Parallel Computer Architecture
Iterative Jacobi Solver

Prof. Naga Kandasamy
ECE Department, Drexel University

May 14, 2019

This assignment, worth fifteen points, is due May 28, 2019, by 11:59 pm via BBLearn. You may work on this problem in a team of up to two people. One submission per group will suffice. You may discuss high-level approaches to solve the problem with other students, Shihao, or me, but must submit original work.

Consider the following system of linear equations $Ax = b$ with n equations and n unknowns:

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

where the unknowns are $x_0, x_1, x_2, \ldots, x_{n-1}$. We have seen from a previous assignment that Gaussian elimination is one way to solve for these unknowns. Another method is by iteration. In the above system, the i^{th} equation

$$a_{i,0}x_0 + a_{i,1}x_1 \cdots a_{i,i}x_i \cdots + a_{i,n-1}x_{n-1} = b_i$$

can be rearranged as

$$x_i = \frac{1}{a_{i,i}} \left(b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_1 \cdots a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \cdots + a_{i,n-1}x_{n-1}) \right),$$

or as

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j}x_j \right) \tag{1}$$

This equation gives x_i in terms of the other unknowns and can be used as an iteration formula for each of the unknowns to obtain better approximations. The iterative method used in the reference code provided to you is the *Jacobi iteration*. It can be shown that the Jacobi method will converge if the diagonal values of a have an absolute value greater than the sum of the absolute values of the

other a's on the row; that is, the array of a's is *diagonally dominant*. In other words, convergence is guaranteed if

$$\sum_{j \neq i} |a_{i,j}| \leq |a_{i,i}|.$$

The matrix A generated by the program satisfies the above condition. Note however, that this condition is a sufficient but not a necessary condition; that is, the method may converge even if the array is not diagonally dominant. The iteration formula in (1) is numerically unstable, however, in that it will not work if any of the diagonal elements are zero because it would require dividing by zero—but you do not have to worry about this issue for the assignment. An example of how the iterative method works to solve the system of equations is illustrated using a small example in the file called `jacobi_example.pdf`.

Given arrays `a[][]` and `b[]` holding the constants in the equation, `x[]` holding the unknowns, and a user-defined tolerance for convergence, the sequential code that implements the Jacobi iteration can be developed as follows.

```
for (i = 0; i < n; i++)          /* Initialize unknowns. */
    x[i] = b[i];
done = 0;
while (!done) {
    for (i = 0; i < n; i++){ /* Loop iterates over each unknown. */
        sum = 0;
        for (j = 0; j < n; j++) { /* Implement Equation 1. */
            if (i != j)
                sum += a[i][j] * x[j];
        }
        new_x[i] = (b[i] - sum)/a[i][i];
    }
    /* Update the unknown values and test for convergence. */
    ssd = 0;
    for (i = 0; i < n; i++) {
        ssd += (x[i] - new_x[i])^2;
        x[i] = new_x[i];
    }
    if (sqrt(ssd) < TOLERANCE)
        done = 1;
} /* End while. */
```

The convergence criteria tests the square root of the sum of the squared differences (SSD) of the `x` values from two consecutive iterations—the current iteration and the one before—against a user-defined error tolerance as

$$\sqrt{\sum_{i=0}^{n-1} (x_i^k - x_i^{k-1})^2} \leq \text{error tolerance},$$

where k is the iteration number.

The reference code given to you takes no input arguments. The solution provided by the GPU is compared to that generated by the CPU by printing out the relevant statistics. The configurable parameters such as the dimensions of the matrices, the error tolerance, thread block sizes, and number of thread blocks can be found within the .h file. *Do not change the error tolerance value.*

Answer the following questions:

- **(5 points)** Complete the functionality of the equation solver on the GPU by editing the `compute_on_device()` and `jacobi_iteration_kernel_naive()` functions. This naive implementation can perform all operations on matrices A, x, and b while maintaining them in *global memory* and accesses to global memory do not have to be coalesced. In other words, your objective is to write a kernel that works correctly without considering performance.
- **(10 points)** Using the naive implementation as a starting point, optimize the performance of your GPU code via efficient use of the GPU memory hierarchy, including *shared memory*. Ensure that your accesses to the matrix elements in global memory are *coalesced*. Complete this functionality within and `jacobi_iteration_kernel_optimized()`. For full credit, all major operations including updating the unknown values and reducing the SSD to a single value must be performed on the GPU.

Note: Referring to the code snippet from earlier, the outer `while` loop (line 4) that tests for convergence *must* run on the host whereas each unknown x_i can be updated in parallel on the GPU. That is, you will parallelize the `for` loop in lines 5 through 12. Calculating the SSD value (lines 15–18) can also be parallelized on the GPU using the reduction technique discussed in class. The SSD value is then transmitted back to the host to check for convergence, and the process repeats until convergence is achieved.

You may add additional kernels as well as host-side helper and timing functions as needed to your code. You may also have to use double precision operations on the GPU to minimize any mismatch to the reference result.

Upload all files needed to run your code as a single zip file. Submit a short report describing: (1) the design of your kernels using code or pseudocode to clarify the discussion; (2) the speedup achieved by both kernel implementations over the serial version for matrix sizes of 512×512, 1024×1024, and 2048×2048, including the overhead due to CPU/GPU communication; and (3) sensitivity of your kernels to thread-block size in terms of the execution time.