Speeding Up Jacobi EigenValue Algorithm using High Performance Computing

Parallelization using Graphical Processing Units

Shivangi Gajjar
202011023
DA-IICT ML
Gandhinagar, India
202011023@daiict.ac.in

Tarang Ranpara
202011057
DA-IICT ML
Gandhinagar, India
202011057@daiict.ac.in

Mantra Sanathra
202011041
DA-IICT ML
Gandhinagar, India
202011041@daiict.ac.in

Vaidik Patel
202011038
DA-IICT ML
Gandhinagar, India
202011038@daiict.ac.in

***Abstract*—Eigenvalue and Eigenvectors are the basis of many machine learning algorithms like PCA, image augmentation and SVD. If they're not implemented efficiently, it results in more extensive execution time. In this project, we propose the use of High Performance Computing concepts to parallelise the Jacobi algorithm to compute eigenvalues and eigenvectors. We intend to use CUDA for parallelisation and profiling. The purpose of this paper is to deduce whether the HPC method gives any computational advantage over conventional serial implementation, and if yes, what are all possible types of parallelisation that can be applied.**

***Index Terms*—Eigenvalues, Eigenvectors, Hermitian matrix, Jacobi Algorithm, GPU, parallelisation, profiling**

I. Introduction

Eigenvectors are the non-zero vectors of a linear transformation that varies by a scalar factor. This real-valued scaling factor is known as Eigenvalue. Each Eigenvector corresponds to an Eigenvalue. The critical property of these eigenvectors is that they do not change even if the linear transformation of the same matrix is changed. This paper proposes a parallel approach to implement the Jacobi EigenValue Algorithm [1] since it works with a widely used type of matrices in machine learning called real-symmetric matrices(Hermitian). The purpose of this paper is to improve the performance of this algorithm using the concept of parallel computing using GPU. In most of the well known Machine Learning applications used currently, the data sets utilised for train-test purposes are quite massive in size. The most preferred data structure used by these applications for performing data operations is matrices. For improving the computational complexity while working with such massive data, specific techniques of dimensionality reduction are used to reduce the number of variables. Eigenvectors form a basis for numerous dimensionality reduction techniques [2]. Computing Eigenvalues-Eigenvectors is fundamental for the Singular Value Decomposition(SVD) [3] technique, Principal Component Analysis(PCA) [4], Data Compression and Feature Selection algorithms. In SVD, the matrix of singular values is deduced using Jacobi rotation matrices, which is basically a generalized form of eigenvalue decomposition [5]. Again, these eigenvalue decomposition are used in computing the principal components in PCA.

As mentioned, Jacobi Eigenvalue Algorithm is applied in a numerous machine learning applications, it has to be computationally efficient. However in this algorithm, a large number of rotations are performed on matrix in order to diagonalize the matrix. In addition to that, for every rotation to be performed, the maximum element is identified to be treated as pivot, which adds up to the computations. Hence, the whole procedure results to be computationally expensive in terms of time complexity. In this paper we target to achieve a significant speedup, by performing these iterative matrix rotations in parallel using multi-threading on GPUs.

The rest of the paper is divided into six sections: 1. Related Work, 2. Mathematical Model, 3. Serial Algorithm, 4. Parallel Algorithm, 5. Experimental Results, 6. Analysis and Conclusion

II. Related Work

There are numerous eigenvalue computation algorithms available. Some of the past works show that there is a significant speedup achieved after parallelising specific portion of the eigenvalue algorithm. However these results have hardly been contrasted with performance on GPUs.
M. Aslam [6] compares the performance of GPGPU based hybrid Jacobi solvers. They points down that the Jacobi iterative method has a high scope for parallel computation, as on a conceptual level it is mere a system of linear equations.
S. Dabhi [2] mentions that their optimized eigenvalue-eigenvector algorithm implementation can achieve up to 4.5x speed up over NumPy for applications requiring partial eigenvectors on CPU, which can be further speeded up using clever multi-threading techniques.
L. Bergamaschi [7] provides that the parallel Jacobi-Davidson(JD) implementation, based upon data-parallel

techniques, records a satisfactory speedup, when the number of processors is not too large with respect to the amount of data to be managed. This experimental study was performed on T3E 1200 machine of CINECA Consortium made by a set of DEC-Alpha 21164 processors.
J. Baida [8] proposed two parallel algorithms for eigenvalue computation. [8] Both the algorithms quite effectively parallelised the sequential method. However, the poor results of the sequential version produce low speedups when they compared the parallel methods with the best available sequential algorithm in LAPACK.
S. Singer [9] proposed multiple parallel approaches for trigonometric and hyperbolic Jacobi algorithms using a block-oriented and full-block approach. They used special pivoting of blocks in each sweep and observed that hyperbolic parallel Jacobi was faster than trigonometric parallel Jacobi algorithm. A significant speedup was observed in full-block approach also because of the dramatic reduction in number of sweeps.

III. Mathematical Model

A. Mathematical Intuition of EigenValues-EigenVectors

In this paper, we assume the matrix being a Hermitian matrix. This assumption implies that computation is performed on a real-symmetric matrix which always have only real eigenvalues and eigenvectors. Considering, A as a matrix or a linear transformation, λ is called the eigenvalue and $\vec{v}$ is the eigenvector if it satisfies the condition:

$$A\vec{v} = \lambda\vec{v}$$

It can be said that λ can be an eigenvalue if and only if the above equation has a trivial solution. We can re-write the equation to get:

$$(A\text{-}\lambda I)\vec{v} = A\vec{v}\text{-}\lambda\vec{v} = \vec{0}$$

The above equation is termed as "characteristic equation" [1] which provides us with eigenvalues of that particular matrix or linear transformation. The above equation provides us a non-trivial solution if and only if it satisfies the below equation:

$$\mid A\text{-}\lambda I \mid = 0$$

All the independent eigenvectors will be computed with the help of the eigenvalues obtained. For every eigenvalues, the basis of the nullspace of (A-λI), in other words, the basis of the subspace of all eigenvalues, gives us independent eigenvectors.

B. Mathematical Intuition of Jacobi Algorithm

Assuming A to be Hermitian matrix, and T = T(i,j,θ) as the transformation matrix also known as rotation matrix, then

$$A' = TAT^T$$

is also a real symmetric matrix having same Frobenius Norm as that of matrix A. We choose an angle θ such that A'$_{ij}$ = 0. In such a scenario, A' has the larger sum of squares on the diagonal:

$$A'_{ij} = \cos(2\theta)A_{ij} + \frac{1}{2}\sin(2\theta)(A_{ii}\text{-}A_{jj})$$

Now, setting A'$_{ij}$ equal to 0,

$$\tan(2\theta) = \frac{2A_{ij}}{A_{jj}-A_{ii}}$$

Considering A$_{jj}$ = A$_{ii}$, we get $\theta = \frac{\pi}{4}$. This method perform rotations until the matrix becomes almost diagonal. The elements of the diagonal are the eigenvalues of the matrix A.

IV. Serial Algorithm

The mathematical model provided the conceptual view for the computation of eigenvalues. For implementation, the mentioned mathematical computation is done using the following basic process:

- Find kth and lth row and column of A which corresponds to off diagonal element having highest value.
- Compute the Jacobi matrix T after calculating the angle of similarity rotation.
- Apply T to A till when the matrix A is completely transformed to a diagonal matrix.
- The diagonal element will be eigenvalues.
- The columns of T will be eigenvectors.

Algorithm 1 shows the main Jacobi procedure from where the computation begins. As an input parameter, matrix A of size N is provided and for this experiment we have set the value of tolerance to 10^{-9}. RotateMax stores the maximum value for the rotations to be performed, which can be given in order of N^2. Here, we set the limit of rotations to 5N^2. Here, we initialized the transformation matrix T as an identity matrix. Two sub-procedures are executed namely MaximumValue and MatrixRotate until the maximum rotation limit is reached. Procedure MaximumValue computes the maximum element (max) in A and returns the index (k,l) of the same. For every iteration, the maximum value is computed, which is treated as the pivot element with respect to which, the matrix is rotated. The worst-case time complexity of this procedure is O(N^2), which is improved to an average-case time complexity of O(N).

Algorithm 2 shows the MatrixRotate Procedure, which is used to rotate the matrix A in each iteration until it converges to give a diagonal matrix. The input parameters include, matrix A, transformation matrix T, pivot index k and l. In this procedure, the tangent (tanθ) is computed with respect to the pivot element and is stored in variable 't'. Variables 'c' and 's' represent the elements sinθ and cosθ of the rotation matrix. Using these values, rows and columns are rotated as shown in the pseudocode. On rotating T, the eigenvectors are updated in the matrix T. The average-case time complexity of one sweep of rotation is O(N^3), which is equivalent to a single matrix multiplication process.

Algorithm 1 Serial Jacobi Algorithm

```
procedure JACOBI($A$,$tolerance$)
    $n = A.length()$
    $RotateMax = setting\ no.\ of\ rotations$
    $T = initialize\ transformation\ matrix$
    for $i \leftarrow 1, RotateMax$ do
        $max, k, l = MaximumValue(A)$
        if $max < tolerance$ then
            $returndiagonal(A)$
        end if
        $MatrixRotate(A, T, k, l)$
    end for
end procedure
```

A. Profiling Results of Serial Jacobi

The above serial algorithm is implemented by setting a loop for varying size of the matrix A from 2^2x2^2 to 2^{10}x2^{10} consisting of single precision float values between range -20 to 20 at a tolerance value of 10^{-9}. The serial code is written in Python and implemented on Nvidia GeForce GTX-680 GPU. We used python memory-profiler for memory profiling and cProfile for time profiling of the code.

Size	MV(sec)	Calls	MR(sec)	Calls
4x4	0.000	4	0.000	3
8x8	0.001	18	0.004	17
16x16	0.004	47	0.019	46
32x32	0.032	121	0.095	120
64x64	0.251	277	0.396	276
128x128	1.982	643	1.587	642
256x256	17.547	1416	7.112	1415
512x512	152.835	3061	30.660	3060

Table 1 Profile of Jacobi Serial

Table 1 shows the profiling results of the serial code. Here, MV represents the time utilization of MaximumValue function and MR represents the time utilization of MatrixRotate function.

V. Parallel Algorithm

The profiling results shows a wide scope of parallelisation in both maximum element computation and rotation part of the algorithm. The current section deals with all possible parallelisation we performed on the Jacobi algorithm to improve time complexity using CUDA. While designing the parallel algorithm, one significant observation was made that the rotation of the blocks done in the serial algorithm is an independent operation. Hence, the off-diagonal elements of these blocks are eliminated in parallel as shown below:

$$\begin{bmatrix} T_{xx} & T_{xy} \\ T_{yx} & T_{yy} \end{bmatrix}^T \begin{bmatrix} A_{xx} & A_{xy} \\ A_{yx} & A_{yy} \end{bmatrix} \begin{bmatrix} T_{xx} & T_{xy} \\ T_{yx} & T_{yy} \end{bmatrix} = \begin{bmatrix} A'_{xx} & 0 \\ 0 & A'_{yy} \end{bmatrix}$$

Having no dependence amongst rows and columns, subsequent

Algorithm 2 MatrixRotate for Serial Jacobi

```
procedure MATRIXROTATE($A$,$T$,$k$,$l$)
    $n = A.length()$
    $Anew = A[l,l] - A[k,k]$
    if $A[k,l] < Anew * tolerance$ then
        t = A[k,l]
    else
        $\phi$ = Anew/2*A[k,l]
        t = 1/($\phi$ + $\sqrt{\phi^2 + 1}$)
        if $\phi < 0$ then
            t = -t
        end if
    end if
    c = 1/$\sqrt{t^2 + 1}$
    s = t*c
    temp = A[k,l]
    A[k,l] = 0
    A[k,k] = A[k,k] - t*temp
    A[l,l] = A[l,l] + t*temp
  $- - Rotation\ of\ k\ and\ l\ rows\ columns - -$
    for $i \leftarrow 1, k$ do
        $\begin{bmatrix} A_{ik} \\ A_{il} \end{bmatrix} := \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} A_{ik} \\ A_{il} \end{bmatrix}$
    end for
    for $i \leftarrow k+1, l$ do
        $\begin{bmatrix} A_{ki} \\ A_{il} \end{bmatrix} := \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} A_{ki} \\ A_{il} \end{bmatrix}$
    end for
    for $i \leftarrow l+1, n$ do
        $\begin{bmatrix} A_{ki} \\ A_{li} \end{bmatrix} := \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} A_{ki} \\ A_{li} \end{bmatrix}$
    end for
  $- - Rotation\ of\ Eigenvectors - -$
    for $i \leftarrow 1, n$ do
        $\begin{bmatrix} T_{ik} \\ T_{il} \end{bmatrix} := \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} T_{ik} \\ T_{il} \end{bmatrix}$
    end for
end procedure
```

row rotations and column rotations are performed in parallel as well.

Algorithm 3 is the main Jacobi Parallel procedure from where all the kernels are executed. It takes matrix A and tolerance as input arguments. We here set the value of tolerance to 10^{-9}, same as that of serial implementation. First the 'xychange' kernel is executed, which updates the pivot elements(max and min) for each block and stores the values in the dx and dy arrays respectively. By doing this, a huge amount of computation time is reduced for evaluating the pivot

Algorithm 3 Parallel Jacobi Algorithm

```
procedure JACOBIPARALLEL($A$,$tolerance$)
    N = $A.length()$
    double *dA, *c, *s;
    int *dN, *dx, *dy;
    $Allocate\ dA \leftarrow N * N(cudaMalloc)$
    $Allocate\ dN, dx, dy \leftarrow N * (N-1)/2(cudaMalloc)$
    $dA \leftarrow A(cudaMemcpy)$
    $dN \leftarrow N(cudaMemcpy)$
    xychange<<<N-1, N/2>>>(N, dx, dy);
    while not $converged$ do
        for $i \leftarrow N-1$ do
            *X = dx+(i*N/2);
            *Y = dy+(i*N/2);
            cosAndsine<<<N/2, 1>>>(dN,dA,c,s);
            cudaDeviceSynchronize();
            RotateRows<<<N/2,N>>>(dN,dA,Atmp,c,s);
            RotateRows<<<N/2,N>>>(dN,dA,Atmp,c,s);
            cudaDeviceSynchronize();
            RotateCols<<<N/2,N>>>(dN,Atmp,dA,c,s,
            X,Y);
            RotateCols<<<N/2,N>>>(dN,T,Ttmp,c,s,X,Y);
            cudaDeviceSynchronize();
            getEv<<<N,N>>>(Ttmp,T);
            cudaDeviceSynchronize();
        end for
    end while
    $A \leftarrow dA(cudaMemcpy)$
    $Free\ the\ allocated\ memories\ (cudaFree)$
end procedure
```

element. This kernel follows the below thread mapping:

tid = (threadidx.x)*(N/2) + blockidx.x

Similar to the serial algorithm, the iterations are performed until the resultant matrix is converged or reached to the set tolerance value. For each iteration in the loop, the rotation matrix is computed in 'cosAndsin' kernel and values of cosθ and sinθ are stored in c and s arrays.

The major parallelisation is done in *Algorithm 4* RotateCols and *Algorithm 5* RotateRows kernels, where the columns and rows are rotated respectively, by applying the rotation matrix achieved through the previous kernel execution. X and Y arrays stores the pivot elements (max and min) of the blocks. These elements are used for multiple times, hence we used shared memory to store them after they are accessed for the first time. RotateCols is executed again for rotating the eigenvalues in the matrix T, which was also the last step in serial algorithm for matrix rotation. By performing the operations in parallel, the time complexity is reduced to the order of O(n). *Algorithm 3* ends by copying corresponding eigenvectors to the matrix T computed on GPU. It is done using the final kernel 'getEv'. This kernel simply copies the eigenvectors element-wise from Ttmp to T in every iteration

Algorithm 4 RotateCols Kernel

```
procedure ROTATECOLS($N$, $A$, $ResA$, $c$, $s$, $X$,$Y$)
    __shared__ int m, n;
    __shared__ double cosine, sine;
    if $threadIdx.x == 0$ then
        m = X[blockIdx.x];
        n = Y[blockIdx.x];
        cosine = c[blockIdx.x];
        sine = s[blockIdx.x];
    end if
    __syncthreads();
    ele1 = m*(*N)+threadIdx.x;
    ele2 = n*(*N)+threadIdx.x
    ResA[ele1] = cosine*A[ele1] - sine*A[ele2];
    ResA[ele2] = sine*A[ele1] + cosine*A[ele2];
end procedure
```

Algorithm 5 RotateRows Kernel

```
procedure ROTATEROWS($N$, $A$, $ResA$, $c$, $s$, $X$,$Y$)
    __shared__ int m, n;
    __shared__ double cosine, sine;
    if $threadIdx.x == 0$ then
        m = X[blockIdx.x];
        n = Y[blockIdx.x];
        cosine = c[blockIdx.x];
        sine = s[blockIdx.x];
    end if
    __syncthreads();
    ele1 = m*(*N)+threadIdx.x;
    ele2 = n*(*N)+threadIdx.x
    if flag == 0 then
        ResA[ele1] = cosine*A[ele1] - sine*A[ele2];
    else
        ResA[ele2] = sine*A[ele1] + cosine*A[ele2];
    end if
end procedure
```

using the following thread-mapping:

tid = threadidx.x + blockidx.x*blockdim.x

After the matrix is converged, eigenvalues are obtained at the diagonal position of matrix A.

A. Profiling Results of Parallel Algorithm

We implemented the parallel algorithm by setting a loop for varying size of the real symmetric matrix A from 2^2x2^2 to 2^{10}x2^{10} consisting of single precision float values between range -20 to 20. The profiling of the code is done using the nvprof profiler on NVidia GeForce GTX-680 GPU. *Table 2* shows the overall utilization of GPU by all kernels.

Kernel	GPUTime(%)	Time(ms)
RotateRows (x2)	34.39 + 34.21	118.3 + 117.7
RotateCols	17.88	61.534
getEv	8.25	28.380
cosAndsine	4.16	14.324
xychange	0.1	0.03

Table 2 Profile of Jacobi Parallel

It can be evident from the profile results that there a significant improvement in the computation time when compared to the previously mentioned serial implementation.

VI. Experimental Results

This section shows the experimental results of the implementation of both algorithms on NVidia GeForce GTX-680 GPU, having a compute capability of 3.0. *Figure 1* shows the size versus time curve for the comparison of the serial and parallel Jacobi Algorithm. One observation that can be made from the curve is that, the serial algorithm is quite efficient for the smaller matrix sizes. As the size increases, there is a sudden exponential increase in time after 2^7x2^7 matrix size. On the other hand, parallel algorithm very efficiently can handle the increase in size of the matrix. *Table 3* shows the exact readings of the run-time for each matrix size.

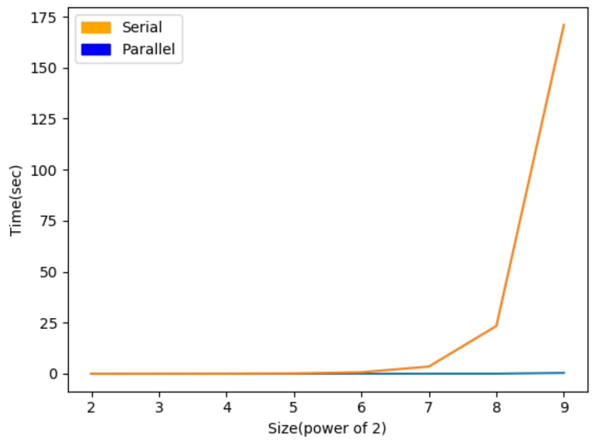

Figure 1. Size v/s Time Curve

Size(NxN)	Serial Time(sec)	Parallel Time(sec)
4 x 4	0.00079	0.00143818
8 x 8	0.00447	0.00182058
16 x 16	0.0249	0.0034308
32 x 32	0.1346	0.0063977
64 x 64	0.728	0.011676
128 x 128	3.510	0.023544
256 x 256	23.410	0.023039
512 x 512	171.039	0.406095

Table 3 Readings

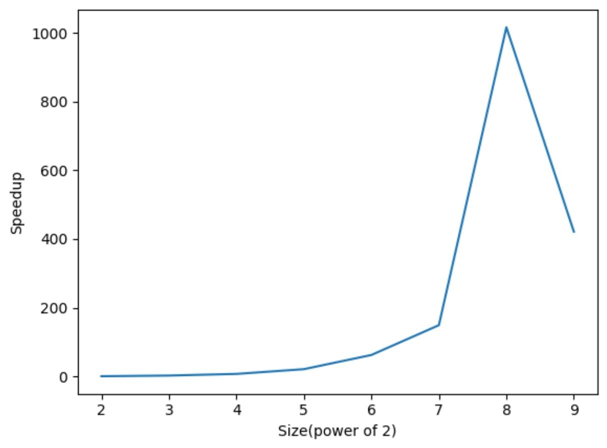

Figure 2. Size v/s SpeedUp Curve

Speedup is calculated using the serial to parallel time ratio. *Figure 2* shows the Speedup curve. It is observed that a significant speedup is achieved after parallelising a numerous computationally expensive operations of the Jacobi eigenvalue algorithm. There is a major contribution of the RotateRows and RotateCols kernels in achieving the mentioned results.

VII. Analysis and Conclusion

Beginning from the serial implementation, it is quite evident that the performance of the algorithm degrades with increase in the size of the matrix. A number of other observations here include the time consumed by individual procedures i.e. MaximumValue and MatrixRotate. Even though rotation of matrix was estimated theoretically to be more expensive than pivot element calculation, the profiling results shows that as the size increases the scenario is changed. Computing maximum element was consuming more time than the rotation procedure for sizes 2^7 x 2^7 and higher. This shortcoming can be further improved in the serial code itself by using an efficient way to compute maximum value.

We here contrasted the results of parallel implementation up-to 2^9 x 2^9 for the reason that the serial algorithm results were comparatively higher and the comparison would not have been much visible on the graph. However, the parallel algorithm is quite efficient for even larger matrix sizes. Looking at the SpeedUp curve, a conclusion can be drawn, that for smaller matrix sizes there is not much difference in the execution time. However, it can be seen that for latter sizes, parallel algorithm is much faster than serial. In this paper, the parallel part was highly focused on parallelising the rotation part of the Jacobi algorithm, which can further be improved by improving the parallelisation applied to maximum element computation of the algorithm as well.

References

[1] H. Rutishauser, “The jacobi method for real symmetric matrices,” *Numerische Mathematik*, vol. 9, no. 10, pp. 1–10, 1966.

[2] S. Dabhi and M. Parmar, “Eigenvector component calculation speedup over numpy for high-performance computing,” 2020.

[3] R. P. Brent and F. T. Luk, “The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays,” *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 69–84, 1985. [Online]. Available: https://doi.org/10.1137/0906007

[4] I. Jolliffe, *Principal Component Analysis*. American Cancer Society, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat06472

[5] J. W. Ketchum, J. R. Walton, M. S. Wallace, S. J. Howard, and H. Inanoglu, “Eigenvalue decomposition and singular value decomposition of matrices using jacobi rotation,” Feb. 22 2011, uS Patent 7,895,254.

[6] M. Aslam, O. Riaz, S. Mumtaz, and A. D. Asif, “Performance comparison of gpu-based jacobi solvers using cuda provided synchronization methods,” *IEEE Access*, vol. 8, pp. 31 792–31 812, 2020.

[7] L. Bergamaschi, G. Pini, and F. Sartoretto, “Computational experience with sequential and parallel, preconditioned jacobi–davidson for large, sparse symmetric matrices,” *Journal of Computational Physics*, vol. 188, pp. 318–331, 06 2003.

[8] J. BADÍA and A. VIDAL, “Parallel algorithms to compute the eigenvalues and eigenvectors ofsymmetric toeplitz matrices,” *Parallel Algorithms and Applications*, vol. 13, no. 1, pp. 75–93, 1998. [Online]. Available: https://doi.org/10.1080/01495739808947361

[9] S. Singer, S. Singer, V. Novaković, A. Ušćumlić, and V. Dunjko, “Novel modifications of parallel jacobi algorithms,” *Numerical Algorithms*, vol. 59, no. 1, p. 1–27, Jun 2011. [Online]. Available: http://dx.doi.org/10.1007/s11075-011-9473-6