GPU-Accelerated Jacobi-like Algorithm for Eigendecomposition of General Complex Matrices

Basileal Imana and Peter Yoon

Department of Computer Science, Trinity College, Hartford, CT

Abstract

Jacobi-like methods, though introduced in 1846, were not popular as they were computationally expensive for sequential computing. With the advent of parallel computing, however, it has become feasible to efficiently implement such algorithms in parallel. In addition, the Jacobi method has been shown to compute very small eigenvalues with high accuracy compared to the conventional methods. In this research, we present a novel parallel implementation of Jacobi method for eigendecomposition of general complex matrices on the GPU. Our preliminary results show a significant improvement over those on the CPU, running up to 94 times faster for general dense complex matrices of moderate size.

Problem

Let $\boldsymbol{A}$ be an nxn matrix general complex matrix whose eigenvalues are to be computed. Jacobi-like methods perform a series of similarity transformations

$$\boldsymbol{A}_{k+1} = \boldsymbol{M}_k^{-1} \boldsymbol{A}_k \boldsymbol{M}_k \qquad K = 0,1,2,\ldots$$

such that the matrix $\boldsymbol{A}$ reduces towards a triangular or diagonal matrix from which eigenvalues are easy to compute. The resulting matrix is similar to $\boldsymbol{A}$ and , thus, will have the same eigenvalues as $\boldsymbol{A}$.

All transformation matrices are accumulated into a matrix $\boldsymbol{E}$ whose column contains the corresponding eigenvectors of $\boldsymbol{A}$.

Parallel Jacobi

- The Jacobi method for eigenvalues proposed by Shroff[1] is an iterative algorithm that repeatedly performs similarity transformations until the matrix becomes almost diagonal.
- Each transformation matrix $\boldsymbol{M}$ is identical to the identity matrix except for elements $\boldsymbol{M}_{pp}$, $\boldsymbol{M}_{pq}$, $\boldsymbol{M}_{qp}$ and $\boldsymbol{M}_{qq}$, where (p, q) is called the pivot of transformation.
- Each transformation is one of the following three types:

  Unitary transformation is used to annihilate the (q, p) element of $\boldsymbol{A}$. It has the following structure in positions that differs from the identity matrix

  $$\begin{pmatrix} \cos x & -e^{i\theta} \sin x \\ e^{-i\theta} \sin x & \cos x \end{pmatrix}$$

  Shear transformation is used to reduce the departure of matrix $\boldsymbol{A}$ from normality. It has the following structure

  $$\begin{pmatrix} \cosh y & -i\, e^{i\alpha} \sinh y \\ e^{-i\alpha} \sinh y & \cosh y \end{pmatrix}$$

  Diagonal transformation reduces the norm of the off-diagonal elements of $\boldsymbol{A}$. It is the same as the identity matrix except for the jth diagonal element which is equal to t_j.
- The transformation parameters $x, y, \theta, \alpha, t_j$ are chosen so that the transformation will annihilate certain elements of the matrix $\boldsymbol{A}$.
- At each iteration, a sweep is performed. At the end of a sweep, every pair $(p, q) \in \{1 \le p < q \le n\}$ has been covered by a unitary and shear transformation, and every row/column has been covered by a diagonal transformation.
- All transformation are accumulated into a matrix $\boldsymbol{E}$, whose columns will store eigenvectors of matrix $\boldsymbol{A}$.
- The algorithm is highly parallelizable as up to $n/2$ transformations can be performed at the same time as long as we ensure each (p, q) pair is unique for each processing unit.
- In our GPU implementation, we used chess tournament algorithm for choosing parallel ordering of pairs. In such ordering, there are $n-1$ matching sets such that each player gets matched against every other player exactly once [2].
- After one matching set is completed, the first player stands still and every other player moves one position in clockwise direction.

Figure 1: One step of Chess Tournament ordering

- In such way a total of $(n-1) \cdot n/2$ shear and unitary transformations and n diagonal transformations are formed during each sweep.
- Jacobi method proceeds by iteratively multiplying matrix A from the left and right by the transformation matrix.
- If $\boldsymbol{M}$ is a transformation matrix with pivot (p, q), $\boldsymbol{M}^{-1} \cdot \boldsymbol{A}$ would update the pth and qth rows of matrix $\boldsymbol{A}$ and $\boldsymbol{A} \cdot \boldsymbol{M}$ would update the pth and qth columns of $\boldsymbol{A}$.
- Since the two matrix multiplications can not occur in parallel, we launch a kernel to perform the first multiplication, wait for all processing units to finish updating the rows of $\boldsymbol{A}$ (host side synchronization) and then perform the second multiplication with another kernel.

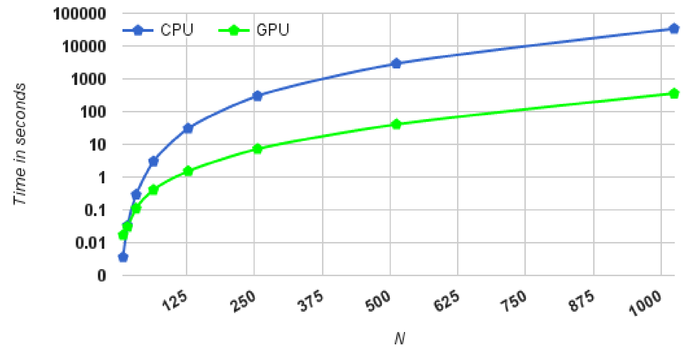
```
Input: An nxn matrix A
 /* Let ε be value of machine precision , L_k be strictly lower triangular part of A */
WHILE norm(L_k) < (n^2/2) · ε DO /* stopping condition */
  FOR EACH ordered pair matching set DO
    calculate_shear_params<<<1,n/2>>>(A);
    shear_kernel_1<<<n/2, n>>>(A, X); /* X used to store intermediate result */
    cudaDeviceSynchronize();
    shear_kernel_2<<<n/2, n>>>(A, X, E) ;/* E used to accumulate all
    transformations*/

    calculate_unitary_params<<<1,n/2>>>(A);
    unitary_kernel_1<<<n/2, n>>>(A, X);
    cudaDeviceSynchronize();
    unitary_kernel_2<<<n/2, n>>>(A, X, E);
  ENDFOR
  calculate_diagonal_params<<<1,n>>>(A)
  diagonal_kernel_1<<<n, n>>>(A, X)
  cudaDeviceSynchronize();
  diagonal_kernel_2<<<n/2, n>>>(A, X, E)
ENDWHILE
Output: An nxn matrix A with eigenvalues on the diagonal and an nxn matrix E whose
columns are corresponding eigenvectors
```

- For the shear and unitary transformation, we launch two GPU kernels with $n/2$ blocks (one processing unit per pivot) and n threads per block (one thread for each vector element).
- For the diagonal transformation, we launch two GPU kernels with n blocks(one processing unit per row/column and n thread per block (one thread per vector element).

Performance

Figure 2: Performance Comparison – CPU vs GPU



- Specification: Intel® Xeon® E5-2620 CPU and NVIDIA® Tesla® K20c GPU with 64 GB main memory and 5GB GPU memory. All floating point operations were performed with double-precision.
- As shown in figure above, GPU implementation code runs up to 94 times faster for general dense complex matrix of size 1024.
- The average residual $\|AP - P\Lambda\|_2$ for computed eigenvalues was 10^{-14} which indicates they are good approximations.
- For well-conditioned matrices, our implementation computes eigenvalues and eigenvectors with low error.
- As indicated in [1], the computed eigenvalues becomes less accurate as the matrix $\boldsymbol{A}$ becomes increasingly non-normal.

Discussion and Future Work

- The kernels which calculate transformation parameters for shear and diagonal transformation take up most of the computation time as the calculations involve summing up elements of all rows and columns of $\boldsymbol{A}$.
- Looking into ways to improve these kernels is area of future work. We plan to further investigate into ways of using reduction kernels to remove the current performance bottleneck.

Acknowledgment

This research was supported by:

- CUDA Teaching Center Program, NVIDIA Research
- Faculty Research Committee, Trinity College

References

[1] Gautam Schroff, “A parallel algorithm for the eigenvalues and eigenvectors of a general complex matrix”, *Numerische Mathematik*, 58(1):779–805, 1990.

[2] M. U. Torun, O. Yilmaz, and A. N. Akansu, “Novel GPU implementation of Jacobi algorithm for Karhunen-Loève transform of dense matrices”, *IEEE 46th Annual Conference on Information Sciences and Systems (CISS)*, pp. 1-6, 2012.

[3] P. J. Eberlein, “On the schur decomposition of a matrix for parallel computation", *IEEE Transaction on Computers*, vol. C, no. 2, pp. 167-174, 1987.

[4] J. Demmel and K. Veselić, "Jacobi’s method is more accurate than QR", *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 4, pp. 1204-1245, 1992.