Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

FPGA, GPU, and CPU implementations of Jacobi algorithm for eigenanalysis[☆]

Mustafa U. Torun, Onur Yilmaz [*], Ali N. Akansu

Electrical and Computer Engineering Department, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA

ARTICLE INFO

Article history:
Received 25 October 2015
Received in revised form
20 May 2016
Accepted 23 May 2016
Available online 31 May 2016

Keywords:
Eigenanalysis
Principal component analysis
Karhunen–Loève transform
Jacobi algorithm
Chess tournament
CORDIC
Memory coalescing
FPGA
GPU
CPU

ABSTRACT

Parallel implementations of Jacobi algorithm for eigenanalysis of a matrix on most commonly used high performance computing (HPC) devices such as central processing unit (CPU), graphics processing unit (GPU), and field-programmable gate array (FPGA) are discussed in this paper. Their performances are investigated and compared. It is shown that CPU, even with multi-threaded implementation, is not a feasible option for large dense matrices. For the GPU implementation, performance impact of the global memory access patterns on the GPU board and the memory coalescing are emphasized. Three memory access methods are proposed. It is shown that one of them achieves 81.6% computational performance improvement over the traditional GPU methods, and it runs 68.5 times faster than a single-threaded CPU for a dense symmetric square matrix of size 1,024. Furthermore, FPGA implementation is presented and its performance running on chips from two major manufacturers are reported. A comparison of GPU and FPGA implementations is quantified and ranked. It is reported that FPGA design delivers the best performance for such a task while GPU is a strong competitor requiring less development effort with superior scalability. We predict that emerging big data applications will benefit from real-time and high performance computing implementations of eigenanalysis for information inference and signal analytics in the future.

1. Introduction

Eigenanalysis plays a key role in an essential tool in engineering and applied mathematics, known as factor analysis, principal component analysis (PCA), or Karhunen–Loève Transform (KLT) [2,33,3]. However, implementation of eigenanalysis demands prohibitive computational resources in most cases. Efficient numerical methods for large dense matrices is still an active research topic. Jacobi algorithm is one of the numerical techniques used for eigenanalysis. Although it was introduced in 1846 [15], it did not generate much interest until the last few decades due to its high computational cost. However, it has an inherent parallelism that is desirable. Moreover, it was shown that it is a more stable numerical algorithm than the popular QR method [10]. Recently, Jacobi algorithm has become more feasible to implement on general purpose graphics processing units (GPGPU or GPU) and field-programmable gate arrays (FPGA) that offer high performance parallel software and hardware computing, respectively.

In traditional software applications, source code is compiled into operational codes (op-codes) according to specific CPU architecture. In single- and multi-threaded CPU applications, there is unavoidable and sometimes significant latency in decoding the op-codes in the control unit of the processing device. In addition, operating systems dealing with multiple tasks introduce additional and unpredictable latency. Although multi-threaded parallel CPU implementations are expected to run faster than the single-threaded counterparts, the overhead of creating, destroying, and synchronizing threads may be very high, especially in the context of eigenanalysis [28]. An alternative parallel computing platform is the GPU. Originally, it was developed for graphics applications. Due to their massive parallel processing capabilities, state-of-the-art GPUs are the leading software computing devices for the most parallel and computationally intensive applications [29]. With the introduction of CUDA^{TM} language by NVIDIA^{TM}, that is an extension of the standard C language, implementing high performance digital signal processing (DSP) algorithms on GPU devices gained

[☆] This paper was presented in parts at the 46th Annual Conference on Information Sciences and Systems (CISS), Princeton, NJ, March 2012; IEEE 13th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC), Cesme, Turkey, June 2012; and International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vancouver, Canada, May 2013.

[*] Corresponding author.

E-mail addresses: onur.yilmaz@njit.edu (O. Yilmaz), akansu@njit.edu (A.N. Akansu).

http://dx.doi.org/10.1016/j.jpdc.2016.05.014
0743-7315/© 2016 Elsevier Inc. All rights reserved.

increasing momentum [17,28,11,24,34,32,36,37]. The third parallel computing device considered in this paper is the FPGA. Allocating dedicated hardware for a specific algorithm, FPGAs have been used to implement many engineering applications in several industries including military, medical, consumer electronics, and wireless communications for nearly three decades. With their massive amounts of digital building-blocks that can be interconnected via a hardware description language (HDL) such as VHSIC HDL (VHDL) [27] or Verilog [31], low-power operation, and relatively cheap cost, FPGAs have become the main-stream hardware parallel computing devices [23]. As it is the case for many technologies, there are drawbacks of FPGAs that include increased implementation time, cost, and complexity.

In this paper, we further the GPU and FPGA implementations of the parallel Jacobi algorithm for the eigenanalysis of real, dense, and symmetric matrices reported in the literature [28,24,34,32,30,6,8,13,16,1,5,20], explain the improved designs in detail, and report a performance comparison between CPU, GPU, and FPGA implementations in terms of speed and computation error. Improvements suggested in the paper for GPU and FPGA implementations are summarized as faster speed due to new memory access patterns, and more flexibility due to a more efficient interconnection mechanism of processors, respectively.

The paper is organized as follows. In Section 2, we revisit and summarize eigenanalysis, Jacobi algorithm and its cyclic version, and its use for parallel architectures via the chess tournament algorithm. In Section 3, we highlight single- and multi-threaded CPU implementations of the cyclic Jacobi algorithm in order to define a benchmark, especially for the GPU performance. In Section 4, we discuss the GPU implementation of the parallel Jacobi algorithm coded in CUDA™ [26]. We emphasize on memory access patterns for performance improvement. Then, we propose three novel memory access methods exploiting the symmetry of the input matrix, and more intelligent use of shared memory among GPU threads. FPGA implementation of the parallel Jacobi algorithm is explained in Section 5 complemented with a brief discussion on prior work reported in the literature. Then, we show the elements of the design from bottom to top for the FPGA design including CORDIC algorithm [9,4], processor design, systolic array [19] employed to interconnect the processors, control and synchronization with the finite state machines, and the number representations. A deterministic analysis of the computation time in clock cycles is also provided in this section. Finally, in Section 6, we display the performance results of CPU, GPU, and FPGA implementations. This study provides a comparison between CPU and GPU, and between GPU and FPGA implementations of eigenanalysis. The concluding remarks and future work are summarized in the last section of the paper.

2. Preliminaries

2.1. Notation

Let $\mathbf{z}$ be an $N \times 1$ column vector, and $\bar{\mathbf{z}}$ be a $1 \times N$ row vector. It is possible to define matrix $\mathbf{Z}$ of size $N \times N$ as

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

where $\mathbf{z}_i$ and $\bar{\mathbf{z}}_i$ are the ith 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

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

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

where $p^{(m)}$ and $q^{(m)}$ are the integers that define the sub-matrix of the data matrix to be rotated in the Jacobi algorithm assigned to the mth processing unit.

2.2. Eigenanalysis and Jacobi algorithm

Let $\mathbf{A}$ be a symmetric matrix of size $N \times N$, i.e., $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$ and $\left[A_{ij}\right] = \left[A_{ji}\right]$ where A_{ij} is the element located on ith row and jth column of $\mathbf{A}$. Eigenanalysis of matrix $\mathbf{A}$ is expressed as [2]

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

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

$$\mathbf{\Phi} = \begin{bmatrix} \boldsymbol{\phi}_1 & \boldsymbol{\phi}_2 & \cdots & \boldsymbol{\phi}_N \end{bmatrix},$$

and $\boldsymbol{\phi}_i$ is an $N \times 1$ eigenvector corresponding to the ith eigenvalue, λ_i. Jacobi algorithm provides an approximated numerical solution to Eq. (1) by iteratively reducing the metric defined based on the off-diagonal elements [12]

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

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

$$\mathbf{A}^{(k+1)} = \mathbf{J}^{\mathrm{T}}(p, q, \theta) \mathbf{A}^{(k)} \mathbf{J}(p, q, \theta), \tag{2}$$

where $1 \le p < q \le N$. 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 Eq. (2) are repeated until $\mathrm{off}(\mathbf{A}) < \epsilon$ where ϵ is a predefined threshold value that stops the iteration. After sufficient number of rotations, matrix $\mathbf{A}$ gets closer to matrix Λ, and the successive multiplications of $\mathbf{J}$ lead us to an approximation of the eigenmatrix $\mathbf{\Phi}$ expressed as [12]

$$\mathbf{\Phi} \simeq \mathbf{J}(p_1, q_1, \theta_1) \mathbf{J}(p_2, q_2, \theta_2) \cdots \mathbf{J}(p_L, q_L, \theta_L), \tag{3}$$

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

$$\begin{bmatrix} A_{pp}^{(k+1)} & A_{pq}^{(k+1)} \\ A_{qp}^{(k+1)} & A_{qq}^{(k+1)} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} A_{pp}^{(k)} & A_{pq}^{(k)} \\ A_{qp}^{(k)} & A_{qq}^{(k)} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \tag{4}$$

where $A_{ij}^{(k)}$ and $A_{ij}^{(k+1)}$ are elements of $\mathbf{A}^{(k)}$ and $\mathbf{A}^{(k+1)}$ located on the ith row and jth column, respectively. Eq. (4) diagonalizes matrix $\mathbf{A}$ after sufficient number of iterations. It follows from Eq. (4) that

$$A_{qp}^{(k+1)} = c\left(sA_{pp}^{(k)} + cA_{qp}^{(k)}\right) - s\left(sA_{pq}^{(k)} + cA_{qq}^{(k)}\right). \tag{5}$$

Rotation angle θ is found by setting $A_{pq}^{(k+1)} = A_{qp}^{(k+1)} = 0$ in Eq. (5). Since matrix $\mathbf{A}$ is symmetric, i.e., $A_{pq} = A_{qp}$, it follows from Eq. (5) that

$$cs\left(A_{pp}^{(k)} - A_{qq}^{(k)}\right) + A_{pq}^{(k)}\left(c^2 - s^2\right) = 0. \tag{6}$$

Using trivial trigonometric identities Eq. (6) can be rewritten as follows

$$\theta = 0.5 \tan^{-1}\left[2A_{pq}^{(k)} / \left(A_{qq}^{(k)} - A_{pp}^{(k)}\right)\right]. \tag{7}$$

Note that since p and q define the rotation matrix $\mathbf{J}(p, q, \theta)$ through Eq. (4) we can drop the angle, θ, and refer it as $\mathbf{J}(p, q)$. Originally, p and q in Eq. (2) are chosen such that $\left|A_{pq}\right| = \max_{i \neq j}\left|A_{ij}\right|$ [12]. However, searching for the maximum value at each iteration is not preferred due to computational cost concerns. The most straightforward modification to the classical method is the cyclic Jacobi algorithm that serially cycles through the data matrix by an ordered fashion, i.e., $(p^{(m)}, q^{(m)}) = \{(1, 2), (1, 3), (1, 4), \ldots, (2, 3), \ldots\}$ where $m = 1, 2, \ldots, N/2$. In Section 2.3, we will demonstrate that it is straightforward to implement the algorithm on a computing device with $N/2$ parallel processing units due to the sparsity of the rotation matrix $\mathbf{J}(p, q)$.

2.3. Parallel Jacobi algorithm

Jacobi rotation matrix is sparse. Thus, it is possible to perform rotations of Eq. (2) 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 [24,21]. 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 in one sweep are defined as

$$\begin{bmatrix} p^{(1)} & p^{(2)} \\ q^{(1)} & q^{(2)} \end{bmatrix} : \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}. \tag{8}$$

Note that $p^{(2)}$ and $q^{(2)}$ are interchanged in the last step since the condition $p < q$ must hold [12]. Moreover, interchanging is necessary after the step $N/2+1$. For $N = 4$ we have $N-1 = N/2+1 = 3$. Thus, the interchange needs to be done only in the last step.

Ensuring that p and q pairs are unique for each processing unit that handles only one of the two problems arising in the parallel implementation. The second problem is due to the overlap of matrix elements in the operations given in Eq. (2). Since the multiplication $\mathbf{J}(p, q)^{\mathrm{T}}\mathbf{A}$ in Eq. (2) would update the pth and qth rows of matrix $\mathbf{A}$, and multiplication $\mathbf{AJ}(p, q)$ in Eq. (2) would update the pth and qth columns of matrix $\mathbf{A}$, it would be problematic to implement Eq. (2) in parallel form without proper synchronization. Introducing an intermediary matrix $\mathbf{X}$ of size $N \times N$ [28] in the computational process is a popular solution for this concern. First operation is performed by the mth processing unit multiplying the $p^{(m)}$th and $q^{(m)}$th rows of $\mathbf{A}$ with the transpose of Jacobi sub-rotation matrix expressed as

$$\overline{\mathbf{X}}^{(m)} \leftarrow \left[\mathbf{J}^{(m)}\right]^{\mathrm{T}} \overline{\mathbf{A}}^{(m)}. \tag{9}$$

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

$$\mathbf{A}^{(m)} \leftarrow \mathbf{X}^{(m)}\mathbf{J}^{(m)}. \tag{10}$$

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

$$\mathbf{\Phi}^{(m)} \leftarrow \mathbf{\Phi}^{(m)}\mathbf{J}^{(m)}, \tag{11}$$

at the same time with Eq. (10) since $\mathbf{J}^{(m)}$ is already available.

3. CPU

Cyclic and parallel Jacobi algorithm with chess tournament, discussed in the previous section, are implemented in standard C language and coded to perform on a single thread and on multiple threads, respectively. For the multi-threaded implementation, POSIX threads library [7] is used. Proper synchronization via condition variables is performed such that there is no racing condition in the calculation of Eqs. (9), (10), and (11). There are two phases at each sweep. In the first phase, $N/2$ threads are spawned that calculate Eq. (9) individually. After all the threads are done, the second phase starts. In this phase, another group of $N/2$ threads are spawned to calculate Eqs. (10) and (11). Any calculation result of the first phase, that is re-used in the second phase, is kept in the system memory in order to improve performance. Note that both CPU implementations are performed in order to have a benchmark for the GPU and FPGA results. Performance comparisons of GPU and FPGA computing for the task at hand are presented in Section 6.

4. GPU

In this section, we discuss the graphics processing unit (GPU) implementation of the parallel Jacobi algorithm. In GPU implementation, two kernel calls; one for Eq. (9), and one for Eqs. (10) and (11), are used for a step in a sweep of the parallel Jacobi algorithm. Second kernel call must wait for the first one to complete its task as handled by the 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). The global synchronization is realized through CPU (host synchronization).

4.1. Memory access in GPU computing

Memory access time (reaching out to a memory location for reading or writing data) is an important limiting factor of the-state-of-the-art GPU computing technologies. Even the GPUs providing L1 and L2 caches to the main memory such as the ones with Fermi™ [25] architecture from NVIDIA™ suffer from performance degradations when the memory access pattern is unstructured and/or non-coalesced [26,11]. Memory coalescing, that is refining the memory access pattern such that the hardware can make combined requests from DRAM, should be employed whenever applicable.

4.2. 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 an array as *row-major* or *column-major* where the elements are linearized based on their rows and columns, respectively. For instance, let $\mathbf{Z}$ be a 2×2 matrix. Row-major and column-major arrays for storing the matrix $\mathbf{Z}$ are given as $\mathbf{z}_R = \begin{bmatrix} Z_{11} & Z_{12} & Z_{21} & Z_{22} \end{bmatrix}$and $\mathbf{z}_C = \begin{bmatrix} Z_{11} & Z_{12} & Z_{21} & Z_{22} \end{bmatrix}^{\mathrm{T}}$, respectively. We implement the parallel Jacobi algorithm in GPU using row-major arrays to store $\mathbf{A}$, $\mathbf{X}$, and $\mathbf{V}$ in Eqs. (9), (10), and (11) 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 Eq. (9). However, both reading and writing operations lead to a non-coalesced access in Eqs. (10) and (11).

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 Eq. (9), and when writing to matrix $\mathbf{A}$ in Eq. (10). Other than those two cases, all access patterns in MA become coalesced that helps the GPU to access its DRAM more efficiently and compute faster. In Section 6, 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 to perform the tasks of Eq. (10).

4.3. Symmetric access method

Since $\mathbf{A}$ is symmetric, its ith row is always identical to its ith column. Therefore, a processing unit can update the pth and qth rows of **A** in Eq. (10) instead of updating the pth and qth columns as given Eq. (10). In other words, modifying Eq. (10) as

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

leads us to the same solution. Explicitly, after every processing unit completes the update given in Eq. (12), matrix **A** is updated in the same way as it would be updated with Eq. (10) since $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$. However, this modification ensures the coalesced access when writing into matrix **A** that improves the computational efficiency. We name this access method as "the Symmetric Access (SA)" and show its performance superiority over MA in Section 6. Note that even with SA, there is still one non-coalesced access in Eq. (9) when writing into matrix **X**. The trick we used in SA for matrix **A** cannot be applied directly to **X** since it is not symmetrical. Nevertheless, we introduce a new method in the next subsection that also provides coalesced access to the memory locations reserved for **X**.

4.4. *Maximum-coalesced access method*

It is possible to ensure all memory access of reading and writing operations in the parallel Jacobi algorithm are coalesced by changing the nature of the update procedure in implementing Eq. (9). Note that the proposed modification makes use of an inherent feature of the chess tournament algorithm in Eq. (8). We call this method "the Maximum-Coalesced Access (MCA)". In MCA, the update given in Eq. (9) is modified as follows

$$\mathbf{X}^{(m)} \leftarrow \mathbf{K}\left[\overline{\mathbf{A}}^{(m)}\right]^{\mathrm{T}}, \tag{13}$$

where **K** is an $N \times N$ matrix defined as

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

$J_{ij}^{(m)}$ is the element of Jacobi sub-rotation matrix and $m = 1, 2, \dots, N/2$. Note that **K** is constant for all processing units in a sweep. In MCA, mth processing unit updates the $p^{(m)}$th and $q^{(m)}$th columns of **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 $i = 1, 2, \dots, N$ and $f(\cdot)$ is a mapping defined as $f(x) = g(x) \cup g^{-1}(x)$, and $g(x)$ is a mapping from set $\left\{p^{(m)}\right\}$ to set $\left\{q^{(m)}\right\}$, i.e., $g : p^{(m)} \rightarrow q^{(m)}$. 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 due to the chess tournament algorithm, it is feasible to realize the update equation given in Eq. (14). It is worth noting that we implicitly exploit the inherent symmetry of matrix **A** again in MCA, i.e., $\left[\overline{\mathbf{A}}^{(m)}\right]^{\mathrm{T}}$ in Eq. (13) accounts for accessing the $p^{(m)}$th and $q^{(m)}$th rows of **A** (in accordance with its row-major array data structure) and using its transpose such that we have a matrix of size $N \times 2$.

4.5. *One step parallel Jacobi algorithm*

It was discussed in Section 2 that multiplying the data matrix with $\mathbf{J}^{\mathrm{T}}(p, q)$ from left, and with $\mathbf{J}(p, q)$ from right, and overwriting the result onto itself, updates the rows and columns of data matrix, respectively. Therefore, in any parallel implementation of the Jacobi algorithm, these two multiplications must be performed in two kernels as given in Eqs. (9) and (10) with proper synchronization among the assigned processing units. However, thanks to MCA discussed earlier, it is possible to perform these two updates in only one kernel. By substituting Eq. (13) into Eq. (12) we obtain

$$\overline{\mathbf{A}}^{(m)} \leftarrow \left[\mathbf{K}\left[\overline{\mathbf{A}}^{(m)}\right]^{\mathrm{T}}\mathbf{J}^{(m)}\right]^{\mathrm{T}}. \tag{15}$$

Note that Eq. (15) updates only the $p^{(m)}$th row and $q^{(m)}$th column of **A**, and it does not need intermediary matrix **X**. We call the algorithm implementing Eq. (15) as "One Step Parallel Jacobi Algorithm (OSPJ)". OSPJ delivers the best performance among other GPU implementations as reported in Section 6.

5. FPGA

In this section, we discuss field-programmable gate array (FPGA) design of the parallel Jacobi algorithm. Several very large-scale integration (VLSI) [30,6,13] and FPGA [16,1,5,20] implementations of parallel Jacobi algorithm have been reported in the literature. Almost all of the hardware designs rely on processors that use coordinate rotation digital computer (CORDIC) [9,4,8,14] which is an iterative algorithm to rotate vectors. Either a large number of processors are interconnected via systolic array [18,30] or only two processors are used in a cyclic fashion [20,5]. The former is designed for massive throughput and low-latency since it exploits the parallelism inherent in Jacobi algorithm. However, matrices with small dimensions can be supported in general as it requires significant amount of area. Similarly, the latter has low throughput and high latency, but it is suitable for small chips. Some of the designs use only the upper-triangle of the systolic array that provides just the eigenvalues not the eigenvectors. A relatively recent study discusses three different methods for implementing Jacobi algorithm in FPGA. Namely, they are the exact, window, and approximate methods with an upper-triangle systolic array [20]. Windowed method exploits the fact that it is highly possible for annihilated off-diagonal element to be reversed in the next step. Therefore, as opposed the exact method, a variable number of iterations are employed in the windowed method [13].

Our main objective in this paper is to report a performance comparison for parallel implementations of the Jacobi algorithm on different high performance computing technologies. Therefore, we implement the exact method with a full systolic array. However, we also improved the design such that the paths in the systolic array can be reconfigured in order to support variable sizes for the input matrix.

5.1. *CORDIC algorithm*

Coordinate rotation digital computer (CORDIC) algorithm, also known as the digit-by-digit method or Volder's algorithm, is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions or to rotate vectors [9,4]. It is commonly used when no hardware multiplier is available. Let $\mathbf{v} = \begin{bmatrix} x & y \end{bmatrix}^{\mathrm{T}}$ be a real vector, **J** be the 2×2 Jacobi rotation matrix, and $\mathbf{v}' = \begin{bmatrix} x' & y' \end{bmatrix}^{\mathrm{T}}$ be the rotated **v** by an angle of θ as defined

$$\begin{gathered} \mathbf{v}' = \mathbf{J}\mathbf{v} \\ \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{gathered} \tag{16}$$

Using the trigonometric identities we express the rotated vector $\mathbf{v}'$ as follows

$$\mathbf{v}' = \left(1 + \tan^2\theta\right)^{-1/2} \begin{bmatrix} x - y\tan\theta & x\tan\theta + y \end{bmatrix}^{\mathrm{T}}. \tag{17}$$

In each iteration of the CORDIC, i, rotation angle, θ, is restricted to the binary grid value $\theta = \tan^{-1}\left(\pm 2^{-i}\right)$. Therefore, computations of $y\tan\theta$ and $x\tan\theta$ are implemented by simple binary shift operations. In order to get closer to the target vector $\mathbf{v}'$, CORDIC performs the following operations in each iteration

$$\mathbf{v}'_{i+1} = K_i \left[x_i - y_i d_i 2^{-i} \quad x_i d_i 2^{-i} + y_i\right]^{\mathrm{T}} \tag{18}$$

where $K_i = \left(1 + 2^{-2i}\right)^{-1/2}$, $d_i = \mathrm{sgn}\left(\varphi_i\right)$, and $\varphi_{i+1} = \varphi_i - d_i \tan^{-1}(2^{-i})$ is the residual angle with $\varphi_0 = \theta$, $x_0 = 1$, and $y_0 = 0$. Note that the calculation of φ_i requires *elementary angles*, $\tan^{-1}(2^{-i})$, that can be stored in a look-up table (LUT). Moreover, calculation of K_i requires a square-root operator. In practice the approximate of its asymptotic value 0.60725 is used. Alternatively, values of K_i can be pre-calculated and be stored in a LUT. The operations of CORDIC discussed so far are called *rotation mode*. It is also possible to measure the angle of a vector via the *vectoring mode* of CORDIC [9,4].

5.2. Processor design

There are two types of processors used in the FPGA design, namely diagonal and off-diagonal processors. Both processors use rotation mode of the CORDIC algorithm during the row and column updates of the Jacobi algorithm. Additionally, diagonal processors calculate the angle θ, required to set the off-diagonal elements of the 2×2 sub-matrices to zero as given in Eq. (7) and distribute this information to the off-diagonal processors located on the same row and column. Note that the value of θ in Eq. (7) is nothing else but half the angle of the 2×1 vector $\left[\left(A_{qq} - A_{pp}\right) \quad 2A_{pq}\right]^{\mathrm{T}}$. Hence, it is determined via hardware by using the vectoring mode of the CORDIC algorithm. Note that for the case of $A_{qq} = A_{pp}$, there is no need for CORDIC since the angle is equal to 45° when $A_{qq} > 0$ and -45° when $A_{qq} \le 0$. Once the angle is determined, all the processors perform the row, column, and eigenvector update operations given in Eqs. (9), (10), and (11), respectively. Row update operation given in Eq. (9) is restated as follows

$$\begin{bmatrix} X_{p,n} \\ X_{q,n} \end{bmatrix} \leftarrow \begin{bmatrix} A_{p,n}\cos\theta - A_{q,n}\sin\theta \\ A_{p,n}\sin\theta + A_{q,n}\cos\theta \end{bmatrix}, \tag{19}$$

where $X_{p,n}$ and $A_{p,n}$ are the elements located on the pth row and nth column of matrices $\mathbf{X}$ and $\mathbf{A}$, respectively, and $1 \le n \le N$. Note that Eq. (19) is actually rotation of the 2×1 vector $\left[A_{p,n} \quad A_{q,n}\right]^{\mathrm{T}}$ that can be performed in parallel by N CORDIC processors. Similarly, for the column-update operation given in Eq. (10) we have

$$\begin{bmatrix} A_{n,p} \\ A_{n,q} \end{bmatrix}^{\mathrm{T}} \leftarrow \begin{bmatrix} X_{n,p}\cos\theta - X_{n,q}\sin\theta \\ X_{n,p}\sin\theta + X_{n,q}\cos\theta \end{bmatrix}^{T}, \tag{20}$$

that is the rotation of the 2×1 vector $\left[X_{n,p} \quad X_{n,q}\right]^{\mathrm{T}}$. Similarly, after the column update operation is completed, processors update the eigenvectors according to Eq. (11).

5.3. CORDIC processor fabric (CPF)

Systolic array, a special parallel computing architecture where each processing unit computes and stores the data separately [19], is used in our design to interconnect the diagonal and off-diagonal processors described in the previous subsection. Instead of broadcasting the data to each processor, systolic array ensures that the data travels around the network by an exchange among neighbor processors that significantly reduces the data transfer complexity. Note that FPGA implementation does not require the matrix $\mathbf{X}$ given in (9) as it distributes the data in a $N/2 \times N/2$ systolic array. However, it operates in three different states that perform the operations described in (9), (10), and (11) that will be explained later. Fig. 1 displays the systolic array of CORDIC processors for the case of $N = 8$. Systolic array is the backbone of one of the main components of our design, i.e., CORDIC processor fabric (CPF), displayed in Fig. 2.

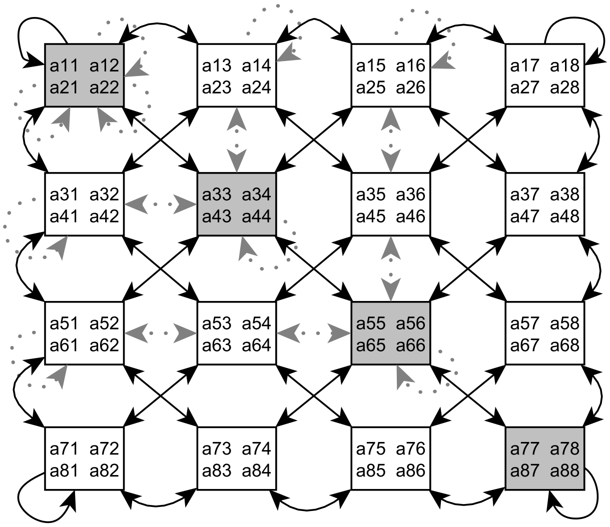


Fig. 1. Systolic array used for interconnecting diagonal and off-diagonal processors. The arrows indicate the direction of data exchanges among neighboring processors. Dashed arrows are activated when the matrix size N is smaller than the maximum size supported. Each processor stores and operates on four elements of the matrix, depicted with *aij*.

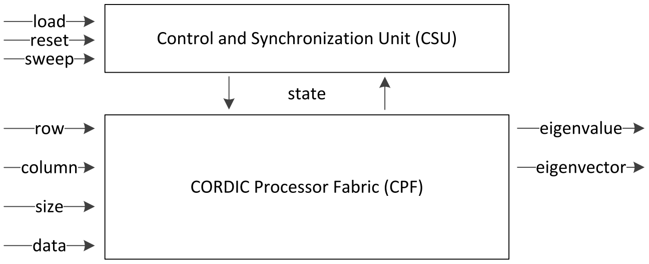

load
reset
sweep
Control and Synchronization Unit (CSU)
state
row
column
size
data
CORDIC Processor Fabric (CPF)
eigenvalue
eigenvector

Fig. 2. Block diagram of the Jacobi based FPGA eigensolver.

Note that Systolic array based eigensolvers reported in the literature (see [20] and references therein) support only matrices of a pre-determined size. However, our improved design supports matrices with different sizes as long as the matrix size, N, is even and it is less than the maximum matrix size supported. This is made possible by turning on and off the communication between the processors and re-routing the transfer based on N in the processor level. The faded arrows in Fig. 1 indicate the possible re-routing of the pipes within the array. The case when N is odd can also be supported by padding zeros to the last row and last column of the data.

5.4. Control and synchronization unit (CSU)

Operations in our design are orchestrated by the control and synchronization unit (CSU) as displayed in Fig. 2. CSU comprises of two finite state machines (FSM). Namely, they are the main FSM (MFSM) and solver FSM (SFSM). Virtually at any state CSU communicates with CPF via signaling paths and broadcasts a change in state to the processors in CPF, accordingly. CSU receives processor state information from CPF. It is also hooked to the I/O bus and receives the signals *load*, *reset*, and *sweep*. MFSM and SFSM are explained next.

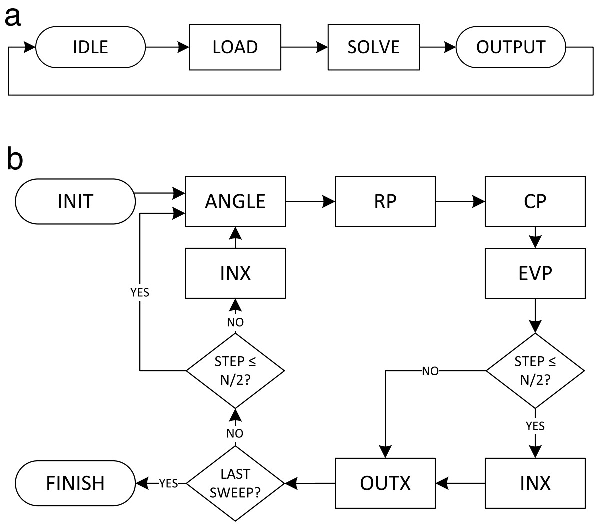


Fig. 3. (a) Main, and (b) solver finite state machines used in the CSU.

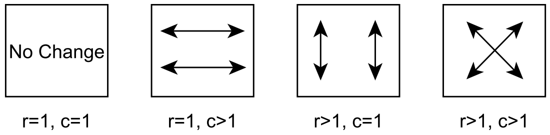
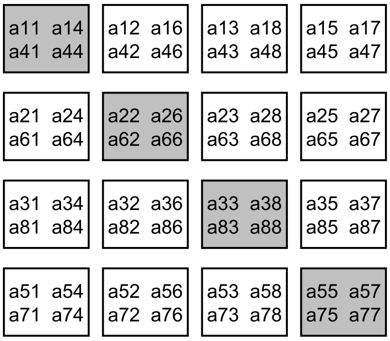


Fig. 4. Inner-exchange scheme of the processors in CPF. Row and column indices are represented with r and c, respectively where $1 \le r, c \le N/2$.



a11 a14 a41 a44	a12 a16 a42 a46	a13 a18 a43 a48	a15 a17 a45 a47
a21 a24 a61 a64	a22 a26 a62 a66	a23 a28 a63 a68	a25 a27 a65 a67
a31 a34 a81 a84	a32 a36 a82 a86	a33 a38 a83 a88	a35 a37 a85 a87
a51 a54 a71 a74	a52 a56 a72 a76	a53 a58 a73 a78	a55 a57 a75 a77

Fig. 5. Data placement in the systolic array of processors after the first OUTX state of the systolic array in Fig. 1 is completed.

5.4.1. Main FSM (MFSM)

MFSM is responsible for the overall operation of the design. It has four states named as IDLE, LOAD, SOLVE, and OUTPUT, and displayed in Fig. 3(a). When the binary *load* signal goes to HIGH from LOW, MFSM switches from INIT to LOAD state. In the LOAD state, each element of the input array of size $N^2 \times 1$ representing a matrix of size $N \times N$ is received through *data* bus serially that is fed into CPF. Along with the data, its corresponding row and column information through *row* and *column* signals are also fed into the processors. Each processor receiving the broadcast data decides on to keep or to ignore it. The *size* signal, as the name suggests, contains the size of the matrix, that is used to enable or disable processors to support various matrix sizes as discussed in the previous subsection. When all N^2 elements are fed into CPF, MFSM switches to SOLVE state that starts SFSM (described next) and waits for it to finish. In its last state, OUTPUT, MFSM commands processors in CPF to deliver eigenvalues and eigenvectors through their corresponding buses. Finally, MFSM switches back to IDLE state.

5.4.2. Solver FSM (SFSM)

SFSM is responsible for controlling the processors in CPF such that the Jacobi rotations defined in Eqs. (19) and (20) are performed. It has eight states named as INIT, ANGLE, RP, CP, EVP, INX, OUTX, and FINISH as displayed in Fig. 3(b). Upon initializing the processors SFSM switches to ANGLE state. In this state, rotation angle defined in Eq. (7) is calculated by diagonal processors by the vectoring mode of the CORDIC algorithm. Calculated angle is then distributed to the off-diagonal processors. Note that ith diagonal processor, distributes the angle to the off-diagonal processors located on the ith row and ith column via separate buses.

Next, SFSM switches to RP, CP, and EVP states, that stand for row-processing, column-processing, and eigenvector-processing, respectively. In RP, and CP, rotations given in Eqs. (19) and (20) are performed via the rotation mode of the CORDIC algorithm. Number of iterations is limited, i.e., $i \le M$ in Eq. (18). However, since the angle is not the same for different processors, some arrive to zero (or negligible) residual angle, φ. Therefore, additional precaution in SFSM is taken such that state does not change unless all the processors finish rotation. In EVP state, eigenvectors are also calculated.

Once all the rotations are completed, SFSM proceeds to INX and OUTX states that stand for inner- and outer-exchange, respectively. Data residing in each processor is exchanged, first within the processor itself, then among neighboring processors, in INX and OUTX states, respectively, such that all the processors are ready for the next step in accordance with the chess tournament algorithm described in Section 2. Note that since the first index is kept static in the chess tournament, processors located on the top row and far left column of the CPF perform inner exchange differently than the rest. Details of inner exchange scheme are displayed in Fig. 4. The data exchanges in the arrow direction inside each processor. Outer-exchange scheme is shown in Fig. 1. The arrows indicate the direction of data exchanges among neighboring processors. Dashed arrows are activated when the matrix size N is smaller than the maximum size supported. The distribution of the data in CPF after the first inner and outer exchanges for $N = 8$ case is depicted in Fig. 5, that is inline with the p and q indices for the second step of a sweep as discussed in Section 2. It is worth restating that after the step $N/2$ in a sweep, we have $q > p$ in the processors. This is resolved in our design by skipping INX state and performing it before the ANGLE state. Details are displayed in Fig. 3(b). Finally, when total number of sweeps defined by user via *sweep* bus is reached, SFSM proceeds to FINISH state that results in MFSM to switch to the OUTPUT state.

5.5. Computation time

It is possible to derive computation time for the worst-case scenario in number of clock cycles for our design. A similar analysis is done in [20] for their design. At each step of a sweep, a processor in CPF uses 14 cycles in operations necessary for initialization and inner- and outer-exchange of the data. Moreover, CORDIC algorithm is used for 4 times within a processor at each step, for the calculation of rotation angle, row and column updates, and eigenvector calculation. In the worst-case scenario, each run of the CORDIC algorithm takes the maximum number of iterations (cycles), i.e., M, to be completed. Note that the number of cycles depends on the angle assigned to the processor. Given that there are $N-1$ steps in a sweep, number of cycles required for a complete sweep is $C_s = (N-1)(14+4M)$. As discussed earlier, N^2 cycles are required for both feeding in and taking out the data from CPF. Therefore, total number of cycles required for the design to provide the result is

$$C_t = 2N^2 + SC_s = 2N^2 + S(N-1)(14+4M), \tag{21}$$

where $S > 0$ as the pre-determined number of sweeps. Note that parameters M and S are directly related to both computation time and accuracy of the result.

5.6. Number representation

There are trade-offs between precision, resources, and speed in using floating- and fixed-point representations. Similarly, rounding circuits provide better accuracy but decreased speed (around 27% more [20]) with increase in the occupied area, compared to truncation circuits. Fixed-point representation and truncation circuits are employed in our design. In fixed-point representation, a number is scaled by a pre-determined factor, usually a power of ten or two. For instance, if scaling factor is chosen to be 10^3, then real number 20.478 can be represented as $20\,478/10^3$. The VHDL library employed in our design uses a power of two scaling factor. If the word length is B bits and the decimal part is stored after the Pth MSB bit, then we have the decimal number $D = \sum_{i=0}^{B} b_i 2^{i-B+P}$ where b_i is the ith LSB with $0 \le i < B$. Note that, precision and the area required to realize the circuit highly depends on the choice of B and P.

6. Performance evaluations

In this section, we provide the performance evaluations for the eigenanalysis implementations discussed in the paper. Randomly generated 20 different matrices were used in the experiments. We complete the section with a comparison in terms of speed and implementation error of the competing high performance computing technologies.

6.1. CPU and GPU

We provide the performance of the CPU and GPU implementations discussed in the paper in terms of speed. We performed our tests on a six core (a total of twelve cores with hyper-threading) Intel® Core™ i7-3960X CPU @ 3.30 GHz with 32 GB RAM machine running on Linux. The GPU used in the tests is an NVIDIA GeForce™ GTX 580 built with Fermi™ architecture [25] with 512 CUDA™ Cores, and 1536 MB global memory. Source codes are compiled with CUDA™ Compiler Driver v5.0. Note that tools and devices used in the tests are faster and better than the ones authors used in their earlier work [34,32]. 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 single- and multi-threaded CPU as well as various GPU implementations with different memory access patterns (TA, MA, SA, MCA, and OSPJ) as a function of input matrix size are tabulated in Table 1, and also displayed in Fig. 6. Multi-threaded CPU implementation can reach the performance of the single-threaded implementation when $N = 1024$. Computation time for $N = 2048$ are 2882.2 and 729.3 s for single- and multi-threaded CPU implementations, respectively. These results are inline with the findings reported in [28], and shows conclusively that overhead of thread creation and synchronization disqualifies CPU from being a feasible environment for large dense matrices.

For the GPU implementations, as expected, speed is the best for the OSPJ. For $N = 1024$, performance improvement of OSPJ over TA, MA, SA, and MCA are 81.6%, 73.2%, 56.7%, and 30.9%, respectively. Speed-up of TA, MA, SA, MCA, and OSPJ on GPU implementations over cyclic Jacobi algorithm on single-threaded CPU versus the input matrix size are shown in Fig. 7. For $N = 1024$, the speed-up of TA, MA, SA, MCA, and OSPJ on GPU over single-threaded CPU are 12.6×, 18.3×, 29.7×, 47.4×, and 68.5× respectively.

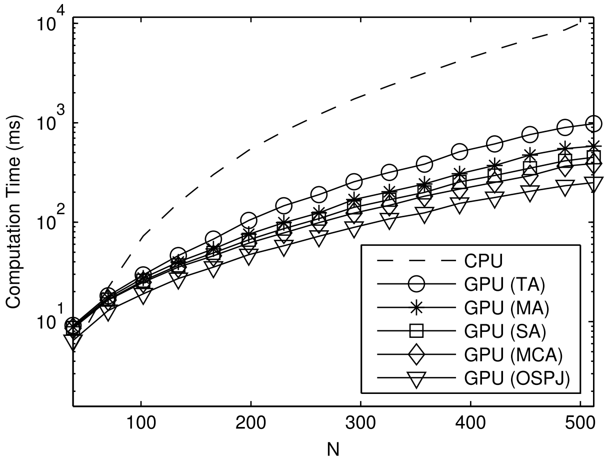


Fig. 6. Computation times of cyclic Jacobi algorithm, in milliseconds, implemented on CPU; TA, MA, SA, MCA, and OSPJ on GPU for various matrix sizes, N.

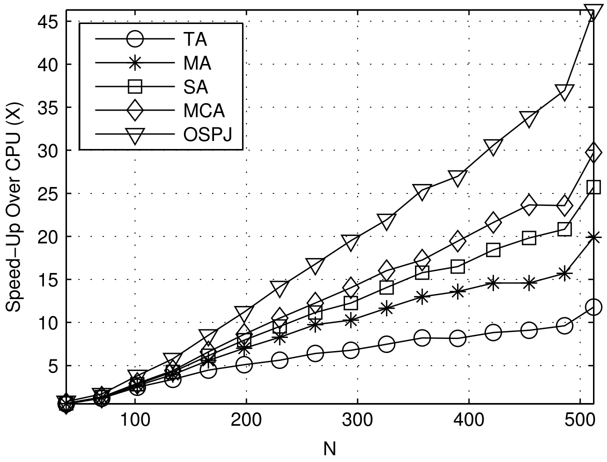


Fig. 7. Speed-up of TA, MA, SA, MCA, and OSPJ implemented on GPU over cyclic Jacobi algorithm implemented on CPU for various matrix sizes, N.

Table 1
Computation times, in milliseconds, for single- and multi-threaded CPU (first and seconds rows) and for GPU implementations with various memory access patterns (third to last rows) and the input matrix size, N. Times required to copy data from and to the system memory are taken into account for GPU experiments.

	$N = 8$	$N = 32$	$N = 128$	$N = 512$	$N = 1024$
CPU	0.05	2.30	136.10	11,570.31	110,761.36
CPU (MT)	6.34	64.48	1252.13	22,845.91	112,226.61
GPU (TA)	1.68	7.72	40.40	980.70	8,771.24
GPU (MA)	1.69	7.46	35.01	581.91	6,037.52
GPU (SA)	1.70	7.28	32.92	449.65	3,735.22
GPU (MCA)	1.72	7.15	31.63	388.88	2,338.69
GPU (OSPJ)	1.41	5.42	24.27	249.83	1,616.85

6.2. FPGA

Eigensolver system discussed in the previous section is implemented in VHDL with $M = 24$ in Eq. (21), and for $4 \le B \le 32$ with $P = B/2$. VHDL is compiled for various target devices, namely Cyclone II and Stratix IV FPGAs from Altera; and Artix7, Kintex7, Virtex6, and Zynq FPGAs from Xilinx. Compilers used for Altera and Xilinx devices are Quartus II v12.0 and Xilinx ISE v14.2, respectively. Resource usage and $f_{\max}$, the maximum frequency at which the circuit can be clocked, provided by the compilers, are reported. Note that both metrics depend on the choice of B, however $f_{\max}$ should not significantly change with the matrix size, N, since the system operates in parallel by design.

Table 2
(a) Maximum speed (MHz) and required resources for (a) Altera Cyclone II EP2C70F896C6 with a total of 66,416 LEs, (b) Altera Stratix IV EP4SGX530KF43C2 with a total of 424,960 ALUTs as a function of maximum matrix size supported, $N_{\max}$, and (c) four different devices from Xilinx for $N_{\max} = 4$ and $B = 16$.

$N_{\max}$	8 bits		16 bits		32 bits	
	$f_{\max}$	LEs	$f_{\max}$	LEs	$f_{\max}$	LEs
4	57.45	5,118	49.12	8,601	34.15	17,467
8	57.52	16,605	48.41	29,998		
12	55.65	35,469				

(a)

$N_{\max}$	8 bits		16 bits		32 bits	
	$f_{\max}$	ALUTs	$f_{\max}$	ALUTs	$f_{\max}$	ALUTs
4	92.69	4,668	86.88	7,640	67.13	14,097
8	91.27	14,775	81.94	25,641	68.96	50,777
12	87.13	31,461	79.71	55,279	59.36	134,799
16	89.12	54,600	78.14	100,628	56.85	334,252
20	88.87	86,799	74.52	191,606		
24	75.23	128,034	81.05	308,266		
28	68.20	189,293	73.01	399,847		
32	85.59	259,525				

(b)

Device	$f_{\max}$	Occupied slices	Available slices
Artix7 200T	71.72	2205	33,650
Kintex7 160T	92.90	2444	25,350
Virtex6 75T	86.88	1922	11,640
Zynq 030	84.33	2531	19,650

(c)

Table 3
Computation time, in milliseconds, for CPU, GPU, and FPGA implementations of JA with various matrix sizes. Values in italics are estimated via (22).

	$N = 16$	$N = 64$	$N = 256$	$N = 512$	$N = 1024$
CPU	0.97	17.25	1,326.88	11,570.31	110,761.36
GPU	2.55	10.79	65.43	248.29	1,610.13
FPGA	0.17	*2.79*	*44.58*	*178.32*	*713.29*

Results for Altera's Cyclone II and Stratix IV are displayed in Table 2(a) and (b), respectively. Note that since these two devices use different technologies, hence different building blocks, unit of the resource usage is total number of logic elements (LE) and adaptive look-up table (ALUT) for Cyclone II and Stratix IV, respectively. Moreover, since Cyclone II is a smaller chip, maximum N that fits is 12, 8, and 4 for word-lengths 8, 16, and 32 bits, respectively (see Table 2(a)). On the other hand, $N = 32$ for $B = 8$ and $N = 16$ for $B = 32$ fit to Stratix IV. For both devices $f_{\max}$ results are similar. The best and worst $f_{\max}$ are 57.52 MHz and 34.14 MHz for Cyclone II, 92.69 MHz and 56.85 MHz for Stratix IV, respectively.

Results for Xilinx's Artix7, Kintex7, Virtex6, and Zynq are displayed in Table 2(c) with $B = 16$ and $N = 4$. Unit for resource-usage is *slice* for Xilinx devices. Therefore, it is not possible to make a fair comparison between Altera and Xilinx devices. However, number of slices displayed in Table 2(c) can be compared with the ones reported in [20]. Authors of [20] implemented an upper-triangle systolic array that calculates only the eigenvalues using Handel-C$^{\text{TM}}$ [22] on an older Xilinx device, namely XC2V6000-6 Virtex-II. It is observed that the design discussed in this paper uses less than half the resources reported in [20] although it uses a full systolic array and calculates both eigenvalues and eigenvectors. Reasons for the improvement might include the direct use of VHDL instead of utilizing a higher-level HDL like Handel-C$^{\text{TM}}$ and/or employing improved compilers and devices.

6.3. Performance comparison of GPU and FPGA designs

It is shown in Section 6.1 that OSPJ, discussed in Section 4.5, delivers the best performance among other GPU implementations. As it was discussed in Section 6.1, numbers are defined with float type as represented by 32 bits in the computer and GPU used for the tests. Hence, in order to be able to make a better comparison, FPGA design with $B = 32$ is chosen. Moreover, the worst-case number of cycles for FPGA given in Eq. (21) takes into account the cycles needed to feed-in the data to the systolic array, i.e., N^2 cycles. Similarly, time needed to copy data from and to the system memory are included in the GPU results displayed in Table 1. Note that, smarter I/O operation may be employed in both designs. For example, remote direct memory access (RDMA) for the GPU might be utilized. Similarly, data can be written to/read from a fast memory with a faster clock adjacent to the FPGA chip. Therefore, for comparison purposes, GPU test for the OSPJ design is re-performed such that only the time needed for computation is measured. Computation times required to perform $S = 6$ sweeps of JA run on different high performance computing devices are tabulated in Table 3 for various matrix sizes, N. Furthermore, FPGA results for $N > N_{\max}$ are estimated by using the rationale that it would take $(N/N_{\max})^2$ times more to solve a larger matrix by using the block JA [12]. More specifically, the worst-case scenario computation time for FPGA is calculated by using Eq. (21)

$$\widehat{t_{FPGA}} = \left(\frac{N}{N_{\max}}\right)^2 \frac{1}{f_{\max}} S \left(N_{\max} - 1\right) \left(14 + 4M\right), \tag{22}$$

where time required for I/O, i.e., $2N^2$, is omitted for a fair comparison with GPU. Parameters of Eq. (22) are of $M = 24$ and $f_{\max} = 56.85$ MHz as displayed in Table 2(b) for Stratix IV. It is clearly observed from Table 3 that FPGA is the superior technology among the ones considered in this paper for the implementation of eigenanalysis. This remark is convincingly substantiated with the realized superior performance of the FPGA implementations over the others. Moreover, FPGA devices are more efficient than GPUs with respect to power consumption. However, in contrast, GPU is a far better option in terms of the time required for development.

Since fixed-point representation is used in FPGA implementation, it causes computation error. The mean squared error (mse), ϵ, between the eigenvalues and eigenmatrices generated by CPU/GPU and FPGA, with fixed-point representation, for $B = 8$, $B = 16$, and $B = 32$ with $P = B/2$ are measured for comparison. The resulting mse for eigenvalues are $\epsilon_8^{\Lambda} = 0.0017$, $\epsilon_{16}^{\Lambda} = 0.0015$, $\epsilon_{32}^{\Lambda} = 0.0014$, respectively. Similarly, mse measurements for eigenmatrices are $\epsilon_8^{\Phi} = 0.4601$, $\epsilon_{16}^{\Phi} = 0.4141$, $\epsilon_{32}^{\Phi} = 0.4012$, respectively. It is noted that the measured mse values do not change significantly with respect to the size of the matrix for fixed wordlength. It is shown in [35] that the effect of the error for $B = 32$ and $P = B/2$ is trivial for eigenfiltering, and it depends on the application.

7. Conclusions

In this paper, we revisited the celebrated Jacobi algorithm for eigenanalysis of large dense symmetric matrices using the chess tournament algorithm and its parallel implementation on various HPC technologies, namely, CPU, GPU, and FPGA. It was shown that, even with multi-threaded implementations, CPU is a poor computing platform for this task. Overhead of thread creation and synchronization is significant for square matrices of size 1024 or smaller. For the GPU implementations, we highlighted the fact that memory is a limiting performance factor. Then, we introduced three novel implementations with more intelligent memory access patterns leading to drastic performance improvements. The best GPU design method proposed, OSPJ, is quantified to achieve 81.6% computational performance improvement over the traditional

GPU methods, and 68.5 times faster operation over the single-threaded CPU for a dense symmetric matrix of size $N = 1024$ under the same test conditions. For the FPGA implementation, we provided the details of the design and introduced an improved systolic array structure that can support various matrix sizes. The performance results for these three HPC technologies are reported in the paper. It is concluded that FPGA design offers the best computing performance for eigenanalysis of matrices using JA although GPU is a strong competitor requiring less development effort with superior scalability.

References

[1] A. Ahmedsaid, A. Amira, A. Bouridane, Improved SVD systolic array and implementation on FPGA, in: Proc. IEEE International Conference on Field-Programmable Technology, 2003, pp. 35–42.

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

[3] A.N. Akansu, M.U. Torun, Toeplitz approximation to empirical correlation matrix of asset returns: A signal processing perspective, IEEE J. Sel. Top. Sign. Proces. 6 (4) (2012) 319–326.

[4] R. Andraka, A survey of CORDIC algorithms for FPGA based computers, in: Proc. ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, 1998, pp. 191–200.

[5] I. Bravo, P. Jimenez, M. Mazo, J. Lazaro, G.A. , Implementation in FPGAs of Jacobi method to solve the eigenvalue and eigenvector problem, in: Proc. Int. Conf. Field Programmable Logic and Applications, 2006, pp. 1–4.

[6] R.P. Brent, F.T. Luk, C.V. Loan, Computation of the singular value decomposition using mesh-connected processors, J. VLSI Comput. Syst. 1 (1985) 242–270.

[7] D. Butenhof, Programming With POSIX Threads, in: Addison-Wesley Professional Computing Series, Prentice Hall, 1997.

[8] J.R. Cavallaro, F.T. Luk, CORDIC arithmetic for an SVD processor, J. Parallel Distrib. Comput. 5 (1988) 271–290.

[9] J.-M. Delosme, CORDIC algorithms: theory and extensions, Proc. SPIE 1152 (1989) 131–145.

[10] J. Demmel, K. Veselic, Jacobi's method is more accurate than QR, SIAM J. Matrix Anal. Appl. 13 (1992) 1204–1245.

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

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

[13] J. Gotze, S. Paul, M. Sauer, An efficient Jacobi-like algorithm for parallel eigenvalue computation, IEEE Trans. Comput. 42 (1993) 1058–1065.

[14] X. Hu, S.C. Bass, R.G. Harber, An efficient implementation of singular value decomposition rotation transformations with CORDIC processors, J. Parallel Distrib. Comput. 17 (4) (1993) 360–362.

[15] C.G.J. Jacobi, Über ein leichtes verfahren, die in der theorie der säkularstörungen vorkommenden gleichungen numerisch aufzulösen, Crelle's J. 30 (1846) 51–94.

[16] M. Kim, K. Ichige, H. Arai, Design of Jacobi EVD processor based on CORDIC for DOA estimation with MUSIC algorithm, Proc. PIMRC 1 (2002) 120–124.

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

[18] H.T. Kung, Why systolic architectures? Computer 15 (1982) 37–46.

[19] H.T. Kung, C.E. Leiserson, Systolic Arrays for (VLSI), CMU-CS, Carnegie-Mellon University, Department of Computer Science, 1978.

[20] Y. Liu, C.-S. Bouganis, P.Y.K. Cheung, Hardware efficient architectures for eigenvalue computation, Comput. Digit. Tech., IET 3 (2009) 72–84.

[21] F.T. Luk, H. Park, A proof of convergence for two parallel Jacobi SVD algorithms, IEEE Trans. Comput. 38 (6) (1989) 806–811.

[22] Mentor Graphics, DK Design Suite Datasheet (2010). URL http://www.mentor.com/products/fpga/handel-c/.

[23] U. Meyer-Baese, Digital Signal Processing with Field Programmable Gate Arrays, third ed., Springer Publishing Company, Incorporated, 2007.

[24] V. Novakovic, S. Singer, A GPU-based hyperbolic SVD algorithm, BIT Numer. Math. 51 (2011) 1009–1030.

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

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

[27] V. Pedroni, Circuit Design and Simulation with VHDL, MIT Press, 2010.

[28] G.S. Sachdev, V. Vanjani, M.W. Hall, Takagi factorization on GPU using CUDA, in: Proc. Symposium on Application Accelerators in High Performance Computing, Knoxville, Tennessee, 2010.

[29] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, first ed., Addison-Wesley Professional, 2010.

[30] R. Schreiber, Systolic arrays for eigenvalue computation, Proc. SPIE 341 (1982) 27–34.

[31] D.E. Thomas, P.R. Moorby, The Verilog Hardware Description Language, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[32] M.U. Torun, A.N. Akansu, A novel GPU implementation of eigenanalysis for risk management, in: Proc. IEEE 13th International Workshop on Signal Processing Advances in Wireless Communications, SPAWC, 2012, pp. 490–494.

[33] M.U. Torun, A.N. Akansu, M. Avellaneda, Portfolio risk in multiple frequencies, IEEE Signal Process. Mag. 28 (5) (2011) 61–71. Special Issue on Signal Processing for Financial Applications.

[34] M.U. Torun, O. Yilmaz, A.N. Akansu, Novel GPU implementation of Jacobi algorithm for Karhunen-Loève transform of dense matrices, in: Proc. IEEE 46th Annual Conference on Information Sciences and Systems, CISS, 2012, pp. 1–6.

[35] M.U. Torun, O. Yilmaz, A.N. Akansu, Fpga based eigenfiltering for real-time portfolio risk analysis, in: Acoustics, Speech and Signal Processing, ICASSP, 2013 IEEE International Conference on, 2013, pp. 8727–8731.

[36] S.R. Upadhyaya, Parallel approaches to machine learning-a comprehensive survey, J. Parallel Distrib. Comput. 73 (3) (2013) 284–292.

[37] P. Wittek, S. Darányi, Accelerating text mining workloads in a mapreduce-based distributed gpu environment, J. Parallel Distrib. Comput. 73 (2) (2013) 198–206.

Mustafa U. Torun received his B.S. and M.S. degrees from the Dokuz Eylul University, Izmir, Turkey, in 2005 and 2007 respectively, both in electrical and electronics engineering, and Ph.D. degree from the New Jersey Institute of Technology, Newark, NJ, in 2013, in electrical engineering. He has been working as a software development engineer at Amazon.com AWS since June 2013. His interests include cloud computing, high performance computing, data-intensive research in signal processing, multi-resolution signal processing, statistical signal processing, pattern classification, neural networks, genetic algorithms; and their applications in quantitative finance, electronic trading, digital communications, digital imaging, and biomedical engineering.

Onur Yilmaz received his B.S. degree in Computer and Educational Sciences and M.S. degree in Computer Science, both from Ege University, Izmir, Turkey, in 2007 and 2011 respectively. Since 2011, he has been a Ph.D. candidate in the Department of Electrical and Computer Engineering at the New Jersey Institute of Technology, Newark, NJ. His research interests include high performance DSP, financial signal processing and machine learning.

Ali N. Akansu received his B.S. degree from the Technical University of Istanbul, Turkey, M.S. and Ph.D. degrees from the Polytechnic University, Brooklyn, New York, all in Electrical Engineering. He has been with the Department of Electrical & Computer Engineering at the New Jersey Institute of Technology since 1987 where he is Professor of Electrical & Computer Engineering. He was a Founding Director of the New Jersey Center for Multimedia Research and NSF Industry-University Cooperative Research Center for Digital Video. Dr. Akansu was the Vice President for Research and Development of IDT Corporation. He was the founding President and CEO of PixWave, Inc., and Senior VP for Technology Development of TV.TV (IDT subsidiaries). He served on the boards of several companies and an investment fund. He visited David Sarnoff Research Center, IBM T.J. Watson Research Center, GEC-Marconi Electronic Systems Corp., and Courant Institute of Mathematical Sciences at the New York University. Dr. Akansu has published numerous articles and several books on his research work. His current professional interests include theory of signals and transforms, electronic trading, financial signal processing, and high performance DSP (FPGA & GPU computing).