DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

ECEC-413

Jacobi Iterative Method with CUDA

Chris Kasper

Prof. Naga Kandasamy

DATE: 06/5/19

GPU Implementation (using CUDA)

Initialize/free code is as follows:

```
/* Perform the Jacobi calculation on the GPU. */
void
compute_on_device (const matrix_t A, matrix_t gpu_naive_sol_x, matrix_t gpu_opt_sol_x, const matrix_t B)
{
    int done = 0;
    int num_iter = 0;

    double ssd;
    double *d_ssd = NULL;

    double mse;
    int i;
    struct timeval start, stop;

    matrix_t new_x_naive = allocate_matrix_on_host (MATRIX_SIZE, 1, 0);
    matrix_t new_x_opt = allocate_matrix_on_host (MATRIX_SIZE, 1, 0);

    /* Initialize gpu x solutions */
    for (i = 0; i < A.num_rows; i++) {
        float e = B.elements[i];
        gpu_naive_sol_x.elements[i] = e;
        gpu_opt_sol_x.elements[i] = e;
    }


    /* Allocate space on GPU for the matricies */
    matrix_t d_A = allocate_matrix_on_device(A);
    check_CUDA_error("Allocating matrix A");
    matrix_t d_naive_sol_x = allocate_matrix_on_device(gpu_naive_sol_x);
    check_CUDA_error("Allocating matrix naive_sol_x");
    matrix_t d_opt_sol_x = allocate_matrix_on_device(gpu_opt_sol_x);
    check_CUDA_error("Allocating matrix opt_sol_x");
    matrix_t d_B = allocate_matrix_on_device(B);
    check_CUDA_error("Allocating matrix B");
    matrix_t d_new_x_naive = allocate_matrix_on_device(new_x_naive);
    check_CUDA_error("Allocating new_x_naive");
    matrix_t d_new_x_opt = allocate_matrix_on_device(new_x_opt);

    /* Copy over matricies A, B, and x solutions to GPU*/
    copy_matrix_to_device(d_A, A);
    check_CUDA_error("Copying matrix A to device");
    copy_matrix_to_device(d_B, B);
    check_CUDA_error("Copying matrix B to device");
    copy_matrix_to_device(d_naive_sol_x, gpu_naive_sol_x);
    check_CUDA_error("Copying matrix naive_sol_x to device");
    copy_matrix_to_device(d_opt_sol_x, gpu_opt_sol_x);
    check_CUDA_error("Copying matrix opt_sol_x to device");

    /* Allocate space for the ssd on the GPU */
    cudaMalloc ((void**) &d_ssd, sizeof (double));
```

```
/* Allocate space for the lock on the GPU and initialize it. */
int *mutex_on_device = NULL;
cudaMalloc ((void **) &mutex_on_device, sizeof (int));
cudaMemset (mutex_on_device, 0, sizeof (int));

/* Kernel setup */
dim3 thread_block (1, THREAD_BLOCK_SIZE, 1);
dim3 grid (1, (A.num_rows + THREAD_BLOCK_SIZE - 1)/ THREAD_BLOCK_SIZE);
printf ("Performing Jacobi naive solution\n");
gettimeofday(&start, NULL);

while (!done) {
    cudaMemset (d_ssd, 0.0, sizeof (double));

    jacobi_iteration_kernel_naive<<<grid, thread_block>>>(d_A, d_naive_sol_x, d_new_x_naive, d_B, mutex_on_devic
    check_CUDA_error("KERNEL FAILURE: jacobi_iteration_kernel_naive\n");
    cudaDeviceSynchronize ();

    jacobi_update_x<<<grid,thread_block>>>(d_naive_sol_x, d_new_x_naive);
    check_CUDA_error("KERNEL FAILURE: jacobi_update_x");
    cudaDeviceSynchronize();

    // Check for convergence
    cudaMemcpy (&ssd, d_ssd, sizeof (double), cudaMemcpyDeviceToHost);
    num_iter++;
    mse = sqrt(ssd);

    if (mse <= THRESHOLD) {
        done = 1;
        printf ("\nConvergence achieved after %d iterations \n", num_iter);
    }

    //printf ("Iteration: %d. MSE = %f\n", num_iter, mse);
}

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

/* Modify kernel setup for optimized kernel */
thread_block.x = thread_block.y = TILE_SIZE;
grid.x = 1;
grid.y = (gpu_opt_sol_x.num_rows + TILE_SIZE - 1)/TILE_SIZE;

printf("\nPerforming Jacobi optimized solution\n");
gettimeofday(&start, NULL);
done = 0;
num_iter = 0;
```

```
    while (!done) {
        cudaMemset (d_ssd, 0.0, sizeof (double));

        jacobi_iteration_kernel_optimized<<<grid, thread_block>>>(d_A, d_opt_sol_x, d_new_x_opt, d_B, mutex_on_devic
        check_CUDA_error("KERNEL FAILURE: jacobi_iteration_kernel_optimized\n");
        cudaDeviceSynchronize ();

        jacobi_update_x<<<grid,thread_block>>>(d_opt_sol_x, d_new_x_opt);
        check_CUDA_error("KERNEL FAILURE: jacobi_update_x");
        cudaDeviceSynchronize();

        // Check for convergence
        cudaMemcpy (&ssd, d_ssd, sizeof (double), cudaMemcpyDeviceToHost);
        num_iter++;
        mse = sqrt(ssd);

        if (mse <= THRESHOLD) {
            done = 1;
            printf ("\nConvergence achieved after %d iterations \n", num_iter);
        }

        //printf ("Iteration: %d. MSE = %f\n", num_iter, mse);
    }

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

    /* Copy back solutions from GPU */
    copy_matrix_from_device(gpu_naive_sol_x, d_naive_sol_x);
    check_CUDA_error("Copying matrix d_naive_sol_x from device");
    copy_matrix_from_device(gpu_opt_sol_x, d_opt_sol_x);
    check_CUDA_error("Copying matrix d_opt_sol_x from device");

    /* Free memory on GPU */
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_naive_sol_x.elements);
    cudaFree(d_opt_sol_x.elements);
    cudaFree(d_ssd);

    cudaFree(mutex_on_device);
    cudaFree(d_new_x_naive.elements);
    cudaFree(d_new_x_opt.elements);

    free (new_x_naive.elements);
    free (new_x_opt.elements);

    return;
}
```

One key difference between the CUDA kernels and the CPU compute_gold(), is that there is an additional kernel function to update the x solution values. This is mostly due to the fact that grid-level thread syncing doesn't exist.

For the naïve kernel, the thread block and grid is based off the #define constant THREAD_BLOCK_SIZE and the number of rows of matrix A. For the optimized approach, which uses tiling, the configuration is based off the #define constant TILE_SIZE, and the number of rows in matrix A.

Checking for convergence is also done on the host-side, as the SSD value is transmitted back to the host from the device.

Here is the kernel code:

```
/* Jacobi iteration using global memory */
__global__ void jacobi_iteration_kernel_naive (const matrix_t A, const matrix_t x, matrix_t x_update, const matrix_t B, int* mutex, double* ssd)
{
    __shared__ double ssd_per_thread[THREAD_BLOCK_SIZE];
    unsigned int i, j;
    unsigned int num_rows = A.num_rows;
    unsigned int num_cols = A.num_columns;
    float new_x;
    double thisSSD = 0.0;

    int threadY = threadIdx.y;
    int blockY = blockIdx.y;

    int row = blockDim.y * blockY + threadY;

    if (row < num_rows) {
        /* Initialize Jacobi sum, begin computation */
        double sum = -A.elements[row * num_cols + row] * x.elements[row];

        for (j = 0; j < num_cols; j++) {
            sum += (double) A.elements[row * num_cols + j] * x.elements[j];
        }

        /* Find new unknown value */
        new_x = (B.elements[row] - sum) / A.elements[row * num_cols + row];

        __syncthreads();

        thisSSD = (double) (new_x - x.elements[row]) * (new_x - x.elements[row]);
        x_update.elements[row] = new_x;

        ssd_per_thread[threadY] = thisSSD;
        __syncthreads();

        /* SSD reduction */
        i = blockDim.y/2;
        while (i != 0) {
            if (threadY < i)
                ssd_per_thread[threadY] += ssd_per_thread[threadY + i];
            __syncthreads();
            i /= 2;
        }

        if (threadY == 0) {
            lock(mutex);
            *ssd += ssd_per_thread[0];
            unlock(mutex);
        }

    }

    return;
}
```

```
/* Jacobit iteration using global and shared memory, including coalecesed memory calls */
__global__ void jacobi_iteration_kernel_optimized (const matrix_t A, const matrix_t x, matrix_t x_update, const matrix_t B, int* mutex, double* ssd
{
    __shared__ float aTile[TILE_SIZE][TILE_SIZE];
    __shared__ float xTile[TILE_SIZE];
    __shared__ double ssd_per_thread[TILE_SIZE];

    unsigned int i, k;
    unsigned int num_rows = A.num_rows;
    unsigned int num_cols = A.num_columns;
    float new_x;
    double sum = 0.0;
    double thisSSD = 0.0;

    int threadX = threadIdx.x;
    int threadY = threadIdx.y;
    int blockY = blockIdx.y;
    int row = blockDim.y * blockY + threadY;

    if (row < num_rows) {
        for (i = 0; i < num_cols; i+= TILE_SIZE) {
            /* Bring TILE_SIZE elements per row of A into shared memory */
            aTile[threadY][threadX] = A.elements[row * num_cols + i + threadX];

            /* Bring TILE_SIZE elements of x and b into shared memory */
            if (threadY == 0)
                xTile[threadX] = x.elements[i + threadX];

            __syncthreads();

            /* Compute partial Jacobi sum for the current tile */
            if (threadX == 0) {
                for (k = 0; k < TILE_SIZE; k+=1)
                    sum += (double) aTile[threadY][k] * xTile[k];
            }

            __syncthreads();
        }

        if (threadX == 0) {
            float aDiag = A.elements[row * num_cols + row];
            float xDiag = x.elements[row];
            float bDiag = B.elements[row];

            sum += -aDiag * xDiag;
            new_x = (bDiag - sum) / aDiag;

            thisSSD = (double) (new_x - xDiag) * (new_x - xDiag);
            x_update.elements[row] = new_x;

            ssd_per_thread[threadY] = thisSSD;
            __syncthreads();

            i = blockDim.y/2;
            while (i != 0) {
                if (threadY < i)
                    ssd_per_thread[threadY] += ssd_per_thread[threadY + i];
                __syncthreads();
                i /= 2;
            }

            if (threadY == 0) {
                lock(mutex);
                *ssd += ssd_per_thread[0];
                unlock(mutex);
            }
        }
    }

    return;
}
```

```
/* Jacobi iteration updating x */
__global__ void jacobi_update_x (matrix_t sol_x, const matrix_t new_x)
{
    unsigned int num_rows = sol_x.num_rows;

    int threadY = threadIdx.y;
    int threadX = threadIdx.x;
    int blockY = blockIdx.y;
    int row = blockDim.y * blockY + threadY;

    if ((row < num_rows) && (threadX == 0)) {
        sol_x.elements[row] = new_x.elements[row];
    }

    return;
}
```

For the naïve approach, each thread is responsible for a row in the matrix, which are also in a thread block with other rows. To compute the SSD value, reduction is used, and then by using a mutex lock, is added to the SSD pointer.

For the optimized approach, more shared memory is being used, as tiling is implemented. Once the tiles are populated, only half the threads in the block do the summation calculation. This is repeated as the tiles go across their respective rows. Once this part is finished, the new x values are found and the SSD computation occurs, just like the naïve function.

The last kernel function is used to update the x solution values from the other kernel calls.

The speedup table below shows the times of the kernel functions, with some different kernel configurations to show sensitivity. I'm not sure why my naïve function is faster than the optimized function for larger matrix sizes. It could be due to the amount of synchronizing that is occurring with those large matrices.

Speedup

Table 1: Speedup with Jacobi Iterative Method on xunil-05

Matrix Size	CPU	GPU naïve (Thread Block Size = 128)	GPU optimized (Tile Size = 16)	GPU naïve (Thread Block Size = 256)	GPU optimized (Tile Size = 8)
512 x 512	3.66 s	1.10 s	1.00 s	1.38 s	1.16 s
1024 x 1024	28.38 s	2.82 s	7.74 s	3.89 s	4.91 s
2048 x 2048	239.30 s	9.85 s	51.40 s	12.09 s	27.27 s

Note: xunil-05 has a Nvidia 1080 GTX GPU