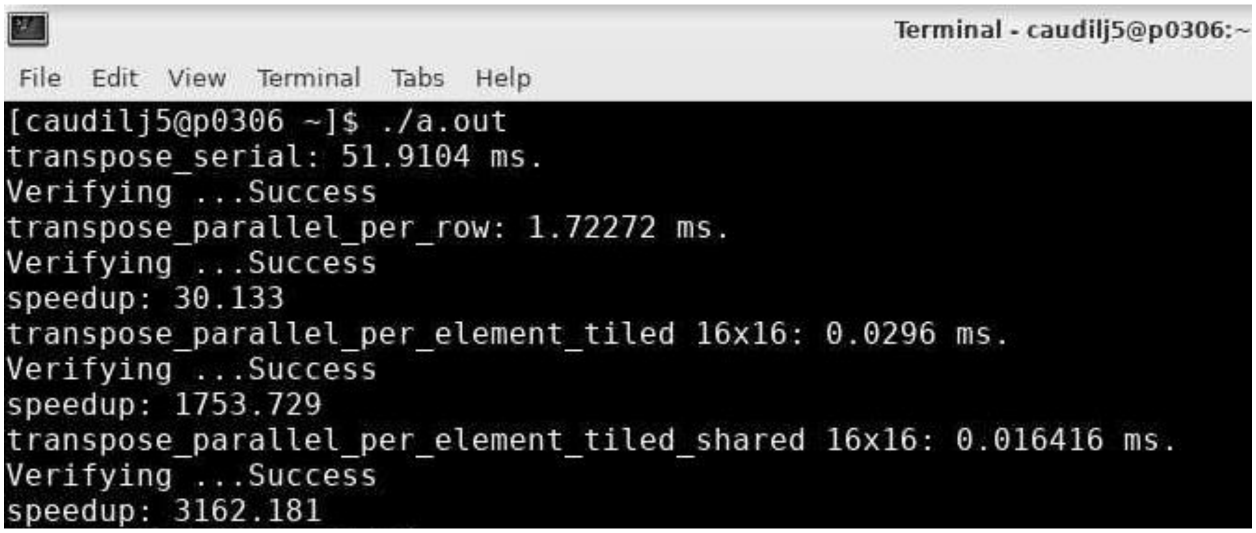
```
    // Tiled versions
    dim3 blocks_tiled(N/K,N/K);
    dim3 threads_tiled(K,K);
    timer.Start();
    transpose_parallel_per_element_tiled<<<blocks_tiled,threads_tiled>>>(d_in, d_out);
    timer.Stop();
    parallel_time = timer.Elapsed();
    for (int i=0; i < N*N; ++i){out[i] = 0.0;}
    cudaMemcpy(out, 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.


Terminal - caudilj5@p0306:~
File Edit View Terminal Tabs Help

```
[caudilj5@p0306 ~]$ ./a.out
transpose_serial: 51.9104 ms.
Verifying ...Success
transpose_parallel_per_row: 1.72272 ms.
Verifying ...Success
speedup: 30.133
transpose_parallel_per_element_tiled 16x16: 0.0296 ms.
Verifying ...Success
speedup: 1753.729
transpose_parallel_per_element_tiled_shared 16x16: 0.016416 ms.
Verifying ...Success
speedup: 3162.181
```

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.