Novel GPU Implementation of Jacobi Algorithm for Karhunen-Loève Transform of Dense Matrices

Mustafa U. Torun, Onur Yilmaz, and Ali N. Akansu

***Abstract*—Jacobi algorithm for Karhunen-Loève transform of a symmetric real matrix, and its parallel implementation using chess tournament algorithm are revisited in this paper. Impact of memory access patterns and significance of memory coalescing on the performance of the GPU implementation for the parallel Jacobi algorithm are emphasized. Two novel memory access methods for the Jacobi algorithm are proposed. It is shown with simulation results that one of the proposed methods achieves 77.3% computational performance improvement over the traditional GPU methods, and it runs 73.5 times faster than CPU for a dense symmetric square matrix of size 1,024.**

***Index Terms*—Eigendecomposition, KLT, Jacobi Algorithm, GPU Computing, CUDA**

I. Introduction

Karhunen-Loève transform (KLT) perfectly decorrelates a signal source (diagonalizes its correlation matrix), and has been successfully used in various engineering applications [1–3]. Finding numerical KLT solutions for large dense matrices is still an active research topic in applied mathematics and engineering. Although Jacobi algorithm for eigendecomposition problem was introduced in 1846 [4] it did not generate much interest until the last few decades due to its computational load and implementation cost. However, it was theoretically shown that it is the most stable numerical algorithm among the popular ones including QR [5]. Recently, Jacobi algorithm has become more feasible to implement with the dramatic advances in computational power of general purpose graphic processing units (GPGPU or GPU) offering high performance parallel computing [6, 7].

In this paper, we implement the parallel Jacobi algorithm to calculate the KLT of large dense symmetric matrices with the chess tournament ordering procedure on GPU coded in CUDA™ [8]. We emphasize on the significance of memory access patterns on the performance of the implementation of the algorithm. We propose two novel memory access methods exploiting the symmetry of the input matrix, and availability of the shared memory among GPU threads. We present the experimental results of processing time for each access method and its speed-up with respect to CPU for performance comparisons. To the best of our knowledge, this is the first study utilizing the Fermi™ [9] architecture in the implementation of the parallel Jacobi algorithm reported in the literature.

The paper is organized as follows. In Section 2, we first revisit the classical, cyclic, and parallel Jacobi algorithms for the eigendecomposition problem. We briefly discuss computing on GPU, and effects of memory access patterns on the performance of GPU computing. Then, we stress the importance of memory coalescing when accessing the global memory of the GPU in the context of parallel Jacobi algorithm with the help of two methods that we call "Traditional Access (TA)" and "Modified Access (MA)." Next, we forward two novel access methods to improve the memory coalescing in the Jacobi algorithm. We name these methods as "Symmetrical Access (SA)" and "Maximum Coalesced Access (MCA)." We conclude Section 2 with an example of MCA method for a 4×4 matrix. In Section 3, we provide simulation results for performance testing of CPU implementation and GPU implementations with TA, MA, SA, and MCA methods, and make the remark that the MCA is the best among other access methods considered in this paper. We summarize the contributions and conclude the paper in Section 4.

The authors are with the New Jersey Institute of Technology, Department of Electrical and Computer Engineering, University Heights, Newark, NJ 07102 USA (Email: akansu@njit.edu)

II. Jacobi Algorithm

In this section, we revisit the Jacobi algorithm for the eigendecomposition problem. Let $\mathbf{A}$ be a symmetric data matrix of size $N \times N$, i.e. $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$ and $[A_{ij}] = [A_{ji}]$ where A_{ij} is the element of matrix $\mathbf{A}$ located on i*th* row and j*th* column. Eigen decomposition of matrix $\mathbf{A}$ is given as [1]

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{\mathrm{T}}, \tag{1}$$

where Λ is a diagonal matrix comprising of the eigenvalues of $\mathbf{A}$, $\lambda_1, \lambda_2, \ldots, \lambda_N$, and $\mathbf{V}$ is an $N \times N$ KLT matrix defined as

$$\mathbf{V} = \left[\begin{array}{cccc} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_N \end{array} \right],$$

$\mathbf{v}_i$ is an $N \times 1$ eigenvector paired with the i*th* eigenvalue, λ_i. Jacobi algorithm provides an approximated numerical solution to (1) by iteratively reducing the diagonalization metric [10]

$$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{N} \sum_{\substack{j=1 \\ i \neq j}}^{N} a_{ij}^2}, \tag{2}$$

by multiplying matrix $\mathbf{A}$ from the left and right with the Jacobi rotation matrix, $\mathbf{J}(p, q, \theta)$, and overwriting onto itself as expressed

$$\mathbf{A}^{(k+1)} = \mathbf{J}^{\mathrm{T}}(p, q, \theta)\mathbf{A}^{(k)}\mathbf{J}(p, q, \theta), \tag{3}$$

where $1 \leq p < q \leq N$ and $k > 0$ is the iteration index. Matrix $\mathbf{J}(p, q, \theta)$ is sparse as defined

$$[J(p, q, \theta)_{ij}] = \begin{cases} \cos\theta & i = p,\ j = p \\ \sin\theta & i = p,\ j = q \\ -\sin\theta & i = q,\ j = p \\ \cos\theta & i = q,\ j = q \\ 0 & otherwise \end{cases}. \tag{4}$$

Note that the only difference between the identity matrix $\mathbf{I}$ and $\mathbf{J}(p,q,\theta)$ of the same size is that the elements J_{pp}, J_{pq}, J_{qp}, and J_{qq} are of non-zero. Matrix multiplications in (3) are repeated until $\text{off}(\mathbf{A}) < \epsilon$ (2) where ϵ is a predefined diagonalization threshold value. After sufficient number of consecutive rotations, matrix $\mathbf{A}$ gets closer and closer to the diagonal matrix Λ, and the successive matrix multiplications lead us to an approximation of the eigenvector matrix $\mathbf{V}$ expressed as [10]

$$\mathbf{V} \simeq \mathbf{J}(p_1,q_1,\theta_1)\mathbf{J}(p_2,q_2,\theta_2)\cdots\mathbf{J}(p_L,q_L,\theta_L),$$

where L is a large integer number. Elements of $\mathbf{J}(p,q,\theta)$, i.e. $c=\cos\theta$ and $s=\sin\theta$, (implicitly the angle θ) are chosen such a way that the following multiplication yields a diagonal matrix

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix}\begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix}\begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} \tilde{A}_{pp} & 0 \\ 0 & \tilde{A}_{qq} \end{bmatrix}, \tag{5}$$

where A_{pq} and $\tilde{A}_{pq}$ are elements of $\mathbf{A}^{(k)}$ and $\mathbf{A}^{(k+1)}$, respectively. Therefore, we have the following equation

$$A_{pq}\left(c^2-s^2\right)+(A_{pp}-A_{qq})cs=0,$$

that can be solved for c and s since the elements A_{ij} of the data matrix are known. Note that p and q define the rotation matrix $\mathbf{J}(p,q,\theta)$ through (5). Hence, we can drop the angle θ and refer it as $\mathbf{J}(p,q)$. Originally, p and q in (3) are chosen such that $|A_{pq}| = \max_{i\neq j}|A_{ij}|$ [10]. However, searching for the maximum value at each iteration is not preferred due to computational performance considerations. The most straightforward modification to the classical method is the cyclic Jacobi algorithm that serially cycles through the data matrix by an ordered fashion as expressed in

$$(p^{(m)},q^{(m)}) = \{(1,2),(1,3),(1,4),\ldots,(2,3),\ldots\},$$

where $m=1,2,\ldots N/2$. Such an algorithm can be implemented on a computing device with $N/2$ parallel processing units due to the sparsity of the rotation matrix $\mathbf{J}(p,q)$. Details of the parallel implementation are presented in the subsequent subsection after a brief discussion on the notation we employed in the paper.

A. Notation

In this section, we present the notation employed in order to make the following discussion clear. Let $\mathbf{z}$ be an $N\times 1$ column vector, and $\overline{\mathbf{z}}$ be a $1\times N$ row vector. It is possible to define matrix $\mathbf{Z}$ of size $N\times N$ as

$$\begin{aligned}\mathbf{Z} &= \begin{bmatrix} \mathbf{z}_1 & \mathbf{z}_2 & \cdots & \mathbf{z}_N \end{bmatrix} \\ &= \begin{bmatrix} \overline{\mathbf{z}}_1^{\mathrm{T}} & \overline{\mathbf{z}}_2^{\mathrm{T}} & \cdots & \overline{\mathbf{z}}_N^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}},\end{aligned}$$

where $\mathbf{z}_i$ and $\overline{\mathbf{z}}_i$ are the i*th* column and row of $\mathbf{Z}$, respectively. We define matrices $\mathbf{Z}^{(m)}$ and $\overline{\mathbf{Z}}^{(m)}$ of sizes $N\times 2$ and $2\times N$, respectively, as follows

$$\mathbf{Z}^{(m)} = \begin{bmatrix} \mathbf{z}_{p^{(m)}} & \mathbf{z}_{q^{(m)}} \end{bmatrix}$$

$$\begin{aligned}\overline{\mathbf{Z}}^{(m)} &= \begin{bmatrix} \overline{\mathbf{z}}_{p^{(m)}}^T & \overline{\mathbf{z}}_{q^{(m)}}^T \end{bmatrix}^{\mathrm{T}} \\ &= \begin{bmatrix} Z_{p^{(m)}1} & Z_{p^{(m)}2} & \cdots & Z_{p^{(m)}N} \\ Z_{q^{(m)}1} & Z_{q^{(m)}2} & \cdots & Z_{q^{(m)}N} \end{bmatrix},\end{aligned}$$

where $p^{(m)}$ and $q^{(m)}$ are the integers that define the submatrix of the data matrix to be rotated in the Jacobi algorithm assigned to the m*th* processing unit. Note that when $\mathbf{Z}$ is a symmetric matrix, we have $\mathbf{z}_i = \overline{\mathbf{z}}_i^{\mathrm{T}}$ and $\overline{\mathbf{Z}}^{(m)} = \left[\mathbf{Z}^{(m)}\right]^{\mathrm{T}}$. Next, we define the 2×2 Jacobi sub-rotation matrix used in the m*th* processing unit as shown

$$\mathbf{J}^{(m)} = \begin{bmatrix} J_{p^{(m)}p^{(m)}} & J_{p^{(m)}q^{(m)}} \\ J_{q^{(m)}p^{(m)}} & J_{q^{(m)}q^{(m)}} \end{bmatrix} = \begin{bmatrix} c_m & s_m \\ -s_m & c_m \end{bmatrix}, \tag{6}$$

where J_{ij} is an element of the original Jacobi matrix defined in (4). Since this is an iterative algorithm, we need to state the iteration index, k, explicitly as expressed in (3). However, in order to keep the text clean, we use assignment operator, $\leftarrow$, instead of the iteration index in the paper.

B. GPU Computing

Originally developed for graphics applications, with their massive parallel processing capabilities, state-of-the-art graphical processing units (GPUs) are the leading computing devices for the most parallel and computationally intensive applications. With the introduction of CUDA™ coding by NVIDIA™, that is an extension of the standard C language, high performance DSP and GPU computing gained tremendous momentum over the last few years. From a computing system perspective, a CUDA™ programmer has the option to define a three dimensional parallel *thread block* on a three dimensional *block grid*. A routine to be run on a thread on GPU is called a *kernel.* A *kernel call* runs a kernel on a predefined grid of blocks [8]. Synchronization among kernel calls is orchestrated by the CPU. In the low level, the computing hardware has multiple processors. In each processor, multiple threads execute the same instruction sequence over the same or different data. Threads are grouped together in *warps,* and each warp consists of 32 threads in the Fermi™ architecture [9]. Each processor has a shared memory that is only visible to itself. They also have access to a global memory (usually a DRAM) that is visible to every other processor and to CPU as well. However, accessing the global memory is much slower in time than accessing the shared memory [11–13].

C. GPU Implementation of Parallel Jacobi Algorithm

Jacobi rotation matrix given in (4) is sparse. Thus, it is possible to perform rotations expressed in (3) by a parallel implementation using $N/2$ processing units provided that p and q pairs are unique for each processing unit. For example, two rotations may be implemented in parallel, with $\left(p^{(1)},q^{(1)}\right) = (1,2)$ and $\left(p^{(2)},q^{(2)}\right) = (3,4)$, for $N=4$. In the next step, pairs might be selected as $\left(p^{(1)},q^{(1)}\right) = (1,4)$ and $\left(p^{(2)},q^{(2)}\right) = (2,3)$. Note that a scenario with $\left(p^{(1)},q^{(1)}\right) = (1,2)$ and $\left(p^{(2)},q^{(2)}\right) = (2,4)$ would violate the non-overlap rule since $q^{(1)} = p^{(2)}$, and they must not be run in parallel. There are many possible methods for effectively

choosing the pairs for each step [6, 14]. One of the most popular algorithms is called the chess tournament (CT). In CT, for N players, there are $N/2$ pairs and $N-1$ matches that have to be held such that each player matches against any other player in the group. Once a match set is completed, the first player stands still and every other player moves one seat in clockwise direction. For $N=4$, the pairs for $N-1=3$ steps are defined as

$$\begin{bmatrix} p^{(1)} & p^{(2)} \\ q^{(1)} & q^{(2)} \end{bmatrix} : \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \tag{7}$$

Note that $p^{(2)}$ and $q^{(2)}$ are interchanged in the last step since the condition $p < q$ must hold [10]. However, ensuring that p and q pairs are unique for each processing unit handles only one of the two problems that arise in the parallel implementation. The second problem corresponds to the overlapping of matrix elements in the operations shown in (3). Since the multiplication $\mathbf{J}(p,q)^{\mathrm{T}}\mathbf{A}$ in (3) would update the p*th* and q*th* rows of matrix $\mathbf{A}$, and multiplication $\mathbf{A}\mathbf{J}(p,q)$ in (3) would update the p*th* and q*th* columns of matrix $\mathbf{A}$, it would be problematic to implement (3) in parallel without proper synchronization. The solution is to introduce an intermediary matrix $\mathbf{X}$ of size $N \times N$ [7] in the computational process. Then, the first operation is performed by the m*th* processing unit multiplying the $p^{(m)}th$ and $q^{(m)}th$ rows of $\mathbf{A}$ with the transpose of Jacobi sub-rotation matrix, (6), as written

$$\overline{\mathbf{X}}^{(m)} \leftarrow \left[\mathbf{J}^{(m)}\right]^{\mathrm{T}} \overline{\mathbf{A}}^{(m)}. \tag{8}$$

Waiting for all of the processing units to complete their assigned tasks (a blocking synchronization) is required before proceeding to the next kernel that is the multiplication of the $p^{(m)}th$ and $q^{(m)}th$ columns of $\mathbf{X}$ by the Jacobi sub-rotation matrix, (6), as expressed

$$\mathbf{A}^{(m)} \leftarrow \mathbf{X}^{(m)}\mathbf{J}^{(m)}. \tag{9}$$

Note that, the eigenvectors may be updated according to the procedure

$$\mathbf{V}^{(m)} \leftarrow \mathbf{V}^{(m)}\mathbf{J}^{(m)}, \tag{10}$$

in the same kernel since $\mathbf{J}^{(m)}$ is already accessed (or calculated) in (9). Therefore, two kernel calls; one for (8), and one for (9) and (10), are required for a step in a sweep of the parallel Jacobi algorithm. Second kernel call must wait for the first one to complete its task via global synchronization. We implement the parallel algorithm in two GPU kernels running with $N/2$ blocks (processing units) and N threads (one thread for each vector element). Since the number of threads per block is limited in GPUs [8], we introduced additional modifications to the kernels that are beyond the scope of this paper. The global synchronization is realized through CPU (host synchronization).

D. Memory Access in GPU Computing

Memory access (reaching out to a memory location for reading or writing data) time is an important limiting factor of state-of-the-art GPU computing technologies. Even the GPUs with the latest Fermi™ [9] architecture from NVIDIA™, providing L1 and L2 caches for the GPU main memory, suffer from performance degradations when the memory access patterns by different threads are unstructured and/or non-coalesced [8, 13]. Memory coalescing, that is refining the memory access pattern such that the hardware can make combined requests from DRAM, should be employed whenever applicable. An example of an unstructured and non-coalesced access pattern is shown in Fig. 1a for an application with 4 threads and 8 memory locations. In the figure, first and second iterations are depicted with solid and dashed lines, respectively. It is observed from Fig. 1a that all threads access memory locations in an unstructured way at each iteration step. Although the example depicted in Fig. 1b is a more structured one compared to the one shown in Fig. 1a, it is still considered as non-coalesced access since threads do not access the adjacent memory locations. An example of a coalesced access pattern that ensures the maximum DRAM efficiency [13] is displayed in Fig. 1c. In the following subsections, we stress the effects of various access patterns on the performance of the parallel Jacobi algorithm, and propose two novel modifications to the algorithm in order to increase the coalesced access.

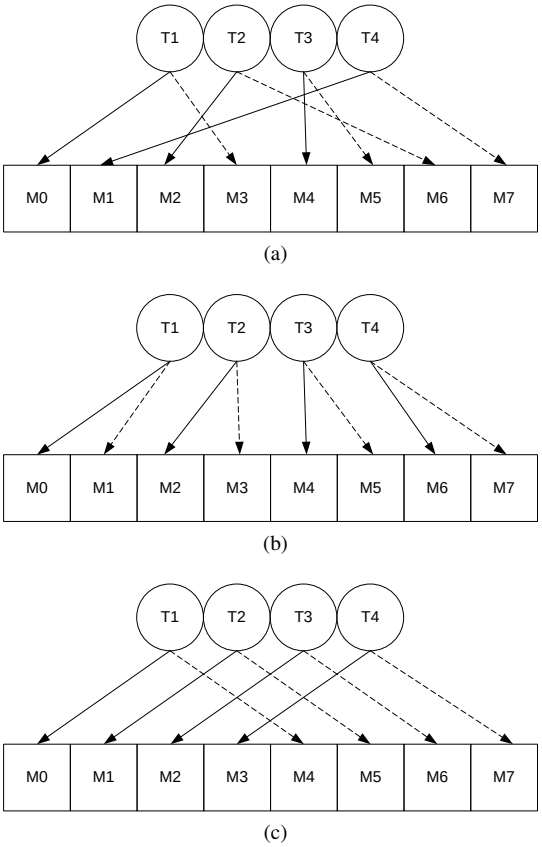


Figure 1: Examples for (a) unstructured and non-coalesced, (b) structured but non-coalesced, and (c) coalesced memory access patterns for a kernel call with 4 threads accessing to 8 memory locations in two iterations. T and M stand for thread and memory, respectively. First and second iterations are depicted with solid and dashed lines, respectively.

E. Traditional and Modified Memory Access Methods

A linear array is the most common data structure used to store dense matrices in a computer memory. Programmers, in general, design the array as *row-major* or *column-major* where the elements are linearized based on their rows and columns, respectively. Let $\mathbf{Z}$ be a 4×4 matrix as given

$$\mathbf{Z} = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}.$$

Then, row-major and column-major arrays for storing the matrix Z become

$$\mathbf{z}_R \triangleq \begin{bmatrix} Z_{11} & Z_{12} & Z_{21} & Z_{22} \end{bmatrix}$$
$$\mathbf{z}_C \triangleq \begin{bmatrix} Z_{11} & Z_{21} & Z_{12} & Z_{22} \end{bmatrix},$$

respectively. We implement the parallel Jacobi algorithm in GPU using row-major arrays to store $\mathbf{A}$, $\mathbf{X}$, and $\mathbf{V}$ in (8), (9), and (10) in memory, and let CUDA threads access them, accordingly. We call this access method as "the Traditional Access (TA)." Note that TA leads to a natural coalesced access in both reading from $\mathbf{A}$ and writing to $\mathbf{X}$ in (8) that corresponds to the access scheme given in 1c. However, both reading and writing operations lead to a non-coalesced access in (9) and (10) that corresponds to the access scheme given in 1b.

A straightforward way to handle this concern is to employ a row-major array to store matrix $\mathbf{A}$, and column-major arrays to store matrices $\mathbf{X}$ and $\mathbf{V}$. We call this access method as "the Modified Access (MA)." MA leads to a non-coalesced access only when writing to matrix $\mathbf{X}$ in (8), and when writing to matrix $\mathbf{A}$ in (9). Other than those two, all access patterns in MA become coalesced that helps the GPU to access its DRAM more efficiently and provide results faster. In the performance comparisons section, we show that the computational performance improves significantly in MA method compared to TA. In the next subsection, we introduce a novel modification to MA method ensuring full coalesced access in performing the tasks of (9).

F. Symmetric Access Method

Since $\mathbf{A}$ is symmetric, its i*th* row is identical to its i*th* column at all times. Therefore, a processing unit can update the p*th* and q*th* rows of $\mathbf{A}$ in (9) instead of updating the p*th* and q*th* columns as given (9). In other words, modifying (9) as

$$\overline{\mathbf{A}}^{(m)} \leftarrow \left[\mathbf{X}^{(m)}\mathbf{J}^{(m)}\right]^{\mathrm{T}}, \tag{11}$$

leads us to the same solution. Explicitly, after every processing unit completes the update given in (11), matrix $\mathbf{A}$ is updated in the same way as it would be updated with (9) since $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$. However, this modification ensures the coalesced access when writing into matrix $\mathbf{A}$ that improves the computational efficiency. We name this access method as "the Symmetric Access (SA)" and show its performance superiority over MA in the performance comparisons section.

Note that even with SA, there is still one non-coalesced access in (8) when writing into matrix $\mathbf{X}$. The trick we used in SA for matrix $\mathbf{A}$ can not be applied directly to $\mathbf{X}$ since it is not symmetrical. Nevertheless, we introduce a new method that also provides coalesced access to the memory locations reserved for $\mathbf{X}$ in the next subsection.

G. Maximum-Coalesced Access Method

By changing the nature of the update procedure in implementing (8) it is possible to ensure all memory access of reading and writing operations in the parallel Jacobi algorithm are coalesced. Note that the proposed modification makes use of the predetermined nature of the chess tournament algorithm (7). We call this method "the Maximum-Coalesced Access (MCA)." In MCA, the update given in (8) is modified as follows

$$\mathbf{X}^{(m)} \leftarrow \mathbf{K}\left[\overline{\mathbf{A}}^{(m)}\right]^{\mathrm{T}}, \tag{12}$$

where $\mathbf{K}$ is an $N \times N$ matrix defined as

$$[K_{ij}] = \begin{cases} J_{11}^{(m)} & i = p^{(m)},\ j = p^{(m)} \\ J_{21}^{(m)} & i = p^{(m)},\ j = q^{(m)} \\ J_{12}^{(m)} & i = q^{(m)},\ j = p^{(m)} \\ J_{22}^{(m)} & i = q^{(m)},\ j = q^{(m)} \\ 0 & otherwise \end{cases}, \tag{13}$$

$J_{ij}^{(m)}$ is the element of Jacobi sub-rotation matrix, (6), and $m = 1, 2, \ldots, N/2$. Note that $\mathbf{K}$ is constant for all processing units in a sweep. In MCA, m*th* processing unit updates the $p^{(m)}th$ and $q^{(m)}th$ columns of $\mathbf{X}$ as follows

$$\begin{aligned} X_{ip^{(m)}} &= A_{ip^{(m)}}K_{ii} + A_{f(i)p^{(m)}}K_{if(i)} \\ X_{iq^{(m)}} &= A_{iq^{(m)}}K_{ii} + A_{f(i)q^{(m)}}K_{if(i)}, \end{aligned} \tag{14}$$

where $i = 1, 2, \ldots, N$ and $f(\cdot)$ is a mapping defined as

$$f(x) \triangleq g(x) \cup g^{-1}(x),$$

$g(x)$ is a mapping from set $\left\{p^{(m)}\right\}$ to set $\left\{q^{(m)}\right\}$ expressed as

$$g : p^{(m)} \rightarrow q^{(m)}.$$

Note that g is one-to-one. Hence, its inverse g^{-1} exists. Since sets $\left\{p^{(m)}\right\}$ and $\left\{q^{(m)}\right\}$ are known in advance in the algorithm, it is feasible to realize the update equation given in (14) . Moreover, since (14) accesses only to $2N$ elements of $\mathbf{K}$, it is more efficient (in terms of memory requirements) to define two $N \times 1$ vectors $[u_i] = K_{ii}$ and $[w_i] = K_{if(i)}$ instead of $N \times N$ sized $\mathbf{K}$. Then, one may modify (14) accordingly, as expressed

$$\begin{aligned} X_{ip^{(m)}} &= A_{ip^{(m)}}u_i + A_{f(i)p^{(m)}}w_i \\ X_{iq^{(m)}} &= A_{iq^{(m)}}u_i + A_{f(i)q^{(m)}}w_i. \end{aligned} \tag{15}$$

It is worth noting that we implicitly exploit the inherent symmetry of matrix $\mathbf{A}$ again in MCA, $\left[\overline{\mathbf{A}}^{(m)}\right]^{\mathrm{T}}$ in (12) accounts for accessing the $p^{(m)}th$ and $q^{(m)}th$ rows of $\mathbf{A}$ (in accordance with its row-major array data structure) and using its transpose such that we have a matrix of $N \times 2$. Also, update given in (15) might still lead to non-coalesced access when reading from the global memory if the algorithm is

not coded carefully. Therefore, in the GPU implementation of MCA, whenever applicable, we first read from global memory in a coalesced way and store the values in the shared memory before performing operations on them. Shared memory is faster and almost prune to non-coalesced access [8, 13].

The complexity of the algorithm is significantly increased in MCA compared to the other methods considered in the paper. Moreover, all processing units need to calculate (or share) the rotation matrix for every pair (matrix $\mathbf{K}$) in the algorithm resulting in higher computational load or memory usage per processing unit. However, we show in the performance comparisons section that this overload is highly negligible, and it is well justified by the significance of improvements provided by MCA. Next, an MCA example is presented in the next subsection that explains the method clearly.

H. MCA Example

Let $N = 4$, hence, $\mathbf{A}$ be a 4×4 matrix. Note that $N/2 = 2$ processing units are needed in this case. Assume the algorithm is at its second step in the sweep where $p^{(1)} = 1$, $p^{(2)} = 2$, $q^{(1)} = 4$, and $q^{(2)} = 3$ (7). At this step, matrix $\mathbf{K}$ (13) is written as

$$\mathbf{K} = \begin{bmatrix} J_{11}^{(1)} & 0 & 0 & J_{21}^{(1)} \\ 0 & J_{11}^{(2)} & J_{21}^{(2)} & 0 \\ 0 & J_{12}^{(2)} & J_{22}^{(2)} & 0 \\ J_{12}^{(1)} & 0 & 0 & J_{22}^{(1)} \end{bmatrix}, \tag{16}$$

where $J_{ij}^{(m)}$ is given in (6). Note that for the case at hand, the two arrays in (15) can be expressed as

$$\begin{aligned} \mathbf{u} &= \begin{bmatrix} J_{11}^{(1)} & J_{11}^{(2)} & J_{22}^{(2)} & J_{22}^{(1)} \end{bmatrix} \\ \mathbf{w} &= \begin{bmatrix} J_{21}^{(1)} & J_{21}^{(2)} & J_{12}^{(2)} & J_{12}^{(1)} \end{bmatrix}, \end{aligned} \tag{17}$$

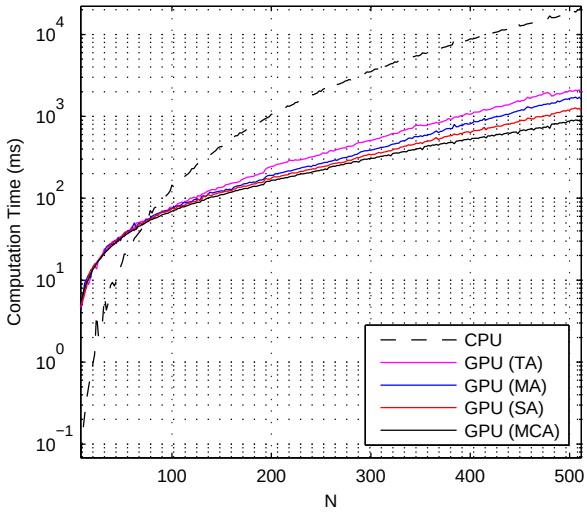
since we have

$$\begin{aligned} g &: \{1 \to 4\} \cup \{2 \to 3\} \\ f &: \{1 \to 4\} \cup \{2 \to 3\} \cup \{4 \to 1\} \cup \{3 \to 2\}. \end{aligned} \tag{18}$$

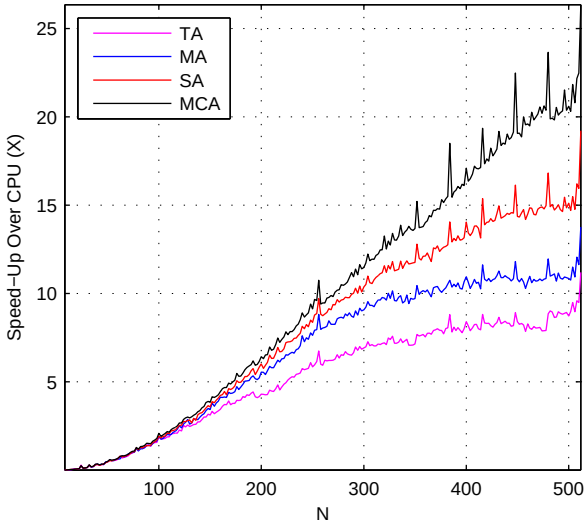
It is clear from (16) and (17) that storing arrays $\mathbf{u}$ and $\mathbf{w}$ instead of matrix $\mathbf{K}$ saves memory space by the elimination of unnecessary storage of the zeros in matrix $\mathbf{K}$. Finally, using (15), (17), and (18) we write the operations that are performed in the first update of the first processing unit as

$$\begin{aligned} X_{11} &= A_{11}K_{11} + A_{41}K_{14} \\ X_{21} &= A_{21}K_{22} + A_{31}K_{23} \\ X_{31} &= A_{31}K_{33} + A_{21}K_{32} \\ X_{41} &= A_{41}K_{44} + A_{11}K_{41}, \end{aligned}$$

where X_{ij} is located at the i*th* row and j*th* column of matrix $\mathbf{X}$. Operations for the second step, and for the both steps of the second processing unit are straightforward. We finish the example by providing the expanded version of (12) within this context in order to stress that MCA method accesses the columns of matrix $\mathbf{X}$ and rows of matrix $\mathbf{A}$ for the update



(a)



(b)

Figure 2: (a) Computation times of cyclic Jacobi algorithm on CPU; TA, MA, SA, and MCA on GPU implementations, for various matrix sizes, N, (b) speed-up of Traditional Access (TA), Modified Access (MA), Symmetrical Access (SA), and Maximum Coalesced Access (MCA) on GPU implementations over cyclic Jacobi algorithm on CPU.

given in (8) as follows

$$\begin{bmatrix} X_{11} & X_{14} \\ X_{21} & X_{24} \\ X_{31} & X_{34} \\ X_{41} & X_{44} \end{bmatrix} = \mathbf{K} \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}^{\mathrm{T}},$$

where matrix $\mathbf{K}$ is defined in (16). Performance improvement of MCA over SA and other methods is quantified in the next section.

III. Performance Comparisons

In this section, we provide and compare the performance of the GPU methods discussed in the paper and cyclic CPU

Table I: Computation time in milliseconds for CPU (first row) and for GPU implementations with different memory access patterns (second to last rows) versus the input matrix size, N.

	$N=8$	$N=16$	$N=32$	$N=64$	$N=128$	$N=256$	$N=512$	$N=1024$
CPU	0.1	0.5	6.3	34.5	270.3	2,404.9	22,107.5	378,122.7
GPU (Traditional Access)	4.8	8.8	24.3	49.9	105.6	356.0	1,974.3	22,612.4
GPU (Modified Access)	4.4	11.7	22.0	42.4	94.1	272.6	1,605.7	15,132.2
GPU (Symmetrical Access)	4.6	11.5	21.4	45.9	96.1	247.5	1,150.4	8,868.6
GPU (Maximum Coalesced Access)	5.5	10.3	21.2	42.0	89.7	223.8	839.1	5,143.2

implementation in terms of speed. We performed our tests on a quad core (a total of eight-cores with hyper-threading) Intel(R) Core™ i7 960 with 24 GB RAM machine running on Linux. The GPU used in the tests is an NVIDIA Tesla™ C2070 with 448 CUDA™ Cores and 5375 MB global memory. GPU is installed on the same machine. All floating point operations are performed with single-precision. Timing results are averaged over 20 runs. Number of sweeps in all tests is fixed to 6 in order to make a fair comparison.

Computation time in milliseconds for CPU and various GPU implementations with different memory access patterns (TA, MA, SA, and MCA) as a function of input matrix size are tabulated in Table I, and also displayed in Fig. 2a. Speed-up of TA, MA, SA, and MCA on GPU implementations over cyclic Jacobi algorithm on CPU versus the input matrix size is shown in Fig. 2b. It is worth noting that the local peaks in Fig. 2a correspond to the input matrix sizes that are multiples of 32 (warp size for Fermi™ architecture). It is known that the GPUs perform better when the number of total threads is an integer multiple of the warp size [8].

As expected, speed of the algorithm is increased as we move from CPU to GPU, and to MCA among the GPU based implementations. Performance of MCA method is superior among memory access methods considered. It is also clearly displayed in Table I and Fig. 2 that GPU implementation even with TA method significantly outperforms CPU implementation when N gets large. The performance improvement of MA method (which is a slightly modified version of TA) over TA is 33.1% for $N = 1024$ that is calculated as follows

$$\left(\frac{t_{TA} - t_{MA}}{t_{TA}}\right) \times 100\%.$$

Similarly, for $N = 1024$, performance improvement of SA over TA and MA are 60.8% and 41.4%, respectively; MCA over TA, MA, and SA are 77.3%, 66.0%, and 42% respectively. For $N = 1024$, The speed-up of TA, MA, SA, and MCA on GPU over CPU are 16.7X, 25.0X, 42.6X, and 73.5X, respectively.

IV. Conclusions

In this paper, we revisited the celebrated Jacobi algorithm for Karhunen-Loève transform calculation of a symmetric real matrix, and its parallel numerical implementation using the chess tournament algorithm. We highlighted the fact that memory is a limiting performance factor in GPU computing, and forwarded several new implementations with a better engineered memory access patterns leading us to drastic performance improvements. Two novel memory access methods are proposed. They are quantified to achieve 77.3% computational performance improvement over the traditional GPU methods, and 73.5 times faster implementation over CPU for a dense symmetric matrix of size $N = 1024$ under the same test conditions.

References

[1] A. N. Akansu and R. A. Haddad, *Multiresolution Signal Decomposition: Transforms, Subbands, and Wavelets.* Academic Press, Inc., 1992.

[2] M. U. Torun, A. N. Akansu, and M. Avellaneda, "Portfolio risk in multiple frequencies," *IEEE Signal Processing Magazine, Special Issue on Signal Processing for Financial Applications*, vol. 28, no. 5, pp. 61 – 71, Sep. 2011.

[3] A. N. Akansu and M. U. Torun, "Toeplitz approximation to empirical correlation matrix of asset returns: A signal processing perspective," *Journal of Selected Topics in Signal Processing*, 2012.

[4] C. G. J. Jacobi, "Über ein leichtes verfahren, die in der theorie der säkularstörungen vorkommenden gleichungen numerisch aufzulösen," *Crelle's Journal*, vol. 30, pp. 51 – 94, 1846.

[5] J. Demmel and K. Veselic, "Jacobi's method is more accurate than QR," *SIAM J. Matrix Anal. Appl.*, vol. 13, pp. 1204 – 1245, 1992.

[6] V. Novakovic and S. Singer, "A GPU-based hyperbolic SVD algorithm," *BIT Numerical Mathematics*, vol. 51, pp. 1009 – 1030, 2011.

[7] G. S. Sachdev, V. Vanjani, and M. W. Hall, "Takagi factorization on GPU using CUDA," in *Symposium on Application Accelerators in High Performance Computing, Knoxville, Tennessee*, Jul. 2010.

[8] *CUDA Programming Guide Version 4.0*, NVIDIA, May 2011.

[9] *Tuning CUDA Applications for Fermi Version 1.0*, NVIDIA, Feb. 2010.

[10] G. H. Golub and C. F. V. Loan, *Matrix Computations*. Johns Hopkins University Press, 1996.

[11] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[12] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[13] R. Farber, *CUDA Application Design and Development*. Elsevier Science, 2011.

[14] F. T. Luk and H. Park, "A proof of convergence for two parallel Jacobi SVD algorithms," *IEEE Transactions on Computers*, vol. 38, no. 6, pp. 806 – 811, Jun 1989.