A novel GPU implementation of eigenanalysis for risk management

Conference Paper · June 2012

DOI: 10.1109/SPAWC.2012.6292956

2 authors, including:

Ali Akansu
New Jersey Institute of Technology
242 PUBLICATIONS **3,842** CITATIONS

SEE PROFILE

A NOVEL GPU IMPLEMENTATION OF EIGENANALYSIS FOR RISK MANAGEMENT

*Mustafa U. Torun and Ali N. Akansu**

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

ABSTRACT

Portfolio risk is commonly defined as the standard deviation of its return. The empirical correlation matrix of asset returns in a portfolio has its intrinsic noise component. This noise is filtered for more robust performance. Eigendecomposition is a widely used method for noise filtering. Jacobi algorithm has been a popular eigensolver technique due to its stability. We present an efficient GPU implementation of parallel Jacobi eigensolver for noise filtering of empirical correlation matrix of asset returns for portfolio risk management. The computational efficiency of the proposed implementation is about 34% better than our most recent study for an investment portfolio of 1024 assets.

Index Terms— Portfolio Risk, Jacobi Algorithm, Eigen Decomposition, GPU, CUDA

1. INTRODUCTION

Standard deviation of portfolio return is a commonly used metric for risk measurement in quantitative finance [1, 2]. It is a function of capital allocation among financial assets in the portfolio, and correlation matrix of asset returns. Filtering the built-in noise of empirical correlation matrix of financial asset returns by using eigendecomposition is a widely used and effective method [2]. It was shown that, the risk measured from the noise filtered version of empirical correlation matrix yields more robust performance [2]. However, eigendecomposition is a computationally expensive algorithm. Hence, its implementation cost is of a main concern for most applications. An analytical framework utilizing discrete cosine transform (DCT) as efficient approximation to Karhunen-Loeve transform (KLT) for certain class of correlation matrix was recently forwarded in the literature [3].

Numerical calculation of eigendecomposition for large dense matrices is still an active and challenging research topic in applied mathematics and engineering. Jacobi algorithm [4] as an eigensolver, introduced in 1846, has not received much attention until recently due to its high computational cost compared to the other algorithms including popular QR. However, it was theoretically shown that it is the most stable numerical eigensolver algorithm [5]. Hence, with the advent of affordable general purpose graphics processing unit (GP-GPU or GPU), research interest on Jacobi algorithm has gained momentum, recently [6, 7].

Robustness and speed are important concerns impacting performance in quantitative finance and electronic trading. Therefore, efficient implementation of eigensolvers is desirable for risk management of an investment portfolio where eigen filtering employed. In this paper, we introduce an improved GPU implementation of parallel Jacobi eigensolver compared to a recent study [8]. Eigen filtering of asset return correlations is given as an application of the proposed GPU implementation of Jacobi algorithm. The improvement is about 34% compared to our recent study for a portfolio of 1024 assets.

2. PORTFOLIO RISK

2.1. Definitions

Let $\mathbf{q} = \begin{bmatrix} q_1 & q_2 & \cdots & q_N \end{bmatrix}^{\mathrm{T}}$ be a size $N \times 1$ investment allocation vector for a portfolio of N financial assets and let $\mathbf{R} = \begin{bmatrix} R_1 & R_2 & \cdots & R_N \end{bmatrix}^{\mathrm{T}}$ be an $N \times 1$ vector comprised of asset returns where superscript T is the matrix transpose operator. Note that q_i may be the dollar amount or ratio of the total capital invested in the i*th* asset. The choice of capital allocation reflects itself to the portfolio risk. The return of the N-asset portfolio is expressed as follows

$$R_p = \mathbf{q}^{\mathrm{T}}\mathbf{R} = \sum_{i=1}^{N} q_i R_i.$$

Portfolio risk is the standard deviation of portfolio return. For an N-asset portfolio, it is calculated as

$$\sigma_p = \left(E\left[R_p^2\right] - \mu_p^2\right)^{1/2} = \left(\mathbf{q}^{\mathrm{T}}\mathbf{\Sigma}^{\mathrm{T}}\mathbf{C}\mathbf{\Sigma}\mathbf{q}\right)^{1/2} \qquad (1)$$
$$= \left(\sum_{i=1}^{N}\sum_{j=1}^{N} q_i q_j \rho_{ij} \sigma_i \sigma_j\right)^{1/2},$$

*Corresponding author: akansu@njit.edu

where $\mu_p = E[R_p] = \mathbf{q}^{\mathrm{T}} E[\mathbf{R}] = \mathbf{q}^{\mathrm{T}}\boldsymbol{\mu}$ is the expected return of the portfolio, $\boldsymbol{\mu}$ is an $N \times 1$ vector, and its elements are expected returns of assets, $\boldsymbol{\Sigma}$ is an $N \times N$ diagonal matrix with elements corresponding to the volatilities of assets (standard deviation of corresponding returns), and $\mathbf{C}$ is $N \times N$ correlation matrix where $[C]_{ij} = \rho_{ij}$. Note that all elements on the main diagonal of $\mathbf{C}$ are equal to one. Furthermore, $\mathbf{C}$ is a symmetric and positive definite matrix. It was shown that $\mathbf{C}$ has intrinsic noise and filtering it out offers more robust risk estimation [2]. The eigen filtering of noise in $\mathbf{C}$ is discussed next.

2.2. Eigen Filtering of Noise

Let us perform eigen analysis of the empirical correlation matrix. We can express $\mathbf{C}$ as

$$\mathbf{C} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{\mathrm{T}}, \tag{2}$$

where $\boldsymbol{\Lambda} = diag(\lambda_1, \lambda_2, \ldots, \lambda_N)$ is a diagonal matrix with the eigenvalues, λ_k as the k*th* eigenvalue $\lambda_k \geq \lambda_{k+1}$, $\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_N \end{bmatrix}$ is an $N \times N$ matrix composed of N eigenvectors as its columns, and $\mathbf{v}_k$ is the $N \times 1$ eigenvector corresponding to the k*th* eigenvalue, λ_k. Note that $\lambda_k \geq 0\ \forall_k$ and $\sum_k \lambda_k = N$. We substitute (2) into (1) and express the portfolio risk as

$$\sigma_p = \left(\mathbf{q}^{\mathrm{T}}\boldsymbol{\Sigma}^{\mathrm{T}}\mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{\mathrm{T}}\boldsymbol{\Sigma}\mathbf{q}\right)^{1/2}.$$

Then, we use eigenfiltered empirical correlation matrix written as [2]

$$\tilde{\mathbf{C}} = \sum_{k=1}^{L} \lambda_k \mathbf{v}_k \mathbf{v}_k^{\mathrm{T}} + \mathbf{E}, \tag{3}$$

where L is the number of selected factors (eigenvalues) with $L \ll N$, and $\mathbf{E}$ is a diagonal noise matrix added into the equation in order to preserve the total variance. The elements of $\mathbf{E}$ are defined as

$$[E]_{ij} = \epsilon_{ij} = \begin{cases} 1 - \sum_{k=1}^{L} \lambda_k v_i^{(k)} v_j^{(k)} & i = j \\ 0 & i \neq j \end{cases}, \tag{4}$$

where $v_i^{(k)}$ is the i*th* element of the k*th* eigenvector. Note that the error term of (4) keeps the trace of the correlation matrix, $\tilde{\mathbf{C}}$, to be equal to *N*. From (3) and (4) elements of noise filtered version of empirical correlation matrix is expressed as

$$\left[\tilde{C}\right]_{ij} = \tilde{\rho}_{ij} = \sum_{k=1}^{L} \lambda_k v_i^{(k)} v_j^{(k)} + \epsilon_{ij}. \tag{5}$$

Therefore, eigen filtered portfolio risk is calculated by substituting the filtered version of ρ_{ij} in (5) into (1) shown as

$$\tilde{\sigma}_p = \left(\sum_{k=1}^{L} \lambda_k \left(\sum_{i=1}^{N} q_i v_i^{(k)} \sigma_i \right)^2 + \sum_{i=1}^{N} \epsilon_{ii} q_i^2 \sigma_j^2 \right)^{1/2}. \tag{6}$$

In order to implement (6) in hardware, a robust and fast eigensolver algorithm is needed. Jacobi algorithm is utilized due to its robustness [5] and friendly nature to highly parallel computational hardware systems [4]. Next, we define the classical and parallel Jacobi algorithms, and highlight the proposed improvements for the GPU implementation of the parallel algorithm as detailed in the following sections.

3. JACOBI ALGORITHM

Let $\mathbf{A}$ be a symmetric 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 [9]

$$\mathbf{A} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{\mathrm{T}}, \tag{7}$$

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 $N \times N$ matrix defined as

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_N \end{bmatrix},$$

$\mathbf{v}_i$ is an $N \times 1$ eigenvector corresponding to the i*th* eigenvalue, λ_i. Jacobi algorithm provides an approximated numerical solution to (7) by iteratively reducing the metric [4]

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

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

$$\mathbf{A}^{(k+1)} = \mathbf{J}^{\mathrm{T}}(p, q, \theta)\mathbf{A}^{(k)}\mathbf{J}(p, q, \theta), \tag{8}$$

where $1 \leq p < q \leq N$. Matrix $\mathbf{J}(p, q, \theta)$ is sparse as defined

$$[J(p, q, \theta)_{ij}] = \begin{cases} \cos\theta & i = p,\ j = p \\ \sin\theta & i = p,\ j = q \\ -\sin\theta & i = q,\ j = p \\ \cos\theta & i = q,\ j = q \\ 0 & otherwise \end{cases}. \tag{9}$$

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 (8) are repeated until $\mathrm{off}(\mathbf{A}) < \epsilon$ where ϵ is a predefined threshold value. After sufficient number of rotations, matrix $\mathbf{A}$ gets closer to matrix Λ, and the successive multiplications lead us to an approximation of the eigenvector matrix $\mathbf{V}$ expressed as [4]

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

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

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} \tilde{A}_{pp} & 0 \\ 0 & \tilde{A}_{qq} \end{bmatrix}, \tag{10}$$

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

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

Note that since p and q define the rotation matrix $\mathbf{J}(p, q, \theta)$ through (10) we can drop the angle, θ, and refer it as $\mathbf{J}(p, q)$. Originally, p and q in (8) are chosen such that $|A_{pq}| = \max_{i \neq j} |A_{ij}|$ [4]. 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, i.e. $(p^{(m)}, q^{(m)}) = \{(1, 2), (1, 3), (1, 4), \ldots, (2, 3), \ldots\}$ where $m = 1, 2, \ldots N/2$. Note that it is straightforward to implement the algorithm on a computing system with $N/2$ parallel processing units due to the sparsity of the rotation matrix $\mathbf{J}(p, q)$.

3.1. Notation

In this section, we present the notation employed in order to make the following discussion clearer. 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

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

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

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

where $p^{(m)}$ and $q^{(m)}$ are the integers that define the submatrix of the data matrix to be rotated in 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 m*th* processing unit as follows

$$\mathbf{J}^{(m)} = \begin{bmatrix} J_{p^{(m)}p^{(m)}} & J_{p^{(m)}q^{(m)}} \\ J_{q^{(m)}p^{(m)}} & J_{q^{(m)}q^{(m)}} \end{bmatrix} = \begin{bmatrix} c_m & s_m \\ -s_m & c_m \end{bmatrix}, \tag{11}$$

where J_{ij} is an element of original Jacobi matrix defined in (9). Since this is an iterative algorithm, we need to state the iteration number, k, explicitly as shown in (8). However, in order to keep the text clean, we use assignment operator, $\leftarrow$, instead of the iteration number in the paper.

3.2. Parallel Jacobi Algorithm

Since Jacobi rotation matrix (9) is sparse, it is possible to perform rotations given in (8) in a parallel implementation using $N/2$ processing units. However, since multiplication $\mathbf{J}(p, q)^{\mathrm{T}}\mathbf{A}$ in (8) would update the p*th* and q*th* rows of matrix $\mathbf{A}$ and multiplication $\mathbf{A}\mathbf{J}(p, q)$ in (8) would update the p*th* and q*th* columns of matrix $\mathbf{A}$, it would be problematic to implement (8) in parallel without proper synchronization. The solution is to introduce an intermediary matrix $\mathbf{X}$ of size $N \times N$ [6]. 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, (11), as given

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

A blocking synchronization (waiting for all the processing units to finish) is required before proceeding to the next kernel that is the multiplication of $p^{(m)}th$ and $q^{(m)}th$ columns of $\mathbf{X}$ by Jacobi sub-rotation matrix, (11), as expressed

$$\mathbf{A}^{(m)} \leftarrow \mathbf{X}^{(m)}\mathbf{J}^{(m)}. \tag{13}$$

Note that, the eigenvectors may be updated according to

$$\mathbf{V}^{(m)} \leftarrow \mathbf{V}^{(m)}\mathbf{J}^{(m)}, \tag{14}$$

in the same kernel since $\mathbf{J}^{(m)}$ is already accessed in (13). Therefore, two kernel calls, one for (12), and one for (13) and (14), are required for one step in a sweep of the parallel Jacobi algorithm. Second kernel call must wait for the first one to finish via global synchronization. In a recent study [8] we implemented 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 [10], we introduced additional modifications to the kernels that are beyond the scope of this discussion. The global synchronization was realized through CPU (host synchronization).

3.3. Improved Parallel Algorithm

Memory access time is an important limiting factor in the state-of-art GPU computing technologies. Even the GPUs with the latest Fermi™ [11] architecture, providing L1 and L2 caches for the GPU main memory, suffer from performance degradations when the memory access patterns are unstructured [10]. Memory coalescing, i.e. refining the memory access pattern such that the hardware can make combined requests from DRAM, should be employed whenever applicable. In a recent work [8], we revisited this limiting factor

in GPU applications [10] and showed that modeling matrices $\mathbf{A}$ and $\mathbf{X}$ as row-major and column-major, respectively, the computational performance improves significantly since the memory access pattern is in alignment with the operations given in (12), (13), and (14). Moreover, we proposed the following modification to (13)

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

that ensures coalesced access [11] to the memory locations reserved for matrix $\mathbf{A}$. This method was called as "Symmetric Access (SA)". Note that after every processing unit finishes the update given in (15), matrix $\mathbf{A}$ is updated in the same way as it would be updated with (13) since $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$. However, this trick can not be applied directly to $\mathbf{X}$ since it is not symmetrical. Nevertheless, we proposed a new method called "Maximum-Coalesced Access (MCA)" that also provides coalesced access to the memory locations reserved for $\mathbf{X}$ [8]. In the next section, we revisit MCA and show that it makes it possible to reduce the kernel calls to just one. Using this structure, we propose a novel implementation that improves the computational efficiency by 34%.

3.4. Maximum-Coalesced Access

Maximum-Coalesced Access (MCA) modifies the update given in (12) as follows

$$\mathbf{X}^{(m)} \leftarrow \mathbf{K}\left[\overline{\mathbf{A}}^{(m)}\right]^{\mathrm{T}}, \tag{16}$$

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

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

$J_{ij}^{(m)}$ is the element of Jacobi sub-rotation matrix, (11), 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 [8]

$$\begin{aligned} X_{ip^{(m)}} &= A_{ip^{(m)}}K_{ii} + A_{f(i)p^{(m)}}K_{if(i)} \\ X_{iq^{(m)}} &= A_{iq^{(m)}}K_{ii} + A_{f(i)q^{(m)}}K_{if(i)}, \end{aligned} \tag{17}$$

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, the update equation given in (17) is feasible to realize. It was shown in [8] that, since (17) accesses only to $2N$ elements of $\mathbf{K}$, it is more efficient (in terms of memory requirements) to define two $N \times 1$ vectors $\left[k_i^{(1)}\right] = K_{ii}$ and $\left[k_i^{(2)}\right] = K_{if(i)}$ instead of $N \times N$ sized $\mathbf{K}$. Then, one may modify (17) accordingly as expressed

$$\begin{aligned} X_{ip^{(m)}} &= A_{ip^{(m)}}k_i^{(1)} + A_{f(i)p^{(m)}}k_i^{(2)} \\ X_{iq^{(m)}} &= A_{iq^{(m)}}k_i^{(1)} + A_{f(i)q^{(m)}}k_i^{(2)}. \end{aligned}$$

Performance improvement of MCA over SA (see Subsection 3.3) is reported to be about 2.7% [8]. Note that in MCA (12) and (13) are still performed by using two kernel calls. Therefore, the resulting $\mathbf{X}$ is needed to be stored in the memory at the end of the first kernel, and it has to be retrieved back at the beginning of the second kernel. In the next section, we show that MCA paves the way for the possibility of performing (12) and (13) by using only one kernel call. Hence, it reduces the unnecessary memory traffic. The proposed GPU implementation provided 34% performance improvement over MCA.

3.5. One Step Parallel Jacobi Algorithm

It was discussed in Subsection 3.2 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 (12) and (13) with proper synchronization among the assigned processing units. However, thanks to MCA, it is possible to perform these two updates in only one kernel. By substituting (16) into (15) we obtain

$$\overline{\mathbf{A}}^{(m)} \leftarrow \left[\mathbf{K}\left[\overline{\mathbf{A}}^{(m)}\right]^{\mathrm{T}}\mathbf{J}^{(m)}\right]^{\mathrm{T}}. \tag{18}$$

Note that (18) updates only the $p^{(m)}th$ row and $q^{(m)}th$ column of $\mathbf{A}$, and it does not need intermediary matrix $\mathbf{X}$. We call the algorithm implementing (18) as "One Step Parallel Jacobi Algorithm (OSPJ)." In the next section, we perform simulations to compare performance of OSPJ and MCA under the same test conditions, and report improvements of GPU over CPU implementation.

4. PERFORMANCE SIMULATIONS

We tested the cyclic Jacobi algorithm by running it on a CPU. The CPU is a four-core (eight with hyper-threading) Intel(R) Core™ i7 960. The workstation has 24 GB RAM, and it has Linux OS. We tested the GPU implementations of MCA and

	64	128	256	512	1024
CPU	34.5	270.3	2,404.9	22,107.5	378,122.7
GPU (MCA)	42.0	89.7	223.8	839.1	5,143.2
GPU (OSPJ)	32.4	69.2	165.0	546.4	3,392.8

Table 1. Computation times of cyclic Jacobi algorithm on CPU, MCA on GPU, and OSPJ on GPU implementations, for various portfolio sizes (dimension of the empirical correlation matrix).

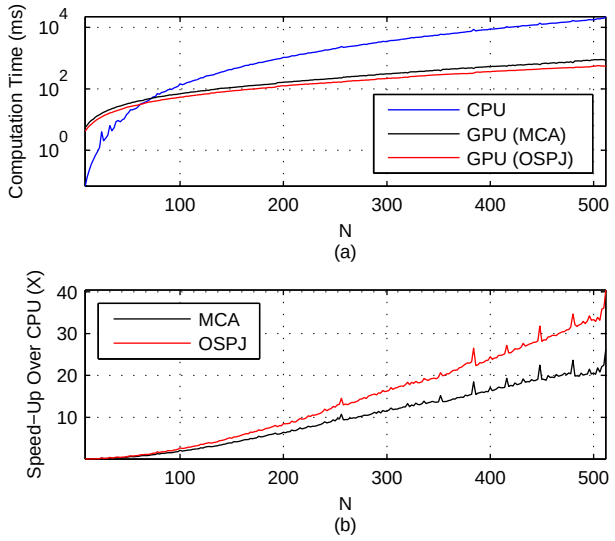

Fig. 1. (a) Computation times of cyclic Jacobi algorithm on CPU, MCA on GPU, and OSPJ on GPU implementations, for various portfolio sizes, (b) Speed-up of MCA and OSPJ algorithms on GPU over cyclic Jacobi algorithm on CPU. Local peaks correspond to the portfolio sizes that are multiples of 32 (warp size for Tesla). GPUs perform better when the number of total threads is a multiple of the warp size [10].

OSPJ by running them on NVIDIA Tesla™ C2070 with 448 CUDA™ Cores with 5,375 MB global memory. The Tesla™ hardware is installed on the same workstation. All floating point operations are performed with single-precision. Timing results are averaged over 20 runs. Computation time for CPU, MCA, and OSPJ versus the portfolio size (dimension of the correlation matrix of the financial asset returns) is tabulated in Table 1. It is observed from the table that performance improvement of OSPJ over MCA is 34%, over CPU is 99.1% for a portfolio of 1024 financial assets. A more detailed performance comparison is displayed in Fig. 1 for various size portfolios of up to 512 assets. The performance improvement of OSPJ over MCA and CPU is clearly quantified in this figure. Note that OSPJ runs over 40x faster than CPU for a portfolio size of 512 assets.

5. CONCLUSIONS

Maximum Coalesced Access (MCA) method for GPU implementation of the Jacobi algorithm is revisited, and a novel algorithm offering 34% performance improvement over MCA is proposed. The proposed algorithm is successfully employed for eigenfiltering of built-in noise in empirical correlation matrix that is widely used for risk calculations of an investment portfolio. It is expected that the GPU computing will play a central role in real-time implementation of data-intensive DSP techniques employed in many fields including financial analytics.

6. REFERENCES

[1] H. M. Markowitz, *Portfolio Selection: Efficient Diversification of Investments*. Wiley, New York, 1959.

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

[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] 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.

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

[8] M. U. Torun, O. Yilmaz, and A. N. Akansu, "Novel GPU implementation of Jacobi algorithm for Karhunen-Loève transform of dense matrices," *IEEE CISS*, 2012. Under review.

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

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

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