A parallel implementation of One-Sided Jacobi SVD for non-symmetric squared matrices on a high-performance GPU

R. I. Acosta-Quiñonez*, D. Torres-Roman*, R. Rodríguez-Ávila† and D. Robles-Valdez*
*CINVESTAV-IPN Department of Electrical Engineering and Computer Science, Telecommunications,
Guadalajara unit, Jalisco, México, Email:{iacosta, dtorres, drobles}@gdl.cinvestav.mx
†Intel Guadalajara Design Center, Intel Labs, Jalisco, México, Email: roel.rodriguez.avila@intel.com

***Abstract*—Success of modern technology enterprises in a highly-competitive and fast-changing market relies on the efficient prototyping of high-performance and low-consumption digital devices. GPUs have become the key for reducing the time-to-market of these digital devices by providing a highly-reconfigurable parallel processing platform for implementing computationally expensive DSP algorithms. This paper exposes the benefits of using a GPU for accelerating DSP algorithms used in big data analysis and scientific computing. Particularly, a parallel-structured implementation of the One-Sided Jacobi algorithm for the SV decomposition is analyzed. Two contributions are highlighted, first, the highest reported speedup for the One-Sided SVD algorithm is achieved and, second, a mixed formal/intuitive analysis technique is applied to cyclic algorithms with the aim of adequate them to GPU platforms.**

I. Introduction

As data rates increase continuously in modern digital devices, efficient implementations of a wide variety of digital signal processing (DSP) algorithms are essential . Efficiency metric is mainly based on throughput and reconfigurability to fulfill incremental improvements for new applications. The ad-hoc implementations of parallel DSP algorithms possess the highest throughput. However, the main drawback is the lack of reconfigurability because the hardware architecture is tied to a specific set of requirements for a particular algorithm. In general, the performance of a DSP parallel implementation is inversely related to the reconfigurability of the hardware architecture that supports it, i.e. the most reconfigurabiity, the less performance. Then, an appropriate trade-off between these features is desired in DSP parallel implementations because it allows an incremental improvement of already-designed applications, then accelerating the prototyping while maintaining an adequate performance.

A platform possessing adequate flexibility to be reconfigured for testing different versions of commonly-used parallelized DSP algorithms has become crucial. Modern graphics cards offer the possibility of using arrays of processors in a parallel structure along with languages and environments specifically designed to efficiently use these resources. By using these graphic cards, the validation time is reduced compared with an ad-hoc parallel architecture specified for different versions of the same algorithm, that is, it has reconfigurability. Specifically the high-performance NVIDIA cards with Kepler architecture are the most efficient graphics cards for parallel-data computing.

Aside, some DSP algorithms are obvious candidates to be implemented in a parallel architecture like a graphics processing unit (GPU) because they are used in a wide variety of modern digital devices. Particularly, the singular value decomposition (SVD) is one of the most-commonly used DSP algorithms in different fields of technology. Citing some examples: Performance analysis for adaptive MIMO transmission in a cellular system [1], image compression [2], inverse calculation of the optimal filter for ultrasonic applications [3] and image processing for face recognition [4]. All these applications use the SVD algorithm as one of their main processing block because it is numerically stable. In addition, there are already-designed parallel version for this algorithms, then the step of implementing and testing them in a GPU is straightforward.

Finally, cyclical algorithms analysis techniques to obtain an expression with reduced data dependency have been developed and can be applied to the SVD algorithm. Thus allowing to implement that modified algorithm on a GPU efficiently.

1) Related work: Lahabar and Narayanan, in [5], proposed the first implementation of the SVD algorithm on a GPU. This work uses the steps of bi-diagonalization followed by the Householder diagonalization and QR decomposition. This implementation was compared to the Intel libraries of mathematical algorithms and it is capable of significantly improve their performance.

Kotas and Barhen, in [6], continued the work in [5] by refining the implementation of the SVD algorithm on a GPU. In this work, a first implementation performs the bidiagonalization step on GPU and the Householder diagonalization and QR steps are performed on the central processing unit (CPU), using the GPU only for updating data. Also in this work, a second implementation is based on the iterative algorithm One-Sided Jacobi. This work reduces the CPU-GPU data

exchange thus reducing latency.
Wang et. al., in [7], compares an implementation of the SVD algorithm for symmetric matrices on a GPU against traditional sequential implementation. This work evidence that the use of modern GPUs to implement DSP algorithms allows for a high-throughput implementation while, at the same time, the efficient use of hardware resources reduces power consumption.

The main drawback of the three previous works is the lack of data-dependency analysis and there is no use of the different memory hierarchies. Only intuitive data-dependency analysis was performed in the three mentioned papers.

This paper proposes a parallel implementation of the One-Sided Jacobi algorithm to solve the SVD of square matrices on an NVIDIA Tesla K20 GPU using compute unified device architecture (CUDA) environment and an Intel Core i7-3770 CPU, Win7 64-bit operating system.

Contributions of this paper are summarized as follows:

- A performance analysis of different parallel-SVD implementations using global memory, surface memory and texture memory was performed with the aim to improve the throughput of the parallel GPU implementation.
- A mixed theoretical-intuitive data dependency analysis to extract the maximum parallelism of cyclic algorithms was applied.
- The two previous contributions allow for a speedup improvement of SVD calculation up to 133766 by using the dynamic parallelism.

Experimental results show that the parallel One-Sided Jacobi algorithm running on a high-performance GPU is a good option for a fast prototyping of high-throughput SVD algorithm.

The paper is organized as follows: Section II presents mathematical fundamentals for the SVD. Section III gives an overview of CUDA and GPU. In Section IV, it is described the sequential One-Sided Jacobi algorithm to solve the SVD of squared matrices. Also, it presents the algorithm implementation on GPU and a data-dependency analysis. Section V shows the experimental results. Finally, Section VI exposes some conclusions.

II. Singular Value Descomposition

This section briefly describes the fundamentals for the SVD algorithm based on iterative Jacobi rotations.

A. Fundamentals

The SVD of a matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ [8], [9], is based on the following theorem:

Theorem 2.1 (Fundamental theorem of linear algebra): Given a ρ-rank matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$, the singular value decomposition of $\mathbf{A}$ is represented by the product of two unitary matrices $\mathbf{U} \in \mathbb{C}^{m\times m}$, $\mathbf{V} \in \mathbb{C}^{n\times n}$ and a diagonal matrix $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ [9], as described in Eq. (1)

$$\mathbf{A} = \mathbf{U\Sigma V^H} \tag{1}$$

The columns in $\mathbf{U}$ are called the *left singular vectors* of $\mathbf{A}$ and the columns in $\mathbf{V}$ are called the *right singular vectors* of $\mathbf{A}$. Also, $\mathbf{\Sigma}$ is a real non-negative matrix as shown in the Eq. (2).

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & \sigma_\rho & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

The diagonal elements of $\mathbf{\Sigma}$ are called *singular values* (σ) of $\mathbf{A}$ and are usually ordered decreasingly as:

$$\sigma_1 \geq, ..., \geq \sigma_\rho > 0,\ \sigma_{\rho+1} = ... = 0, \tag{3}$$

In the case of a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, the eq.(1) is expressed as:

$$\mathbf{A} \;=\; \mathbf{U\Sigma V^T}, \tag{4}$$

where $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are orthonormal matrices by definition:

$$\begin{array}{lclcl} \mathbf{UU^T} & = & \mathbf{U^TU} & = & I^{m\times m} \\ \mathbf{VV^T} & = & \mathbf{V^TV} & = & I^{n\times n} \end{array} \tag{5}$$

Considering this property and Eq.(4), the products $\mathbf{AA}^T$ and $\mathbf{A}^T\mathbf{A}$ are defined by:

$$\begin{array}{l} \mathbf{AA}^T = \mathbf{U} \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & \lambda_\rho & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{U}^T \\ \mathbf{A}^T\mathbf{A} = \mathbf{V} \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & \lambda_\rho & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{V}^T \end{array} \tag{6}$$

Other important properties for the SVD can be found in [10].

B. Jacobi rotation

Jacobi rotation is a rotation of a 2-dimensional subspace in an n-dimensional space, denoted by $\mathbf{J}$. A pair of elements of a symmetric matrix $\mathbf{B} \in \mathbb{R}^{n\times n}$ are set to zero after the application of this transformation, denoted as $\mathbf{B} \longmapsto \mathbf{J^TBJ} = \mathbf{B}'$ [14].

$$\mathbf{J}(i,j,\theta) \;=\; \begin{array}{c} \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \begin{array}{c} \\ \\ i \\ \\ j' \\ \\ \\ \end{array} \\ \qquad i \qquad\quad j \end{array} \tag{7}$$

where $c = \cos\theta$, $s = \sin\theta$ and θ is the rotation angle in the (i, j)-plane. Only i-th and j-th rows of $\mathbf{B}$ are affected. Similarly, only the j-th and i-th columns are affected. The elements b'_{ij}, b'_{ji}, b'_{ii} and b'_{jj}, in $\mathbf{B}$ are used to calculate the angles in the rotation matrices. Rotation matrices eliminate the elements b'_{ij} and b'_{ji}, as shown in Eq. (7). Jacobi algorithm performs a sequence of orthogonal updates, where each new matrix $\mathbf{B}$ is more diagonal than its predecessor. Eventually the off-diagonal elements are small enough, so they can be considered as zeros. In particular, each Jacobi rotation involves a pre-multiplication and a post-multiplication of $\mathbf{B}$ by orthogonal matrices. In general, $(n^2 - n)/2$ rotations are performed (in the case of a symmetric matrix) trying to make zero all off-diagonal elements. This $(n^2 - n)/2$ transformations constitute a sweep [14]. Commonly, the Jacobi rotations are applied by using one of the following approaches: performing cyclical rotations per row or performing cyclical rotations per column. In these approaches, the pair (i, j) is selected in a row by row or column by column fashion, respectively. For example, if $n = 4$, the rotation sequence is: $(i, j) = (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)$.

C. Parallel-Order Jacobi algorithm

In the traditional sequential implementations of the Jacobi algorithm, the parallelism is not exploited, mainly due to the data dependence of a rotation with its predecessor. The parallel Jacobi algorithm described by Brent et. al., in [15], exploits the maximum parallelism for the decomposition of a symmetric matrix. This algorithm requires $n - 1$ number of steps, each step having $n/2$ rotations. All rotations within a step can be executed in parallel. As an example, consider an $n \times n$ matrix where $n = 4$. Then, it takes 3 steps for the decomposition of a symmetric matrix and each step has 2 rotations.

$$\begin{array}{llll} \textit{step} & 1 & (1,2) & (3,4) \\ \textit{step} & 2 & (1,4) & (2,3) \\ \textit{step} & 3 & (1,3) & (2,4) \end{array}$$

III. CUDA and GPU

There is a transition in modern computer systems where data processing, being traditionally performed in the CPU, is now split between CPU and GPU. NVIDIA has developed the CUDA parallel computing architecture, included in modern graphic cards like GeForce, ION, Quadro and Tesla [16]. CUDA architecture is referred both as a compiler and a set of development tools created by NVIDIA that allows to use a variant of the C programming language to code algorithms on NVIDIA GPUs. Wrappers can be used to interact with Python, Fortran and Java rather than C/C++. The host code is compiled and executed in the CPU and it can be written in standard language C/C++. The device code can be written using extended C identifiers to label functions running on the GPU, called kernel function, and their associated data structures. The GPU works together with the CPU by communicating through PCI Express port [17].

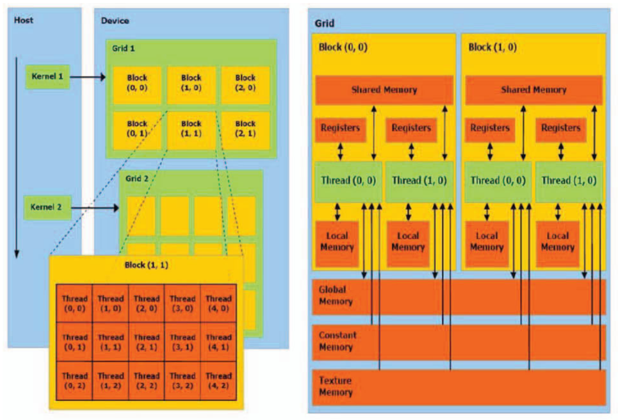

Fig. 1. Memory hierarchy for NVIDIA GPUs

CUDA leverages the NVIDIA GPU power to increase the performance of the parallel computations compared to traditional architectures. It explores the advantages of the GPUs compared to general purpose CPUs using the parallelism offered by their multiple cores, which allow the processing of a large number of simultaneous threads. Therefore, if an application is designed to use many threads, then this is ideal to be implemented on a GPU. Also, GPUs are useful in applications with a large arithmetic cost that offset the cost of many accesses to the main memory.

This is due to the high degree of parallelism and high bandwidth of the memories [16].The memory space of CUDA is divided into register, local memory, shared memory, global memory, constant memory, and texture memory and the parallel kernel computation is arranged in blocks or grids. The left side in Fig. 1 shows the threads arranged in a two-level grid. Each grid is conformed by one or more blocks of threads. All blocks are the same size in a grid. Moreover, all the blocks contain the same number of threads. When a kernel is invoked, it runs as a grid of parallel threads. Each grid generally contains thousands to millions of threads. A block is a batch of threads that can cooperate together by synchronization on the execution and secure memory access. Two threads of different blocks can not cooperate [18], [19].

The right part of Fig. 1 shows the memory structure and different allowed accesses. Two memories can be observed, the global memory and the constant memory. The host can read and write in both memories. However, the GPU only can access the constant memory in read mode. The texture is a read-only memory to the device and is written by the host. The surface memory is similar to texture memory, with the difference that it has reading and writing functions in the kernel. The NVIDIA Tesla Accelerated Computing Platform is the leading platform for scientific computing.

IV. One-Sided Jacobi Algorithm for Computing the Singular Value Decomposition

The One-Sided Jacobi algorithm is derived from the Two-Sided Jacobi algorithm and it is used to compute the eigenvalues decomposition of a matrix $\mathbf{B} = \mathbf{A^T A}$. Where the matrix $\mathbf{A} = [\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n}]$, and $\mathbf{a_n}$ are column vectors. Off-diagonal elements are removed from $\mathbf{B} = \mathbf{A^T A}$ [20]. It is required to calculate the elements b_{ij}, b_{jj} and b_{ii} as shown below:

$$\begin{aligned} \alpha &= b_{ii} = \mathbf{a_i^T a_i}, \\ \beta &= b_{jj} = \mathbf{a_j^T a_j}, \\ \gamma &= b_{ij} = \mathbf{a_i^T a_j} \end{aligned} \tag{8}$$

Once b_{ii}, b_{jj} and b_{ij} are obtained, Jacobi rotations corresponding to the parameters c and s as in the algorithm Two-Sided Jacobi are calculated by using the auxiliar variables ζ and t as:

$$\zeta = \frac{\beta-\alpha}{2\gamma}, \tag{9}$$

and,

$$t = \frac{sgn(\zeta)}{|\zeta|+\sqrt{\zeta^2+1}}, \tag{10}$$

This leads to the expression for c and s as:

$$\begin{aligned} c &= \frac{1}{\sqrt{1+t^2}}, \\ s &= ct \end{aligned} \tag{11}$$

As a summary, the steps of using sequential Jacobi algorithm are given as follows:

1. Generate a rotation matrix $\mathbf{J}$.
   1.1 Calculate α, β, γ as in Eq. (8) for a given pair (i, j).
   1.2 Calculate Jacobi rotation matrix that diagonalizes $\mathbf{B}$ with equations (9), (10) y (11).
2. Post-Multiply the $\mathbf{J}$ rotation matrix with the pair of columns i, j of $\mathbf{A}$
3. Post-Multiply the $\mathbf{V}$ matrix of singular vectors by the $\mathbf{J}$ rotation matrix with the pair of columns i, j.
4. If all columns satisfy: $convergence \leq TOL$ then proceed to step 5, else go to step 1 and update i, j.
5. Calculate the matrix $\mathbf{\Sigma}$ containing singular values by applying Eq. (12).

$$||\mathbf{A v_i}|| = \sigma_\mathrm{i} \tag{12}$$

6. Calculate the matrix $\mathbf{U}$ containing the left singular vectors by using Eq.(13).

$$\mathbf{AV} = \mathbf{U\Sigma} : \mathbf{u_i} = \mathbf{A v_i}/\sigma_\mathrm{i}, \tag{13}$$

The convergence criterion measures how much columns are orthogonal to each other. It is considered that all columns are orthogonal when the following threshold has been met:

$$|\gamma| / \sqrt{\alpha\beta} \tag{14}$$

Note that $\mathbf{A}$ is updated by $\mathbf{U\Sigma}$ in the iterative algorithm starting at step 3.

A. Sequential implementation of Jacobi rotations on GPU

Algorithm 1 shows the sequential implementation of cyclical-by-row Jacobi rotations. For this algorithm, the only sections that can be performed in parallel within the GPU are steps 2 and 3 until the convergence of the algorithm as well as the calculating matrices $\mathbf{U}$ and $\mathbf{\Sigma}$ once the convergence has been met. This is because the high data dependency between steps 1 - 3.
Several implementations have been tested by using some of the characteristics of CUDA libraries as CUBLAS, however, they could not remove high dependencies. This results in GPU implementations slower than implementation that runs on the CPU. It can be seen from algorithm 2. There are two steps implemented in the GPU for the sequential Jacobi algorithm and they correspond to steps 2 and 3. The resulting parallelism is $n/2$. To adequately design the parallel implementations of the algorithm in CUDA, it was necessary to analyze data dependencies among loops in the One-Sided Jacobi algorithm. Important observations on this algorithms are:

- The dot product of columns i-th and j-th of the matrix $\mathbf{A}$ is necessary for the calculation.
- The dot product of columns i-th and j-th of the matrix $\mathbf{A}$ is needed to update the matrix $\mathbf{V}$.
- The dot product of columns i-th and j-th of the matrix $\mathbf{A}$ updates these columns in the same matrix

Using the dependency analysis algorithm 5.1 in [21], the following direction vectors were obtained for matrix $\mathbf{A}$, where index j corresponds to Line 2, i to Line 3 and k to Line 9 in the Algorithm 1. In addition, values 0 and 1 in direction vectors represents a non-dependent and a dependent index, correspondingly:

- For variables A[k][i] and A[k][i] is [1, 0, 0].
- For variables A[k][i] and A[k][j] is [1, 1, 0].
- For variables A[k][j] and A[k][i] is [1, 1, 0].
- For variables A[k][j] and A[k][j] is [0, 1, 0].

For the case of the matrix $\mathbf{V}$, the same set of direction vectors is obtained, then showing that there is only dependency at level one (j) and level two (i) in the nested loops. The index k can be performed concurrently for updating corresponding matrices.

It is worth mentioning that, by using this analysis algorithm, great amount of time can be saved when trying to obtain the parallel expression for a particular sequential algorithm. This is because it leads to apply the intuitive part of loop analysis only to the indexes with no dependency. In other words, applying the intuitive part of the loop analysis since the beginning can lead to wasting time trying to parallelize highly-dependent operations. Another thing to notice is that this analysis algorithm does not expose how to obtain the parallel expression, but it focus the attention on the correct part. Then, by applying this analysis algorithm followed by an intuitive data-dependency analysis is the most efficient technique for obtaining the parallel expression of a particular cyclic algorithm.

Algorithm 1 Sequential_Jacobi_One_Sided_SVD

```
Require: Assume $\mathbf{A}^{n\times n}$, $\mathbf{V}^{n\times n}$ initially contains the identity matrix.
1: while $Converge > TOL$ do
2:   for $j = 1,\ N$ do
3:     for $i = 0,\ j < i$ do
4:       Determine $\alpha = \mathbf{a}_i^T\mathbf{a}_i,\ \beta = \mathbf{a}_j^T\mathbf{a}_j,\ \gamma = \mathbf{a}_i^T\mathbf{a}_j$
5:       $Max = \frac{|\gamma|}{\sqrt{\alpha\beta}}$
6:       if $Converge < Max$ then
7:         $Converge = Max$
8:       end if
9:       Determine $t,\ s,\ c$
         $tmp = \mathbf{a}_i$
         $\mathbf{a}_i = tmp * c - \mathbf{a}_j * s$
         $\mathbf{a}_j = tmp * s + \mathbf{a}_j * c$
         $tmp = \mathbf{v}_i$
         $\mathbf{v}_i = tmp * c - \mathbf{v}_j * s$
         $\mathbf{v}_j = tmp * s + \mathbf{v}_j * c$
10:    end for
11:  end for
12: end while
13: Determine the singular values from the magnitude of the resulting $\mathbf{A}$ matrix, $\mathbf{\Sigma}(i,i) = \|\mathbf{A}(:,j)\|$
14: Determine the left singular vectors from the resulting $\mathbf{A}$ matrix, $\mathbf{U}(:,j) = \frac{\mathbf{A}(:,j)}{\mathbf{\Sigma}(i,i)}$
```

Algorithm 2 Sequential_Jacobi_One_Sided_SVD_on_GPU

```
Require: Assume a one-dimensional array $\mathbf{A}[n\times n]$, $\mathbf{V}[n\times n]$ initially containing the identity matrix.
1: Allocate global memory in the device for the arrays $\mathbf{A}$, $\mathbf{V}$, $\mathbf{U}$ and $\mathbf{\Sigma}$
2: Copy arrays $\mathbf{V}$, $\mathbf{U}$ and $\mathbf{\Sigma}$ from the host main memory to the device global memory, before step 2
3: Execute Algorithm 1. Every time Line 9 in the Algorithm 1 has been reached, copy array $\mathbf{A}$ to GPU memory. Call Kernel described in Algorithm 3 to update $\mathbf{A}$ and $\mathbf{V}$. Then copy from device to host the resulting matrix $\mathbf{A}$
4: Once the convergence has been met, copy array $\mathbf{A}$ from the host main memory to the device global memory
5: Call the kernel for determine the singular values from the magnitude of the resulting $\mathbf{A}$ matrix, $\mathbf{\Sigma}(i,i) = \|\mathbf{A}(:,j)\|$
6: Call the kernel for determine the left singular vectors from the resulting $\mathbf{A}$ matrix, $\mathbf{U}(:,j) = \frac{\mathbf{A}(:,j)}{\mathbf{\Sigma}(i,i)}$
7: Copy arrays $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ from the device's main memory to the host global memory
8: Free memory of the GPU
```

Algorithm 3 kernel_Post-Multiply_Matrices_A-V_on_GPU

```
Require: Assume one-dimensional arrays $\mathbf{a}[n\times n]$ and $\mathbf{V}[n\times n]$, i, j, s and c are variables.
1: $ThreadID \longleftarrow blockIdx.x \times BLOCK\_SIZE + threadIdx.x$ \\ $blockIdx.x$ and $threadIdx.x$ are variables. $BLOCK\_SIZE$ is the number of threads in one $BLOCK$.
2: if $ThreadID < n$ then
3:   Each thread is used to multiply $c$ and $s$, store the elements $a[i] - a[j]$ and $v[i] - v[j]$ and update correspondingly
4: end if
```

B. Parallel implementation of Jacobi rotations on the GPU

In the case of Algorithm 4, a rearrangement of rotations in order to eliminate data dependencies is applied. By doing this, the inner loop for the variable $i = n/2$ can be performed in parallel, as well as step 10. The GPU implementation is presented in the algorithm 5, where the modification is reflected in Line 4 of Algorithm 4.
There are two kernel implementations for Algorithm 4. The first is the described previously and presented in algorithm 6. The second one use dynamic parallelism applied to Algorithm 3 and presented in Algotihm 7.

Algorithm 4 Parallel_Jacobi_One_Sided_SVD

```
Require: Assume $\mathbf{A}^{n\times n}$, $\mathbf{V}^{n\times n}$ initially contains the identity matrix.
1: while $Converge > TOL$ do
2:   for $j = 1,\ N - 1$ do
3:     for $i = 1,\ \frac{N}{2}$ do
4:       Determine $p, q$
         $p = min(top(j), bot(j))$
         $q = max(top(j), bot(j))$
5:       Determine $\alpha = \mathbf{a}_p^T\mathbf{a}_p,\ \beta = \mathbf{a}_q^T\mathbf{a}_q,\ \gamma = \mathbf{a}_p^T\mathbf{a}_q$
6:       $Max = \frac{|\gamma|}{\sqrt{\alpha\beta}}$
7:       if $Converge < Max$ then
8:         $Converge = Max$
9:       end if
10:      Determine $t,\ s,\ c$
         $tmp = \mathbf{a}_p$
         $\mathbf{a}_p = tmp * c - \mathbf{a}_q * s$
         $\mathbf{a}_q = tmp * s + \mathbf{a}_q * c$
         $tmp = \mathbf{v}_p$
         $\mathbf{v}_p = tmp * c - \mathbf{v}_q * s$
         $\mathbf{v}_q = tmp * s + \mathbf{v}_q * c$
11:    end for
12:  end for
13: end while
14: Determine the singular values from the magnitude of the resulting $\mathbf{A}$ matrix, $\mathbf{\Sigma}(i,i) = \|\mathbf{A}(:,j)\|$
15: Determine the left singular vectors from the resulting $\mathbf{A}$ matrix, $\mathbf{U}(:,j) = \frac{\mathbf{A}(:,j)}{\mathbf{\Sigma}(i,i)}$
```

Algorithm 5 Parallel_Jacobi_One_Sided_SVD_on_GPU

```
Require: Assume $\mathbf{A}^{n\times n}$, $\mathbf{V}^{n\times n}$ initially contains the identity matrix.
1: Allocate device global memory for arrays $\mathbf{A}$, $\mathbf{V}$, $\mathbf{U}$ and $\mathbf{\Sigma}$
2: Copy arrays $\mathbf{A}$, $\mathbf{V}$, $\mathbf{U}$ and $\mathbf{\Sigma}$ from the host main memory to the device global memory
3: Perform the Algorithm 4 and call the kernel in Algorithm 6 every time Line 3 has been reached.
4: When the algorithm reaches the convergence, call the kernel for determine the singular values from the magnitude of the resulting $\mathbf{A}$ matrix, $\mathbf{\Sigma}(i,i) = \|\mathbf{A}(:,j)\|$
5: Call the kernel to determine the left singular vectors from the resulting $\mathbf{A}$ matrix, $\mathbf{U}(:,j) = \frac{\mathbf{A}(:,j)}{\mathbf{\Sigma}(i,i)}$
6: Copy arrays $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ from the device main memory to the host global memory
7: Free memory of the GPU
```

Algorithm 6 kernel_Parallel_Jacobi_rotation_on_GPU

```
Require: Assume one-dimensional array $\mathbf{a}[n\times n]$ and $\mathbf{v}[n\times n]$ initially contains the identity matrix, and j
1: $ThreadID \longleftarrow blockIdx.x \times BLOCK\_SIZE + threadIdx.x$ \\ $blockIdx.x$ y $threadIdx.x$ are variables, $BLOCK\_SIZE$ is the number of threads in one $BLOCK$.
2: if $ThreadID < n/2$ then
3:     Each thread is used to compute $c$ and $s$. Multiply $c$ and $s$, and store the elements $a[i]-a[j]$ and $v[i]-v[j]$ and update correspondingly (sequential)
4: end if
```

V. Experimental results and analysis

The experimental work environment is: using an Intel Core i7-3770 CPU, graphics card NVIDIA TESLA K20, and a 64-bit Win7 operating system. Compiling and running is Visual Studio 2012. The memory required to process each matrix is $4 \times N \times N \times 7$. The size of the squared matrices processed and the number of required sweeps are observed in the Table I. The number of iterations for the algorithm to converge depends on the matrix dimension. Then, 10 trials of randomly generated matrices to find convergence with the same number of iterations for each matrix size were performed. To verify that the algorithm performs correctly, the mean squared error metric was considered for the reconstructed matrix.

Algorithm 7 kernel_dynamic_parallelism_on_GPU

```
Require: Assume one-dimensional array $\mathbf{a}[n\times n]$ and $\mathbf{v}[n\times n]$ initially contains the identity matrix, and j
1: $ThreadID \longleftarrow blockIdx.x \times BLOCK\_SIZE + threadIdx.x$ \\ $blockIdx.x$ y $threadIdx.x$ are variables, $BLOCK\_SIZE$ is the number of threads in one $BLOCK$.
2: if $ThreadID < n/2$ then
3:     Each thread is used to compute $c$ and $s$. Multiply $c$ and $s$ and store the elements $a[i]-a[j]$ and $v[i]-v[j]$ and update correspondingly calling the *kernel post-multiply* (algorithm 3)
4: end if
```

The first experiment shows the implementations of the parallel algorithm 4 using global memory, texture memory, surface memory and dynamic parallelism. In such implementation, the texture memory is the fastest of all. With respect to sequential implementation, the execution time of the parallel algorithm decreases as the matrix size increase. The sequential algorithm has a processing time with lineal tendency with respect to matrix size. For 2048 to 4096 matrices, this increase is not significant because the number of iterations required to converge is similar, the calcuation of the matrix 2048 is 17.2 times faster than the sequential algorithm and for 4096 is 20.2 times. All these conclusions can be observed from Fig. 2.

TABLE I
MATRIX SIZES

Size	Sweep
2	1
4	4
8	4
16	5
32	6
64	7
128	9
256	11
512	12
1024	12
2048	15
4096	2

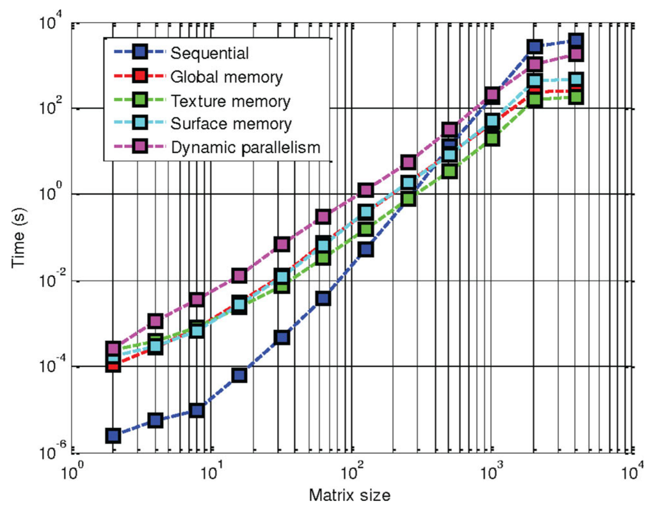

Fig. 2. Matrix size vs processing time for algorithm 5 using different memories.

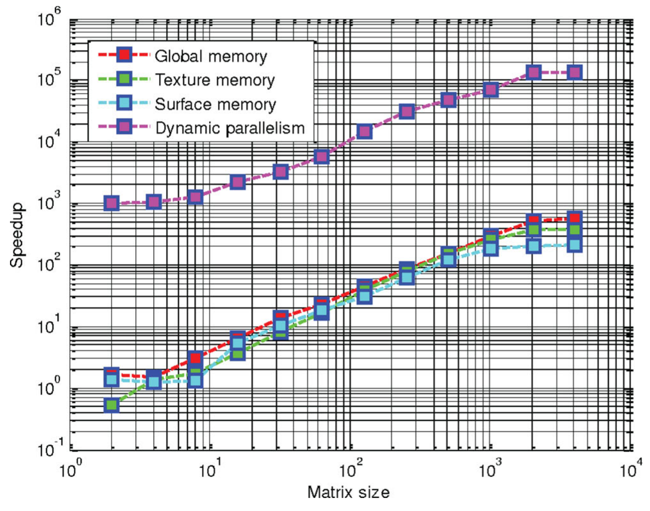

Fig. 3. Matrix size vs speedup for algorithm 5 using different memories.

To measure the improvement of SVD algorithm exposed in Algorithm 4, Fig. 3 shows the speedup results. They are the ratio between the processing times of the algorithm running on one processor and the parallel algorithm with $n/2$ processors. It can be seen that, as the matrix size increases, the speedup ratio increases as well, i.e., the speedup is greater when the matrix size is larger. The implementation using dynamic parallelism has the higher speedup (for a 4096-matrix is 133766). This because within the same kernel, a second kernel is invoked and performed in the same processor. The speedup for a 4096-matrix using global memory is 575.4.

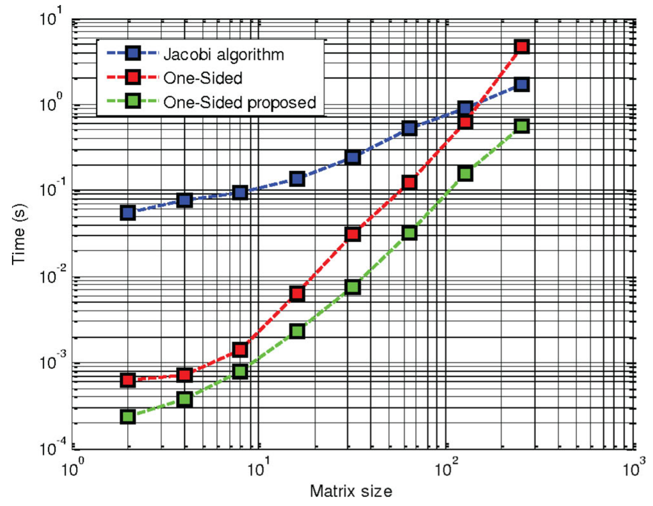

Fig. 4. Processing time comparison.

The One-Sided Jacobi algorithm was implemented in [5] using matrices up to $n = 256$ and it is important to use it as a comparison point because it targets the same calculation. In [22], the Jacobi algorithm was implemented on GPU. Despite this implementation is different, it is important to make a comparison because it uses the same GPU and targets the same calculation. In Fig. 4 appears that the implementation proposal has better processing times that the algorithm in [22]. This corroborates the importance of using the surface memory and texture memory, besides the global memory, in order to have a correct comparison and better implementation.

In Fig. 5 and Fig. 6, the speedup for the Jacobi algorithm in [22] is compared against the speedup of parallel One-Sided Jacobi algorithm proposed. The first one presents the fastest implementations for each algorithm. The Jacobi algorithm uses surface memory while the One-Sided Jacobi algorithm uses the texture memory. The second one compares the two implementations using global memory. The speedup increases for both implementations by using another type of memory. The higher speedup is presented by the parallel One-sided Jacobi algorithm proposed. Some differences between the One-Sided Jacobi and Jacobi algorithm is that the One-Sided can process non-symmetric square matrices and that Jacobi algorithm has a fixed number of iterations defined. For example, for a matrix of $n = 1024$, the number of iterations are about 5000 for the Jacobi algorithm and a minimum of 523776 for One-Sided Jacobi. By knowing that, it is straightforward to think that, for large matrices, the One-Sided Jacobi algorithm is slower than Jacobi algorithm even when it has a better speedup.

The number of iterations depends on the particular convergence criterion. The algorithm and the speedup ratio is strongly related to matrix length and it is very stable. Also, the implementation does not add too computational costs. The processing time of sequential algorithms is clearly outperformed by parallel implementations in the GPU.

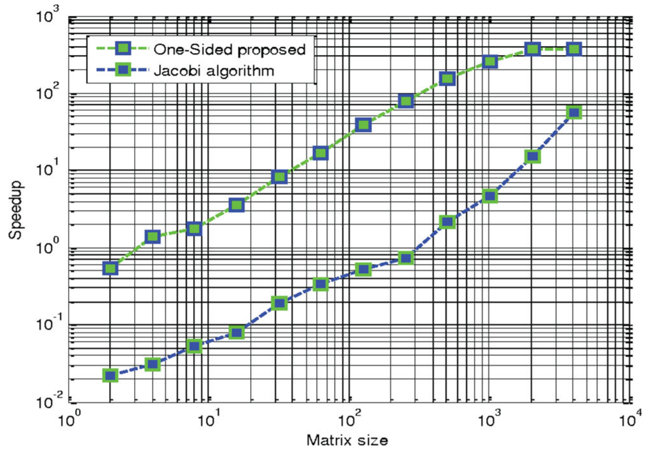

Fig. 5. Speedup comparison between proposed One-Sided algorithm and traditional Jacobi algorithm using texture memory and surface memory.

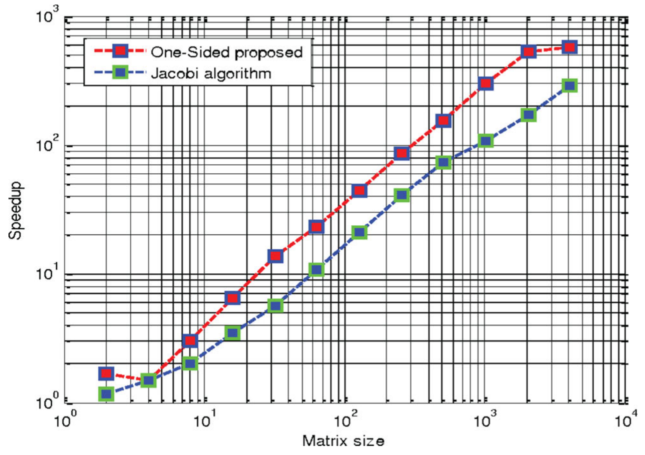

Fig. 6. Speedup comparison between proposed One-Sided algorithm and traditional Jacobi algorithm using global memory.

VI. Conclusions

Results show that the GPU platform is appropriate to implement a high performance SVD algorithm. It is clearly observed that, as the size of the matrix increases, the performance of the GPU implementation improves. For matrix size $n \geq 128$, all GPU implementations outperforms the sequential version, which has its best version using texture memory.

An important thing to notice from this work is that the comparison between different memory usage is important when implementing a parallel algorithm in GPU. This is because there is no a rule of thumb in using the memory hierarchy and this usage is highly related to the algorithm characteristics.

The loop parallelization analysis technique is highly recommended when adequating algorithms to be implemented on GPU because it exposes directly the section that presents data independence and can be processed in parallel. This technique, along with an intuitive loop analysis, leads to the best parallel implementation.

References

[1] P. J. Smith, M. Shafi, and L. M. Garth, "Performance analysis for adaptive mimo svd transmission in a cellular system," in *Communications Theory Workshop, 2006. Proceedings. 7th Australian*. IEEE, 2006, pp. 49–54.

[2] O. Bryt and M. Elad, "Compression of facial images using the k-svd algorithm," *Journal of Visual Communication and Image Representation*, vol. 19, no. 4, pp. 270–282, 2008.

[3] E. S. Ebbini, P.-C. Li, and J. Shen, "A new svd-based optimal inverse filter design for ultrasonic applications," in *Ultrasonics Symposium, 1993. Proceedings., IEEE 1993*. IEEE, 1993, pp. 1187–1190.

[4] D. Zhang, S. Chen, and Z.-H. Zhou, "A new face recognition method based on svd perturbation for single example image per person," *Applied Mathematics and computation*, vol. 163, no. 2, pp. 895–907, 2005.

[5] S. Lahabar and P. Narayanan, "Singular value decomposition on gpu using cuda," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[6] C. Kotas and J. Barhen, "Singular value decomposition utilizing parallel algorithms on graphical processors," in *OCEANS 2011*. IEEE, 2011, pp. 1–7.

[7] T. Wang, L. Guo, G. Li, J. Li, R. Wang, M. Ren, and J. He, "Implementing the jacobi algorithm for solving eigenvalues of symmetric matrices with cuda," in *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*. IEEE, 2012, pp. 69–78.

[8] R. Wang, *Introduction to Orthogonal Transforms: With Applications in Data Processing and Analysis*. Cambridge University Press, 2012.

[9] G. Strang, "The fundamental theorem of linear algebra," *American Mathematical Monthly*, pp. 848–855, 1993.

[10] S. Lee and M. H. Hayes, "Properties of the singular value decomposition for efficient data clustering," *Signal Processing Letters, IEEE*, vol. 11, no. 11, pp. 862–866, 2004.

[11] S. I. Grossman, M. G. Osuna, and F. P. Soto, *Álgebra lineal*. Grupo Editorial Iberoamericana, 1983.

[12] E. Biglieri and K. Yao, "Some properties of singular value decomposition and their applications to digital signal processing," *Signal Processing*, vol. 18, no. 3, pp. 277–289, 1989.

[13] J. J. Gerbrands, "On the relationships between svd, klt and pca," *Pattern recognition*, vol. 14, no. 1, pp. 375–381, 1981.

[14] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[15] R. P. Brent, F. T. Luk, and C. Van Loan, "Computation of the singular value decomposition using mesh-connected processors," *Journal of VLSI and computer systems*, vol. 1, no. 3, pp. 242–270, 1982.

[16] N. Corporation, *CUDA C Programming Guide*, 2013.

[17] ——, *CUDA C Best Practices Guide*, 2012.

[18] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[19] W.-m. Hwu and D. Kirk, "Programming massively parallel processors," *Special Edition*, vol. 92, 2009.

[20] J. Demmel and K. Veselic, "Jacobi's method is more accurate than qr," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 4, pp. 1204–1245, 1992.

[21] U. Banerjee, *Loop Transformation for Restructuring Compiler: The Foundations*. Kluwer, 1994.

[22] L. Lopez, "Desarrollo en software paralelo de librerias matematicas basadas en la transformada de karhunen-loeve y aplicaciones a telecomunicaciones," Master's thesis, CINVESTAV GDL, Dic 2013.