Tam Phi
Hanh Do Phung

Assignment 5: Jacobi parallelization with CUDA

I. Introduction

Jacobi solver is a method of solving a system of n-linear equations of the form Ax = b in which x is a (n x 1) vector of all x_1 to x_n solutions and A is an n x n sized matrix representing the coefficients of variables x_1 to x_n.

II. Design and Implementation

There are 2 steps in implementation.

1) Exec() function:

-Allocate memory on GPU for the output matrix U, call the kernel function to perform parallelization

Pseudo code{

/*Allocate memory on device */

/* Copy matrix to device */

/* Initialize mutex lock */

}

Naïve version

```
    while (! done){
       jacobi_iteration_kernel_naive
          <<< grid,thread_block >>>
          (device_A.elements, device_B.elements, curr_X.elements, prev_X.elements, input_elements, ssd_device);

        cudaDeviceSynchronize();
        cudaMemcpy(ssd_host, ssd_device, elements * sizeof(double), cudaMemcpyDeviceToHost);

        double ssd = 0;
        int i = 0;
        for (i = 0; i < elements; i++)
            ssd += ssd_host[i];
        // ping pong
        float *X_temp = prev_X.elements;
        prev_X.elements = curr_X.elements;
        curr_X.elements = X_temp;

        num_iter++;

        double mse = sqrt (ssd);
        if (mse <= THRESHOLD){
           done = 1;
        }
    }
```

Optimized version

Tam Phi
Hanh Do Phung

```
    while(!done)
    {
        (ping_pong)
          ?
          (jacobi_iteration_kernel_optimized<<< grid, threads >>>(device_A, opt, device_B, out_X, ssd_device, mutex_on_device))
          : \
          (jacobi_iteration_kernel_optimized<<< grid, threads >>>(device_A, out_X, device_B, opt, ssd_device, mutex_on_device));

        cudaDeviceSynchronize();

        num_iter++;

        ping_pong = !ping_pong;
        cudaMemcpy(&ssd, ssd_device, sizeof(double), cudaMemcpyDeviceToHost);
        mse = sqrt (ssd);
        if (mse <= THRESHOLD)
            done = 1;
    }
```

2) GPU kernel functions:

- **naive_kernel():** A function that performs division of current row by pivot element

```
__global__ void jacobi_iteration_kernel_naive(float *d_A, float * d_B, float *curr_X, float *prev_X, int *input_elements, double *ssd)
{

    int tid = threadIdx.y;
    double sum = 0;
    int num_elements = *input_elements;

    int i = (blockIdx.y * THREAD_BLOCK_SIZE) + tid;
    for( int j = 0; j < num_elements; j++){
        if (i != j)
            sum += d_A[i * num_elements + j] * curr_X[j];
    }

    prev_X[i] = (d_B[i] - sum)/d_A[i * num_elements + i];

    ssd[i] = (curr_X[i] - prev_X[i]) * (curr_X[i] - prev_X[i]);

    return;
}
```

- **opt_kernel():** A function that performs reduction of under rows

```
__global__ void jacobi_iteration_kernel_optimized(float *d_A, float *opt, float *d_B, float *out_X, double *ssd, int *mutex)
{
    __shared__ double ssd_device[THREAD_BLOCK_SIZE];

    unsigned int tid = threadIdx.x;
    unsigned int blkid = blockIdx.x * blockDim.x + threadIdx.x;

    //reset ssd = 0
    if (blkid == 0)
        *ssd = 0.0;

    // begin iteration
    double sum = -d_A[blkid * MATRIX_SIZE + blkid] * opt[blkid];
    for (int j = 0; j < MATRIX_SIZE; j++) {
        sum += d_A[blkid + MATRIX_SIZE * j] * opt[j];
    }
    out_X[blkid] = (d_B[blkid] - sum)/d_A[blkid * MATRIX_SIZE + blkid];

    if (blkid < MATRIX_SIZE)
        ssd_device[tid] = (out_X[blkid] - opt[blkid]) * (out_X[blkid] - opt[blkid]);
    else
        ssd_device[tid] = 0.0;

    __syncthreads();

    for (unsigned int stride = blockDim.x >> 1; stride > 0; stride = stride >> 1) {
       if(tid < stride)
          ssd_device[tid] += ssd_device[tid + stride];
       __syncthreads();
    }

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

    return;
}
```

Tam Phi
Hanh Do Phung

The optimized version calculate single value of ssd on GPU while the naïve version copy the ssd_array back to CPU, then calculate the ssd on host side.

Result and discussion

All accuracy test was passed for a variety of matrix size as suggested in the figure below.

```
Generating 1024 x 1024 system

Performing Jacobi iteration on the CPU

Convergence achieved after 16797 iterations
Execution time = 28.953085s
Average diff between LHS and RHS: 0.001917

Performing naive Jacobi iteration on device
Execution time = 10.669594s
Done in 16797 iterations
Average diff between LHS and RHS: 0.001917

Performing optimized Jacobi iteration on device
Execution time = 9.194625s
Done in 16797 iterations
Average diff between LHS and RHS: 0.001917
```

<u>Execution time</u>

Matrix width	Num threads	gold_time	naïve_time	opt_time
512	128	3.656533	4.199220	4.108942
512	256	3.673595	4.445559	4.141872
1024	128	28.30203	9.337974	8.826369
1024	256	28.953085	10.669594	9.194625
2048	128	258.089325	24.00955	20.450293
2048	256	236.729752	26.709866	20.975603

Conclusion

As the data moving step between the CPU and GPU is not included in the execution time measurements, the trend of more speed up is observed as the matrix size increases. This is because of the one-time overhead of cudaMalloc() and cudaMemcopy() executed is more significant overall compared to the execution time of the algorithm itself on the CPU/GPU. However, size of the thread block does not affect speed up for matrix of size 2048x2048 or smaller.