DREXEL UNIVERSITY
Electrical and
Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: Final Exam Problem 2: Gaussian Elimination

GROUP MEMBERS: Gregory Matthews and Mark Klobukov

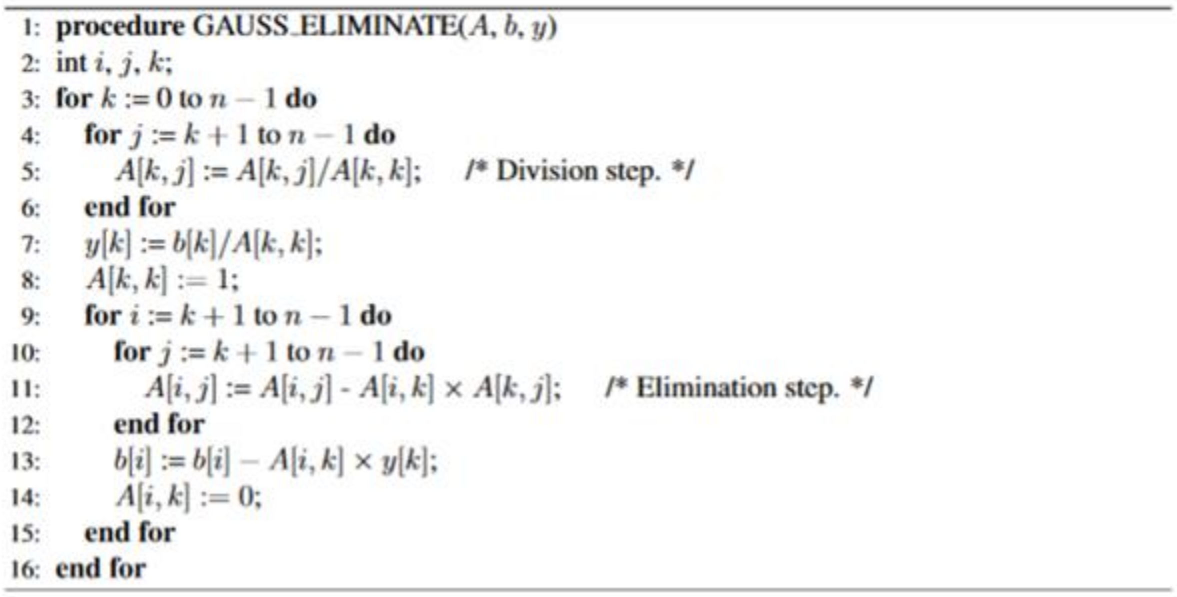
INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 3/25/2017

DATE DUE: 3/25/2017

Assignment description:

This problem involved parallelizing a given program that performs Gaussian elimination of an NxN system of linear equations using CUDA. The algorithm is shown in the pseudo-code below:



```
1:  procedure GAUSS_ELIMINATE($A$, $b$, $y$)
2:  int $i$, $j$, $k$;
3:  for $k := 0$ to $n-1$ do
4:     for $j := k+1$ to $n-1$ do
5:        $A[k,j] := A[k,j]/A[k,k]$;     /* Division step. */
6:     end for
7:     $y[k] := b[k]/A[k,k]$;
8:     $A[k,k] := 1$;
9:     for $i := k+1$ to $n-1$ do
10:       for $j := k+1$ to $n-1$ do
11:          $A[i,j] := A[i,j] - A[i,k] \times A[k,j]$;     /* Elimination step. */
12:       end for
13:       $b[i] := b[i] - A[i,k] \times y[k]$;
14:       $A[i,k] := 0$;
15:    end for
16: end for
```

Parallelization approach:

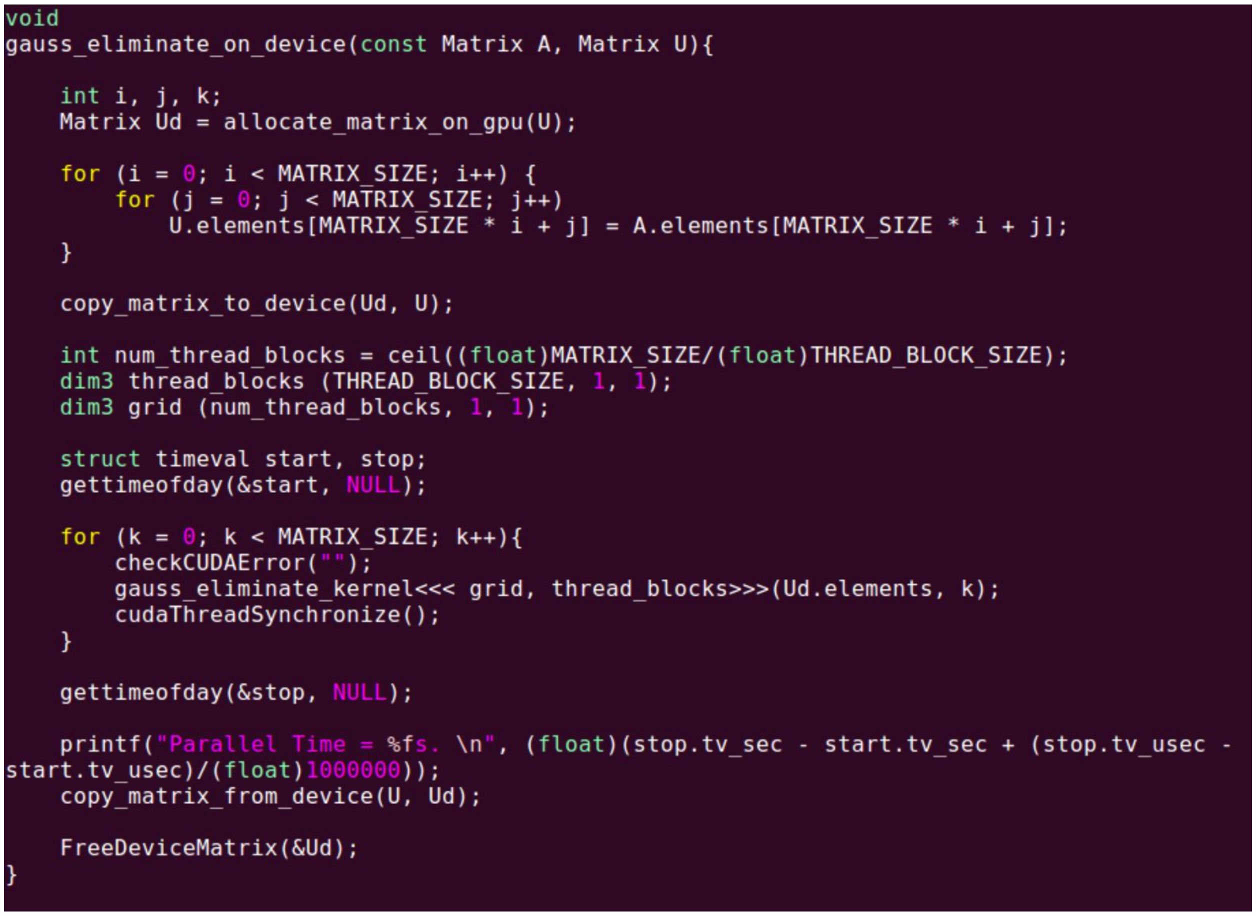
The outermost for-loop (on line 3) cannot be parallelized. The reason for this is that each successive iteration of this loop operates on the values of the matrix A that were updated in the previous iteration. This is a successive procedure.

The for-loops on lines 4 and 11, on the contrary, can be parallelized. We created one-dimensional, horizontal thread blocks that cover the entire row of the matrix, and move one row down for each call of the kernel. The division step is parallelized by having one GPU thread divided its corresponding matrix element by the pivot column element A[k,k]. The loop on line 10 is deleted because the CUDA threads already cover a given row in its entirety, and only column-wise iteration is necessary.

Memory usage optimization is achieved by using shared memory and row-wise orientation of the thread blocks. The shared memory stores: 1) The pivot element A[k,k], by which each element in a given row is divided on line 5; 2) elements A[k,j] for each corresponding thread block so they can be repeatedly used in line 11 during elimination. Horizontal orientation of the thread blocks allows us to exploit the principle of spacial locality of the L2 cache of the GPU, which will help reduce global memory access times by increasing the likelihood that a given element of the matrix is in the cache.

Host Side Function:

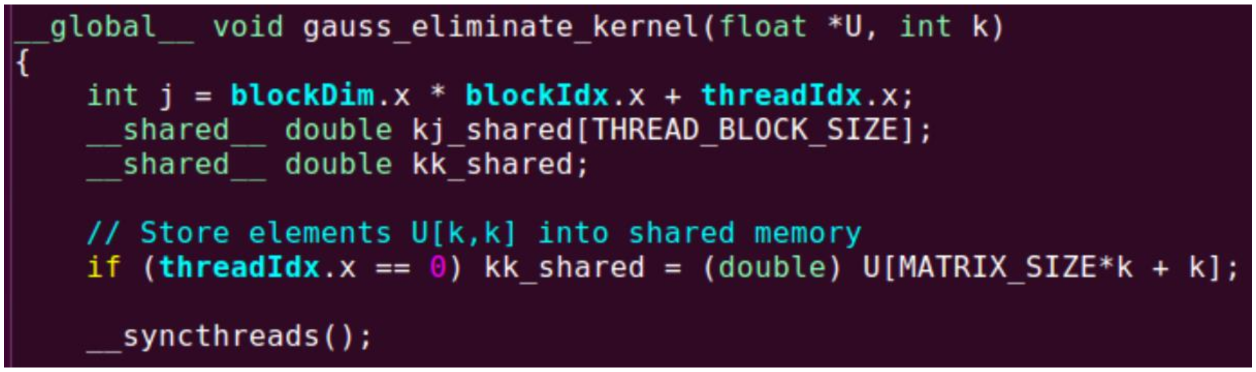
______For the host side function, the first necessary step was to allocate Matrix U onto the GPU, copy the contents of A onto the local Matrix U, and then copy over the local Matrix U's elements into the GPU side functions U Matrix. The thread block size was kept as a predefined macro that we can change in the header function only having threads in the x direction, while the grid sizing was calculated by dividing the number of elements in each row of the Matrix by the thread block size, also being in the x direction only. We then iterated over the k loop that's given in the Gaussian Elimination Algorithm, and call the gpu to separately calculate every k iteration, with a cudaThreadSynchronize() after every loop.



```
void
gauss_eliminate_on_device(const Matrix A, Matrix U){

    int i, j, k;
    Matrix Ud = allocate_matrix_on_gpu(U);

    for (i = 0; i < MATRIX_SIZE; i++) {
        for (j = 0; j < MATRIX_SIZE; j++)
            U.elements[MATRIX_SIZE * i + j] = A.elements[MATRIX_SIZE * i + j];
    }

    copy_matrix_to_device(Ud, U);

    int num_thread_blocks = ceil((float)MATRIX_SIZE/(float)THREAD_BLOCK_SIZE);
    dim3 thread_blocks (THREAD_BLOCK_SIZE, 1, 1);
    dim3 grid (num_thread_blocks, 1, 1);

    struct timeval start, stop;
    gettimeofday(&start, NULL);

    for (k = 0; k < MATRIX_SIZE; k++){
        checkCUDAError("");
        gauss_eliminate_kernel<<< grid, thread_blocks>>>(Ud.elements, k);
        cudaThreadSynchronize();
    }

    gettimeofday(&stop, NULL);

    printf("Parallel Time = %fs. \n", (float)(stop.tv_sec - start.tv_sec + (stop.tv_usec -
start.tv_usec)/(float)1000000));
    copy_matrix_from_device(U, Ud);

    FreeDeviceMatrix(&Ud);
}
```

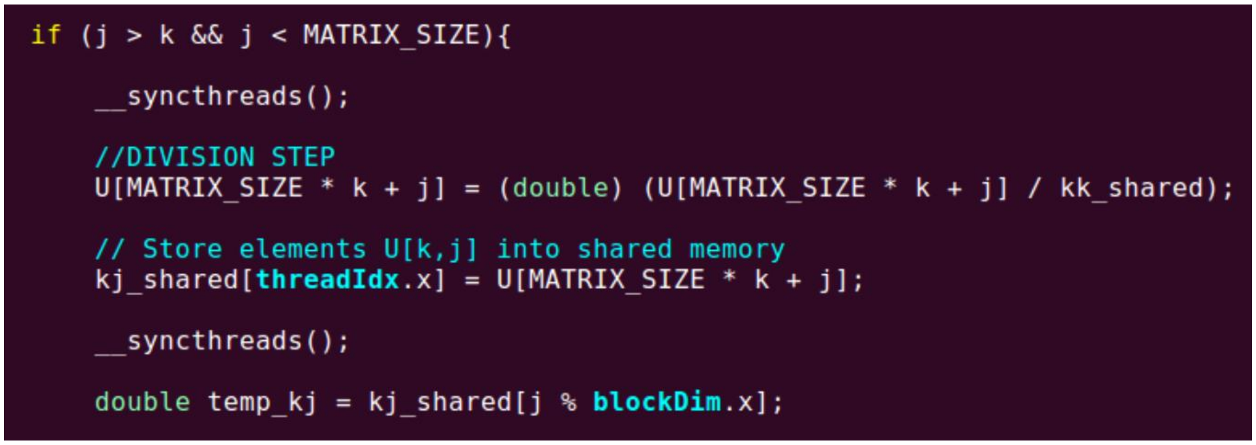
<u>Device Side Function:</u>

The CUDA kernel accepts pointer to the matrix elements and k, which corresponds to the index of the outermost for loop from the pseudocode algorithm description. Each thread stores its location in the matrix by calculating the variable **j**. As described in the "Parallelization approach" section, element A[k,k] and part of the row A[k, :] are stored in shared memory to speed up the program. Only the first thread from each block stores A[k,k] into shared memory, as shown below.



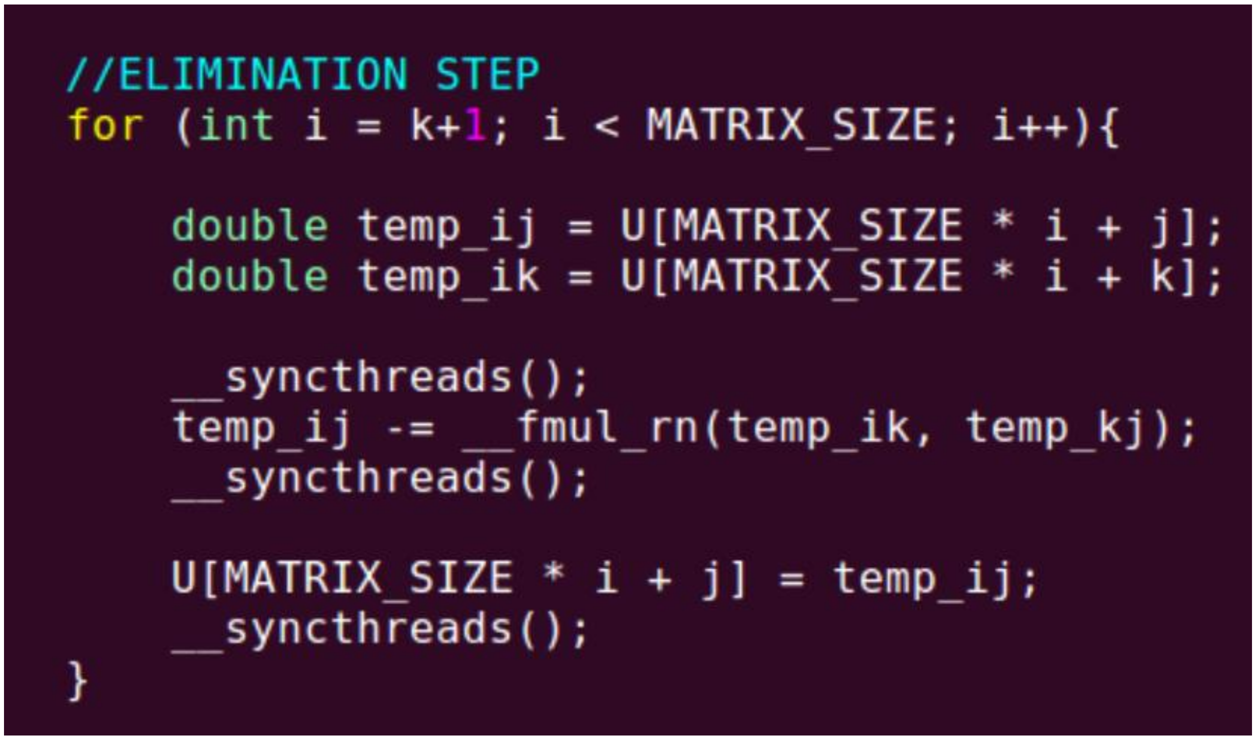
```
__global__ void gauss_eliminate_kernel(float *U, int k)
{
    int j = blockDim.x * blockIdx.x + threadIdx.x;
    __shared__ double kj_shared[THREAD_BLOCK_SIZE];
    __shared__ double kk_shared;

    // Store elements U[k,k] into shared memory
    if (threadIdx.x == 0) kk_shared = (double) U[MATRIX_SIZE*k + k];

    __syncthreads();
```

Because the Gaussian elimination algorithm progressively reduces the area of the matrix within which the calculations are done, it is necessary to check if any given thread need to perform any work for each k iteration. It is idle if its index is less than **k** or if it is out of bounds (the latter is never the case with our selection of thread block size, but it is added anyways for robustness). Division step is performed exactly as in the pseudocode description. Right before the elimination section begins, a part of the kth row is stored into shared array **kj_shared** that will be used repeatedly in the loop for the elimination step.



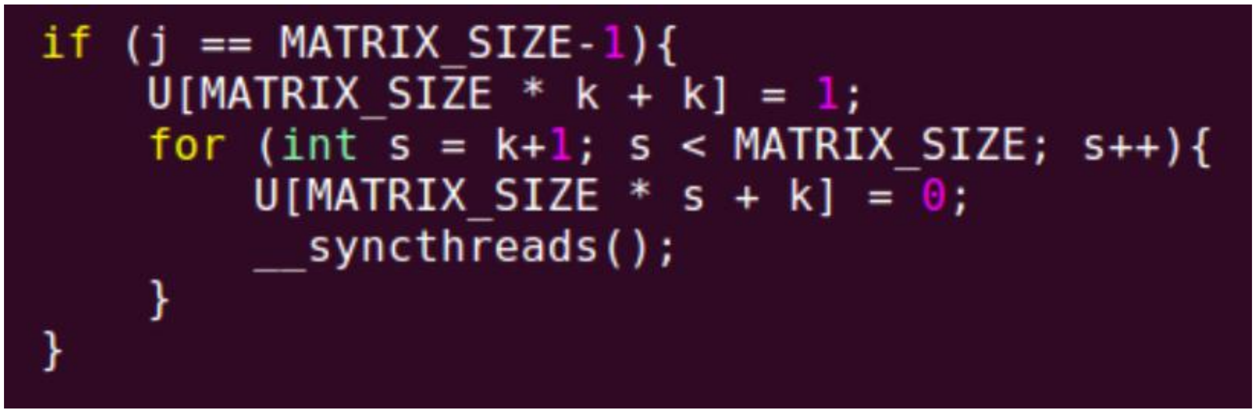
```
if (j > k && j < MATRIX_SIZE){

    __syncthreads();

    //DIVISION STEP
    U[MATRIX_SIZE * k + j] = (double) (U[MATRIX_SIZE * k + j] / kk_shared);

    // Store elements U[k,j] into shared memory
    kj_shared[threadIdx.x] = U[MATRIX_SIZE * k + j];

    __syncthreads();

    double temp_kj = kj_shared[j % blockDim.x];
```

The screenshot below shows the code for elimination step. Each thread that gets to this step iterates from row k+1 to the last row. To ensure accuracy of computation, the float values from the global memory are stored into **double** precision registers temp_ij and temp_ik. The value of A[i,j] is then updated by performing the operation from line 11 of pseudocode. To ensure precision of the FP multiplication, we used the **fmul** function, and then stored the multiplication result back into the global memory matrix.



```
//ELIMINATION STEP
for (int i = k+1; i < MATRIX_SIZE; i++){

    double temp_ij = U[MATRIX_SIZE * i + j];
    double temp_ik = U[MATRIX_SIZE * i + k];

    __syncthreads();
    temp_ij -= __fmul_rn(temp_ik, temp_kj);
    __syncthreads();

    U[MATRIX_SIZE * i + j] = temp_ij;
    __syncthreads();
}
```

The zeroing out of lower triangle elements and making the Matrix diagonal elements' all 1's, was performed as the final step to make sure there was no contention with other threads during the division and elimination operations. The if condition checks that the last thread in a thread block, given by the variable j performs the zeroing of lower triangle elements serially, while also turning all diagonal elements in the matrix to 1.



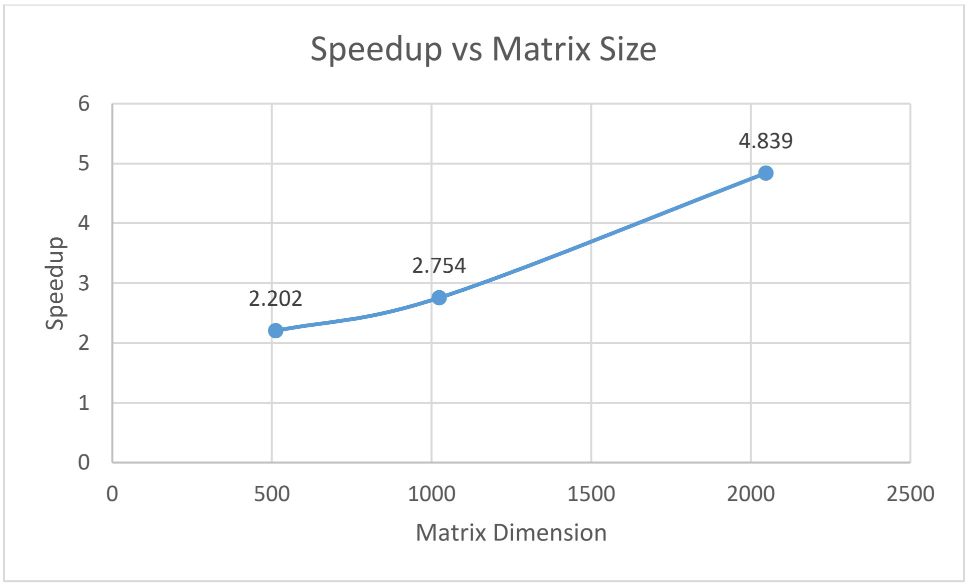
```
if (j == MATRIX_SIZE-1){
    U[MATRIX_SIZE * k + k] = 1;
    for (int s = k+1; s < MATRIX_SIZE; s++){
        U[MATRIX_SIZE * s + k] = 0;
        __syncthreads();
    }
}
```

Results:

The following results were obtained when we ran the algorithm for three different matrix sizes.

Table 1: Timing results

Matrix Size	Time Serial (s)	Time CUDA (s)	Speedup
512x512	0.09778	0.0444	2.202
1024x1024	0.5354	0.1944	2.754
2048x2048	3.8541	0.7964	4.839



The results shown above indicate that our implementation achieves speedup over serial implementation. In addition, the results illustrate Gustafson's law: the speedups become greater for larger matrices.