ECEC 413: Parallel Computer Architecture

Programming Assignment 8

CUDA: Gaussian Elimination

Tri Pham (EE), Dinh Nguyen (CE), Manh Cuong Phi (CE)

Instructor: Prof. Naga Kandasamy

ECE Department

Drexel University

March 5, 2021

Contents

Introduction

Overview

In mathematics, Gaussian elimination, also known as row reduction, is an algorithm for solving systems of linear equations. It consists of a sequence of operations performed on the corresponding matrix of coefficients.

$$
\begin{array}{lllll}
a_{0,0}x_0 & + a_{0,1}x_1 & + \cdots & + a_{0,n-1}x_{n-1} & = b_0, \\
a_{1,0}x_0 & + a_{1,1}x_1 & + \cdots & + a_{1,n-1}x_{n-1} & = b_1, \\
\cdot & \cdot & & \cdot & \cdot \\
\cdot & \cdot & & \cdot & \cdot \\
a_{n-1,0}x_0 & + a_{n-1,1}x_1 & + \cdots & + a_{n-1,n-1}x_{n-1} & = b_{n-1}.
\end{array}
$$

In this assignment we are going to implement the Gaussian algorithm with a multithreading approach. We will be testing out the performance of multi-threading CUDA implementation on GPU to compare with the CPU performance. The upper-diagonal matrix generated by the GPU is compared against the CPU result and if the solutions match within a certain tolerance, the application will print out “Test PASSED” to the screen before exiting to make sure the implementation was done correctly

Project Description

Implementation

CUDA Deployment

We first start computing using CUDA by allocating memory and copy data and matrix onto the device. Grid size is fixed to 32 and the block size can be 8, 16 or 32.

```
struct timeval start, stop;

Matrix U_device = allocate_matrix_on_gpu(A);
copy_matrix_to_device(U_device, A);

dim3 threads(THREAD_BLOCK_SIZE, 1);
fprintf(stderr, "Setting up a 32 x 1 grid of blocks with %d x 1 threads each block\n", THREAD_BLOCK_SIZE);
dim3 grid(32, 1);

gettimeofday(&start, NULL);
/* Launch kernel */
```

CUDA memory and array allocation code

```
/* Launch kernel */
for (int k = 0; k < MATRIX_SIZE; k++){
    division__kernel<<< grid, threads >>>(U_device.elements, MATRIX_SIZE, k);
    cudaDeviceSynchronize();
    // #ifdef DEBUG
    // // if (k==0){
    //     printf("DEBUG: GPU Division step k=%d\n",k);
    //     copy_matrix_from_device(U, U_device);
    //     // cudaMemcpy(U.elements, U_device, bytes, cudaMemcpyDeviceToHost);
    //     print_matrix(U);
    // // }
    // #endif
    elimination__kernel<<< grid, threads >>>(U_device.elements, MATRIX_SIZE, k);
    cudaDeviceSynchronize();
    // #ifdef DEBUG
    //     printf("DEBUG: GPU Elimination step k=%d\n",k);
    //     copy_matrix_from_device(U, U_device);

    //     // cudaMemcpy(U.elements, U_device, bytes, cudaMemcpyDeviceToHost);
    //     print_matrix(U);
    // #endif
}
```

Kernel Execution Code

```
gettimeofday(&stop, NULL);
fprintf(stderr, "GPU Execution time = %fs\n", (float)(stop.tv_sec - start.tv_sec +\
                                              (stop.tv_usec - start.tv_usec)/(float)1000000));

check_CUDA_error("Error in overall kernel");
/* Copy matrix back from device */
copy_matrix_from_device(U, U_device);

/* Free matrix on device */
free_matrix_on_device(&U_device);
```

For k ranging from 0 to n − 1, the Gaussian elimination procedure will be processed in separated kernel: division and elimination

Division and Elimination

The division kernel takes in the output matrix to identify pivot element then divide elements of current row k by pivot element. Cuda barrier synchronization was used to ensure that all threads have completed iteration i before starting iteration i + 1.

```
// FIX ME
__global__ void division__kernel(float *U, int matrix_dim, int k)
{
    float pivot = U[matrix_dim * k + k];

    int tid = (blockIdx.x*blockDim.x) + threadIdx.x;
    int stride = gridDim.x * blockDim.x;
    while (k+1 + tid < matrix_dim){
      float temp = U[k * matrix_dim + k+1 + tid];
      U[k * matrix_dim + k+1 + tid] = (float) temp/pivot;
      tid = tid + stride;
    }
    __syncthreads();
    if (blockIdx.x == 0 && threadIdx.x == 0)
      U[matrix_dim * k + k] = 1;
    return;
}
```

Division kernel code

```
// FIX ME
__global__ void elimination__kernel(float *U, int matrix_dim, int k)
{
    int row = k+1 + blockIdx.x;
    while (row < matrix_dim){
      int tid = threadIdx.x;
      while (k+1 + tid < matrix_dim){
        // a - bc = x -> x = b(a/b - c)
        float a = U[row * matrix_dim + k+1 + tid];
        float b = __fmul_rn(U[row * matrix_dim + k], U[k * matrix_dim + k+1 +tid]);
        U[row * matrix_dim + k+1 + tid] = a - b;
        tid = tid + blockDim.x;
      }
      __syncthreads();
      if (threadIdx.x == 0){
        U[row * matrix_dim + k] = 0;
      }
      row += gridDim.x;
    }
    return;
}
```

Elimination kernel code

Elimination kernel is used to identify pivot and eliminate rows under the current row k once the division step is done. Again, Cuda barrier synchronization was used to ensure that all threads have completed iteration i before starting iteration i + 1. Because Cuda handles floats in lesser bits than the CPU for normal operations like multiplication, subtraction, addition or division, to avoid losing floating point over operations, we used cuda math API __fmul_rn to perform multiplications between two floats and rounded it to the nearest even.

Performance Analysis

In the multi-threads method, due to the overhead when creating threads and updating the matrix, the single thread method is expected to be faster than the multi-threads

method for smaller computation scale, therefore, to see the performance benefit from multi-threading, the computational scale has to be large enough. We've tested the program with the matrices' size of 512 x 512, 1024 x 1024, 2048 x 2048 using 8, 16, and 32 threads block size. Below are the results for each case:

```
Gaussian Elimination for Matrix: 512 x 512
Gaussian elimination on the CPU was successful.
CPU Execution time = 0.083379s
Setting up a 32 x 1 grid of blocks with 8 x 1 threads each block
GPU Execution time = 0.038562s
Max epsilon = 0.000000.
Test PASSED
```

512 x 512 matrix with 8 x 1 thread blocks

```
Gaussian Elimination for Matrix: 512 x 512
Gaussian elimination on the CPU was successful.
CPU Execution time = 0.083780s
Setting up a 32 x 1 grid of blocks with 16 x 1 threads each block
GPU Execution time = 0.024957s
Max epsilon = 0.000000.
Test PASSED
```

512 x 512 matrix with 16 x 1 thread blocks

```
Gaussian Elimination for Matrix: 512 x 512
Gaussian elimination on the CPU was successful.
CPU Execution time = 0.083911s
Setting up a 32 x 1 grid of blocks with 32 x 1 threads each block
GPU Execution time = 0.017893s
Max epsilon = 0.000000.
Test PASSED
```

512 x 512 matrix with 32 x 1 thread blocks

```
Gaussian Elimination for Matrix: 1024 x 1024
Gaussian elimination on the CPU was successful.
CPU Execution time = 0.581049s
Setting up a 32 x 1 grid of blocks with 8 x 1 threads each block
GPU Execution time = 0.341712s
Max epsilon = 0.000000.
Test PASSED
```

1024 x 1024 matrix with 8 x 1 thread blocks

```
Gaussian Elimination for Matrix: 1024 x 1024
Gaussian elimination on the CPU was successful.
CPU Execution time = 0.582316s
Setting up a 32 x 1 grid of blocks with 16 x 1 threads each block
GPU Execution time = 0.189231s
Max epsilon = 0.000000.
Test PASSED
```

1024 x 1024 matrix with 16 x 1 thread blocks

```
Gaussian Elimination for Matrix: 1024 x 1024
Gaussian elimination on the CPU was successful.
CPU Execution time = 0.569305s
Setting up a 32 x 1 grid of blocks with 32 x 1 threads each block
GPU Execution time = 0.106670s
Max epsilon = 0.000000.
Test PASSED
```

1024 x 1024 matrix with 32 x 1 thread blocks

```
Gaussian Elimination for Matrix: 2048 x 2048
Gaussian elimination on the CPU was successful.
CPU Execution time = 4.148756s
Setting up a 32 x 1 grid of blocks with 8 x 1 threads each block
GPU Execution time = 2.795094s
Max epsilon = 0.000000.
Test PASSED
```

2048 x 2048 matrix with 8 x 1 thread blocks

```
Gaussian Elimination for Matrix: 2048 x 2048
Gaussian elimination on the CPU was successful.
CPU Execution time = 4.131399s
Setting up a 32 x 1 grid of blocks with 16 x 1 threads each block
GPU Execution time = 1.476668s
Max epsilon = 0.000000.
Test PASSED
```

2048 x 2048 matrix with 16 x 1 thread blocks

```
Gaussian Elimination for Matrix: 2048 x 2048
Gaussian elimination on the CPU was successful.
CPU Execution time = 4.235225s
Setting up a 32 x 1 grid of blocks with 32 x 1 threads each block
GPU Execution time = 0.811015s
Max epsilon = 0.000000.
Test PASSED
```

2048 x 2048 matrix with 32 x 1 thread blocks

		Run Time (sec)	
Matrix Size	**Block Size**	**GPU**	**CPU**
512 x 512	8 x 1 threads	0.038	0.083
	16 x 1 threads	0.024	
	32 x 1 threads	0.017	
1024 x 1024	8 x 1 threads	0.341	0.582
	16 x 1 threads	0.189	
	32 x 1 threads	0.106	
2048 x 2048	8 x 1 threads	2.795	4.131
	16 x 1 threads	1.746	
	32 x 1 threads	0.811	

Comparing the performance between the CPU and GPU, the run time on the GPU was faster than on the CPU up to 409%. The larger the matrix size, the bigger the performance increase. Also, there is a huge performance difference between using different sizes of thread block. The bigger block size increased the performance efficiency up 244%. Specifically as seen in the performance table above, the 8 x 1 block's execution time was at 2.795 sec while the 32 x 1 block ran for only 0.811 second which is approximately correct as we increased the threads block size 4 times from 8 to 32. This is because larger block sizes contain more threads which equate to more computing power.

Conclusion

We have successfully implemented the Gaussian Elimination on the GPU. We created two separate kernels to perform the Gaussian elimination which is division and elimination kernels. Cuda barrier synchronization was used in both kernels to ensure that all threads have completed the first iteration before running the second iteration and so on. The performance gained from this implementation was enormous. This is faster than the serial version on the CPU up to 409% and we can continue to see the performance gap grow bigger as the calculation scale increases.