

Iterative Jacobi Solver

Prof. Naga Kandasamy
ECE Department, Drexel University

This assignment, worth twenty points, is due March 20, 2025, 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. Please submit original work.

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

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

where the unknowns are $x_0, x_1, x_2, \dots, x_{n-1}$. 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

$$x_i = \frac{1}{a_{i,i}} (b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} + \cdots + a_{i,n-1}x_{n-1})),$$

or as

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

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.

```

1  for (i = 0; i < n; i++)           /* Initialize unknowns. */
2      x[i] = b[i];
3  done = 0;
4  while (!done) {
5      for (i = 0; i < n; i++){ /* Loop iterates over each unknown. */
6          sum = 0;
7          for (j = 0; j < n; j++) { /* Implement Equation 1. */
8              if (i != j)
9                  sum += a[i][j] * x[j];
10             }
11             new_x[i] = (b[i] - sum)/a[i][i];
12         }
13         /* Update the unknown values and test for convergence. */
14         ssd = 0;
15         for (i = 0; i < n; i++) {
16             ssd += (x[i] - new_x[i])^2;
17             x[i] = new_x[i];
18         }
19         if (sqrt(ssd) < TOLERANCE)
20             done = 1;
21     } /* 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:

- **(10 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)** Optimize the performance of your GPU code by ensuring that accesses to matrix elements in global memory are *coalesced*. To achieve this: (1) change the layout of matrix A to column-major form prior to transferring A to the GPU; and (2) write kernel code within `jacobi_iteration_kernel_optimized()` which operates on this column-major form. 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.

Implementation tip: 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. Recall that global memory is persistent between kernel calls. So, values of x that were updated by the kernel during a previous iteration will be available in memory for the next iteration. Use ping-pong buffers in GPU memory to store values generated between two iterations. The following pseudo-code can be a starting point for your code development.

```
// Host-side code

1. Allocate device memory for A, b, x, SSD, and copy over data.
   Vector x is initialized using values from b prior to transfer.
   Allocate ping-pong buffers on the device to hold values of x
   between iterations.

2. Set up execution grid on device.

3. done = 0

4. while (!done)
    a. cudaMemset() SSD to zero on device.
    b. Launch kernel. For fastest implementation, a single launch can
       update the x vector and calculate the SSD value. Use parallel
       reduction to calculate SSD on device.
    c. Copy SSD back to host from device.
    d. If SSD has converged, done = 1.
    e. Flip pointers for ping-pong buffers on the device.

5. Copy final values of x vector from device back to host.
```

You may add kernels as well as host-side helper functions as needed to your code. Use double precision operations on the GPU to minimize any mismatch to the reference result.

Upload all source files needed to build and run your code as a single zip file (after running `make clean` to remove any pre-built objects files and executable). 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. The matrix size is defined in *jacobi_iteration.h*.