ELEC6230 Homework 9

Answers to Parallel Processing Homework 9

Michael J. Carroll

November 12, 2010

Part 1 Results

For Homework 9, I reimplemented the previous week's homework assignment. The addition for this week's code was the ability to cache matrix values from global memory into the much faster shared memory. To accomplish this, each spot in two submatricies was copied from the global matricies by each thread in a block. The multiplication was then performed over the block. The results of using block size B = 4, 8, and 16 are included below, and compared to the previous results without using shared memory. The code for this implementation is attached in Appendix A.

The speedup from using the shared memory is significant. The reason is that the processors can spend less time fetching from the memory (due to reduced latency), and more time performing the actual matrix multiplication. The ratio of memory access to actual arithmetic is what defines the performance on the GPU processing.

I additionally got to use the NVidia visual profiler to see where (if any) choke points existed in my code implementation. While I didn't find any, I experimented with using the "#pragma unroll" command on the inner for loop in the kernel. The pragma did not provide any noticeable performance increase (under 0.5%) in the GPU.

Another interesting result is that the larger the block size, the higher the speedup. This is counter to previous programming environments (OpenMP, MPI). The GPU is more fully utilized (almost 100% with a 16x16 block size) when more processes are added. This is a good example of optimizing the block and grid sizes to the GPU's requirements.

Part 2 Results

I was wholly unsuccessful in implementing Part 2 of the homework.

Theoretically, the performance should increase as the window size increases, up to a point. The Nvidia visual profiling tool revealed to me that not all of the shared memory was being used during the execution of the matrix multiplication. Because of this, we can cache more into the the shared memory over

Program Type	Sequential	Parallel (4)	Parallel (8)	Parallel (16)
Global Memory Implementation	34.47 s	19.96 s	8.09 s	5.75
Shared Memory Implementation	34.47 s	5.41 s	1.42 s	0.85 s
Speedup	1.00	6.37	24.27	40.55

Table 1: Parallel Speedup

the course of the operation of a single thread.

By using as much shared memory as possible (and as a result using the fastest memory possible), we can further reduce the ratio of memory fetch time to actual execution instructions. This would give additional performance enhancements, although not on the order of switching from global to shared memory.

At a certain point, this performance would degrade as the shared memory would get full. I would estimate that a window of 4x4 or 8x8 would give the best performance in terms of speedup while maintaining correctness.

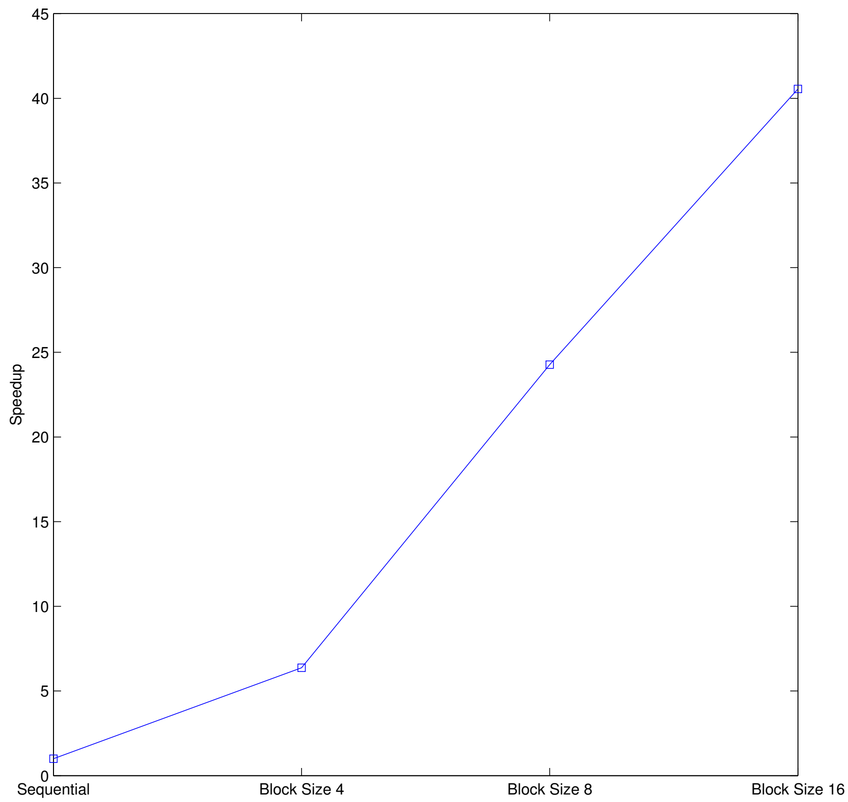

Figure 1: Speedup Plot

Appendix A - Global Memory CUDA Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>
#include <sys/time.h>

#define BLOCK_SIZE 16
#define MATRIX_SIZE 4096
#define WINDOW_SIZE 2

bool InitCUDA(void)
{
        int count = 0;int i = 0;
        cudaGetDeviceCount(&count);
        if(count == 0) {
                fprintf(stderr, "There␣is␣no␣device.\n");
                return false;
        }
        for(i = 0; i < count; i++) {
                cudaDeviceProp prop;
                if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
                        if(prop.major >= 1) {
                                break;
        }}}
        if(i == count) {
                fprintf(stderr, "There␣is␣no␣device␣supporting␣CUDA.\n");
                return false;
        }
        cudaSetDevice(i);
        printf("CUDA␣initialized.\n");
        return true;
}

__global__ void MatMulKernel(float* Md, float* Nd, float* Pd)
{
    int tx = threadIdx.x; int ty = threadIdx.y;
    int bx = blockIdx.x; int by = blockIdx.y;

    float Pvalue = 0;

    for(int m = MATRIX_SIZE * BLOCK_SIZE * by, n = BLOCK_SIZE * bx;
            m <= MATRIX_SIZE * BLOCK_SIZE * by + MATRIX_SIZE -1;
            m += BLOCK_SIZE, n += BLOCK_SIZE * MATRIX_SIZE)
    {
        __shared__ float Mds[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Nds[BLOCK_SIZE][BLOCK_SIZE];

        Mds[ty][tx] = Md[m + MATRIX_SIZE * ty + tx];
        Nds[ty][tx] = Nd[n + MATRIX_SIZE * ty + tx];

        // Make sure that all the threads have copied to shared memory before
       // performing the actual multiplication
```

```
        __syncthreads();

#pragma unroll
        for( int k = 0; k < BLOCK_SIZE; ++k)
        {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }

        // Syncronize after the multiplication.
        __syncthreads();
    }
    Pd[MATRIX_SIZE * BLOCK_SIZE * by +
        BLOCK_SIZE * bx +
        MATRIX_SIZE * ty
        + tx] = Pvalue;
}

int main(int argc, char* argv[])
{
    struct timeval t0,t1;
    // Initialize CUDA using the ASC helper function
    if(!InitCUDA()) {
            return 0;
    }
    // Define some sizes for malloc
    unsigned int size = MATRIX_SIZE * MATRIX_SIZE;
    unsigned int mem_size = sizeof(float) * size;
    // Declare the variables to be used
    float* A = (float*) malloc(mem_size);
    float* B = (float*) malloc(mem_size);
    float* C = (float*) malloc(mem_size);
    float* Md;
    float* Nd;
    float* Pd;
    // Initialize the A and B matricies to the homework specifications
    int row,col;
    for( int i=0; i<size; i++)
    {
        row = i/MATRIX_SIZE;
        col = i%MATRIX_SIZE;
        A[i] = ((row + 1.0)*(col + 1.0))/MATRIX_SIZE;
        B[i] = (col + 1.0)/(row + 1.0);
    }

    gettimeofday(&t0,0);
    // Allocate the matricies on the video card
    cudaMalloc((void**) &Md, mem_size);
    cudaMalloc((void**) &Nd, mem_size);
    cudaMalloc((void**) &Pd, mem_size);
    // Copy the matricies to the video card
    cudaMemcpy(Md, A, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(Nd, B, mem_size, cudaMemcpyHostToDevice);
    // Perform the Kernel
    dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
```

```
    dim3 dimGrid(MATRIX_SIZE/dimBlock.x,MATRIX_SIZE/dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(Md,Nd,Pd);
    // Copy the results
    cudaMemcpy(C, Pd, mem_size, cudaMemcpyDeviceToHost);
    // Clear the memory on the video card
    cudaFree(Md);cudaFree(Nd);cudaFree(Pd);

    gettimeofday(&t1,0);
    // Print a 16x16 "test section" to prove results are correct.
    for( int i=0; i<16; i++){
        for( int j=0; j<16; j++){
            printf("%6.2f␣",C[j*MATRIX_SIZE+i]);
        }
        printf("\n");
    }

    printf("\nTime␣Results\n");
    float totalInt = t1.tv_sec - t0.tv_sec + (t1.tv_usec - t0.tv_usec)*1.0E-06;
    printf("Total␣Execution␣Time:\t%e\n",totalInt);

    free(A);free(B);free(C);
    return 0;
}
```