Parallel QR Factorization using Givens Rotations in MPI-CUDA for Multi-GPU

Miguel Tapia-Romero[1], Amilcar Meneses-Viveros[2]
Departamento de Computación
Cinvestav-IPN
Mexico City, Mexico

Erika Hernández-Rubio[3]
Instituto Politécnico Nacional
SEPI-ESCOM
Mexico City, Mexico

***Abstract*—Modern supercomputers incorporate the use of multi-core processors and graphics processing units. Applications running on these computers take advantage of these technologies with scalable programs that work with multicores and accelerator such as graphics processing unit. QR factorization is essential for several numerical tasks, such as linear equations solvers, compute inverse matrix or compute a diagonal matrix, to name a few. There are several factorization algorithm such as LU, Cholesky, Givens and Householder, among others. The efficient parallel implementation of each parallelization algorithm will depend on the structure of the data and the type of parallel architecture used. A common strategy in parallel programming is to break a problem into subproblems to solve them in different processing units. This is very useful when dealing with complex problems or when the data is too large to work with the available memory. However, it is not clear how data partitioning affects subtask performance when mapping to processing units, specifically to graphical processing units. This work explores the partitioning of large symmetric matrix data for QR factorization using Givens rotations and its parallel implementation using MPI and CUDA is presented.**

Keywords*—Givens factorization; CUDA; heterogeneous programming; scalable parallelism

I. Introduction

Every time it is more common to work with large amounts of data. One of the most commonly used tasks in processing these large volumes of data is QR factorization for square matrices. QR factorization is used in processes such as solving linear equations, inverting matrices, and in the process of diagonalizing matrices, to name a few. There are various methods for factoring such as LU, Cholesky, Householder, or Givens. These numerical tasks are a tool of common use in areas such as physics, chemistry and engineering. Also, artificial intelligence papers using Givens rotations on large volumes of data have been reported [1], [2], [3], [4].

Modern supercomputers incorporate the use of multi-core processors and accelerators such as graphics processing units (GPUs). Applications running on these computers take advantage of these technologies with scalable programs that work with multicores and GPUs. Although it has been reported that applications with GPUs can speed up a lot, GPUs suffer from the amount of memory they have available for data management. There are already GPU cards with more than 12GB of memory, however many computers still have cards with 6GB of memory or less, so the use of various GPUs helps resolve this limitation. Modern applications running on supercomputers must be able to take advantage of various architectures that help speed up computing and must have the ability to scale, that is, to work on different nodes.

Methodologies have been proposed to develop programs for the new supercomputers [5], [6]. These methodologies include phases such as partitioning, aggregation, and mapping phases, among others. Partitioning refers to break a problem into subproblems to solve them in different processing units. This is very useful when dealing with complex problems or when the data is too large to work with the available memory. Aggregation refers to grouping subtasks, which is useful when identifying processes that can work with shared memory. The mapping phase refers to the association of tasks with processing units.

These methodologies have assisted in the development of scalable parallel programs and ensure the use of the various types of processing units. However it is not clear how a partitioning strategy can affect performance when tasks are mapped to graphics processing units. This work explores the partitioning of large symmetric matrix data for QR factorization using Givens rotations and its parallel implementation using MPI and CUDA is presented. A single GPU card version is made for comparison and analysis purposes. Its means, the program can use different GPU cards that are on the same node or on different nodes of a computer cluster. For the purpose of studying the different partitioning of the matrix, this work focuses on large symmetric matrices. The results show that the row or column partitioning of the matrix play an important role in the performance of CUDA kernels, also the communication between the main memory and the memory of the GPUs is important. Also changes in nVidia GPU technologies can affect the performance of the application.

This paper is organized as follows. Section 2 present the related work. Section 3 describe the Givens rotation procedure. Section 4 presents the design of the CUDA parallel program of QR factorization using Givens rotations for dense matrices. Section 5 describes scalable parallel implementation prioritizing row partitioning of matrices. Section 6 describe the experiments that were performed. Section 7 discusses the consequences of row and column partitioning on the performance of tasks running on GPUs. Finally, Section 8 presents the conclusions of this work.

II. Related Work

Traditionally the Intel MKL Math Library is used for Givens factorization. This library is highly optimized for the use of multicore processors [7][8][9][10]. Other mathematical libraries for lineal algebra have also been developed of which use parallel implementations such as MAGMA, PLASMA, ViennaCL, Armadillo and dmath, to name a few [11][12][13][14][15]. This libraries are designed for multicore architectures and GPUs. In addition, this work has been ported to Xeon Phi [16] and other library was optimized for multicore [17].

Sameh, in 1978, present a parallel algorithm for solving a system of linear equations using Givens rotations [18]. This work considers a parallel computer with shared memory to solve a dense tridiagonal linear system and it shows that the complexity to solve the tridiagonal system is $O(n)$ steps, compared to $O(n \log n)$ steps reported from previous work through Gaussian elimination [19]. Later, other authors complement Sameh's work to construct the QR factorization with Givens rotations of a dense rectangular matrix [20] or to construct QR factorization using Givens rotations [21].

There are other works where different factorization methods (LU, Cholesky or Householder) are parallelized for multicore architectures or GPUs [22], [23], [24], [25], [26][25], [27], [28]. Some of these works are oriented to the study of communications between processes, optimizing implementations for multi-core architectures, optimizing partitioning for architectures with GPUs, among others. The parallelization strategies vary even with the same factorization method. This is because the strategies depend on the parallel architecture in which the factorization is implemented.

III. Givens Rotation

The main idea in Givens rotations is to rotate a vector to annihilate, or zero, one of its elements. Therefore a rotation matrix is used. Then if two row vectors, u^t and $v^t \in \mathbb{R}^m$, are rotated.

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} \mu_1 & \cdots & \mu_n \\ v_1 & \cdots & v_n \end{pmatrix} = \begin{pmatrix} \mu_1' & \mu_2' & \cdots & \mu_n' \\ 0 & v_2' & \cdots & v_n' \end{pmatrix} \tag{1}$$

The values for v_i' and μ_i' are

$$\begin{array}{rcl} \mu_1' & = & (\mu_1^2 + v_1)^{1/2}, \\ \mu_i' & = & c\mu_i + sv_i, \ \ 2 \leq i \leq m, \\ v_i' & = & -s\mu_i + cv_i, \end{array} \tag{2}$$

and the values c and s of the matriz rotation are

$$\begin{array}{rcl} c & = & \mu_1/\mu_1', \\ s & = & v_1/v_1'. \end{array} \tag{3}$$

As discussed in [18], a sequential program need $n(n-1)/2$ rotations to get an upper triangular matrix from $n \times n$ square matrix .

IV. Algorithm Adaptation for GPU

There are various strategies to harness the computing power of GPUs. The main one is that the programs must be SPMD, work with fine granularity, reduce data transfer between card memory and main memory, and avoid synchronization between threads, among others. In the CUDA programming environment, the CPU and its memory are called a host, and a GPU card is called a device. The part of a program that runs on a GPU is called the kernel. To run a kernel, the GPU memory must have the input data and a memory space to store the results. The part of a program that runs on a GPU is called the kernel. To run a kernel, the GPU memory must have the input data and a memory space to store the results. This implies that information must be moved between the host and the device. When designing a CUDA program, you must be careful with memory management, otherwise you can generate a large overhead.

Matrix factorization is an operation that consumes a lot of memory and CPU time. Selecting a matrix factorization method depends on the type of matrix and the architecture where it will be implemented. In this work, the Givens rotations are used for the QR factorization, since a method that adapts to the shared memory architecture that the GPUs use, and that can also avoid the synchronization of threads with a good implementation.

As explained in [18], when applying Givens rotation to a matrix A to annihilate the a_{ij} element, the rotation matrix affects two rows of a matrix A, rows $i-1$ and i. The result of the rotation makes the jth element of row i zero. It is possible to parallelize the computation of the columns of rows $i-1$ and i, because they are computations that do not generate dependencies between the computations of the columns. So a CUDA thread can be assigned to the calculation of each column.

The implementation that was carried out involves communication between MPI processes and the synchronization of work between the CPU and the GPU, so the implementation is heterogeneous. MPI allows to distribute data between processes and control synchronization with GPUs; and with CUDA computations are performed on the GPU.

Algorithm 1 presents the QR factorization algorithm using Givens rotations in GPU card. Lines 5 and 6 of Algorithm 1 are executed in GPU. The rest of algorithm run in a CPU.

Algorithm 1 QR factorization with Givens rotation

Require: $A \in \mathbb{R}^{n \times n}$, a symmetric square matrix; $I \in \mathbb{R}^{n \times n}$, an identity matrix.
Ensure: $R \in \mathbb{R}^{n \times n}$, an upper triangular matrix; $Q \in \mathbb{R}^{n \times n}$, an orthogonal Matrix.
1: $R \leftarrow A$.
2: $Q \leftarrow I$.
3: Copy R and Q to GPU memory.
4: **for** $i = 0, 1, \ldots, n-1$. **do**
5: To get i column i from R.
6: To apply Givens rotations to i column to R and Q.
7: **end for**

1) To get i column from R: As mentioned, applying a Given rotation affects two lines. Therefore, when the Givens rotation is applied to the i row, you must wait for the $i-1$ row to change all its values in order to apply the rotations to this row. This must be done with a synchronization between the threads involved in the rotation.

In order to avoid synchronization, each thread compute the values c and s, according to equation 3, to carry out the rotations in the corresponding column. Therefore, a copy of the column to which the rotations are applied is stored, this copy is stored in read-only memory for all threads, so each thread can obtain the values of this column and then use them to calculate the c and s.

In order to get and store the i column from R matrix, line 5 in the Algorithm 1, each thread in the kernel gets a value from the i column of R matrix and stores it into array in global memory of the GPU card, this is achieved with the help of thread identifier. So in a single call to this kernel all the values of the required column are stored.

2) To apply Givens rotations to i column to R and Q: Algorithm 2 shows how to apply Givens rotations. From equation 2, it is possible notice that Givens rotation affects 2 rows. This algorithm is implemented in a CUDA kernel. Each thread executes the same process, the only thing that changes is that each thread works on a different column, so at the end of calling the algorithm once 2 we will modify to $0'$s the values of a column under the diagonal main and after to call $n-1$ times the algorithm we will have the upper triangular matrix R and Q^T.

From

$$A = QR \Rightarrow Q^T A = R \Rightarrow Q^T = G_n \dots G_1 I,$$

results in Q^T matrix by applying the Givens rotations to the identity matrix. Because Q is orthogonal, $Q^T = Q^{-1}$.

Algorithm 2 To Compute Givens Rotations Per Column

Require: $R \in \mathbb{R}^{n \times n}$, symmetric square matrix; $Q \in \mathbb{R}^{n \times n}$, identity matrix; $L \in \mathbb{R}^n$, column from R; col, column identifier.

Ensure: R and Q matrices with Givens rotations applied to column col.

1: $i =$ thread identifier.
2: $j = n$.
3: $\mu_i' = \sqrt{{l_{j-1}}^2 + {l_j}^2}$.
4: $c = \frac{l_{j-1}}{\mu_i'}$ y $s = \frac{l_j}{\mu_i'}$.
5: **while** $j > col$ **do**
6: $\mu_i = r_{i,j-1}$ y $\nu_i = r_{i,j}$.
7: $\alpha_i = q_{i,j-1}$ y $\beta_i = q_{i,j}$.
8: $r_{i,j-1} = c\mu_i + s\nu_i$ y $r_{i,j} = -s\mu_i + c\nu_i$.
9: $q_{i,j-1} = c\alpha_i + s\beta_i$ y $q_{i,j} = -s\alpha_i + c\beta_i$.
10: $j = j - 1$.
11: $a = \mu_i'$.
12: $\mu_i' = \sqrt{{l_{j-1}}^2 + a^2}$.
13: $c = \frac{l_{j-1}}{\mu_i'}$ y $s = \frac{a}{\mu_i'}$.
14: **end while**

Fig. 1 shows how algorithm 2 affects a matrix. So, this algorithm needs to be applied to the first $n-1$ columns of a matrix to get the matrices R and Q^T, Fig. 2.

From Fig. 1 and 2, it can be seen that, as the process progresses, there are some threads that are left unworked, and this happens for the last steps. But when the number of threads is less than the number of columns in the matrix, the threads

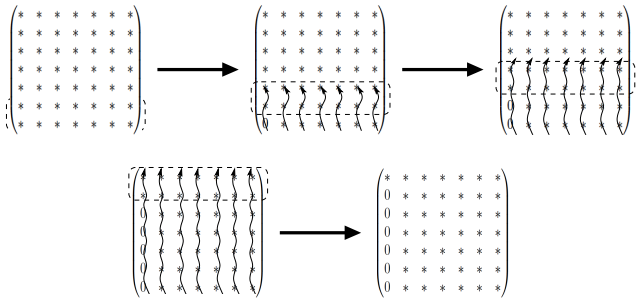

Fig. 1. Givens Rotations to the First Column.

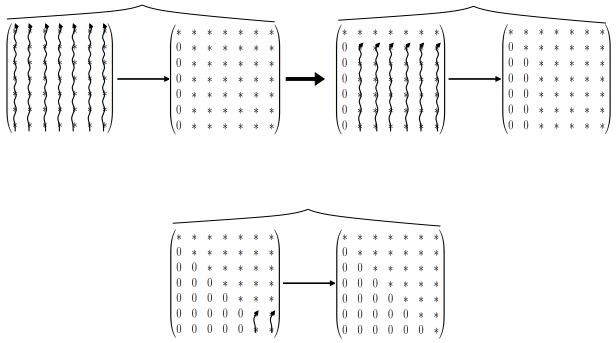

Fig. 2. Process to get R, an Upper Triangular Matrix with Givens Rotations.

are traversed. Although interesting results have been shown when the number of threads is $n-1-col$, and although there are several threads that do few computations, the results in time are better, since CUDA threads are kept in fine grain.

V. Multi GPU Strategy

One of the main restrictions on the use of GPUs is memory. Many GPU cards have 8GB of memory or less. While nVidia has provided memory sharing strategies across multiple GPUs or directly communicate the main memory with the GPU memory, these solutions can be expensive for many users. One strategy could the use multiple stream to get an speedup and use short pieces of data to process, but this strategy is not always easy to implement in many problems.

To work with multiple GPUs, a strategy for partitioning data is designed. A strategy would be to divide the columns so that they are processed by different cards, as shown in Fig. 3. Because matrices are stored by row in C, this partitioning method generates an overhead. Since the input matrix A in Algorithm 1 is symmetric, then the partitioning of A by rows can be performed 3. This matrix partitioning makes the communication between the processes that handle these sub-matrices more efficient. And when applying Givens rotations the R^T matrix is obtained.

In the implementation of this work, it was decided to use MPI with CUDA, so each MPI process is in charge of communicating with the GPU card, and the partitioning of the data is done by lines, instead of columns. Thus, each MPI process has a GPU card assigned to execute CUDA processes. The main MPI process, or root process, is in charge

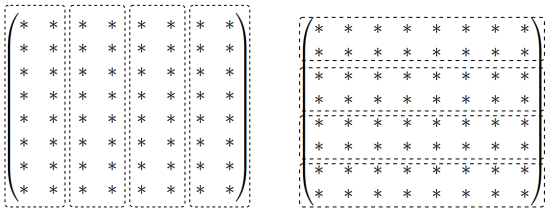

Fig. 3. Column and Row Partitioning of Data.

of doing the read operations and doing a scatter operation of the matrix A to the other processes. Subsequently, the root MPI process send to the other processes the row of A, to which Givens rotations will be applied. Each MPI process sends to the memory of the GPU card that has assigned the sub-matrix A, the row of A, and sends to execute the kernel of the Givens rotations. The MPI process must send the first $n-1$ lines among them, one line at a time, to each MPI process so that they apply the rotation to the sub-matrices it has. Every time an MPI process receives a row from the root process, it must upload it to the GPU memory and run the Givens rotation kernel. At the end each MPI process will have a partial result of the partial triangular matrix R^T. If each MPI process initializes the corresponding identity sub-matrix, a communication between the root process and the other MPI processes is avoided, and the corresponding Givens rotation is applied to it, at the end an Q submatrix is also obtained. Finally, the data of each process must be gathered in a single data set to the root MPI process, to be able to show the results, in this case the Q and R^T matrices.

In Algorithm 3 is shown the parallel QR factorization using MPI processes for multi GPU.

In Algorithm 3, there are some lines that deserve a review. In the line 8, select the GPU card that will be assigned to each process, the CUDA function to select the card is `cudaSetDevice()`. The assignment of the card will depend on the hardware configuration of each node. For example, if we have nodes with a multicore processor with two GPU cards, each node associates 0 and 1 with the identifiers of the cards. If MPI processes $i, i+1$, with $i = 0, 1, \ldots, np$, are assigned on each node. Then a module operation is used to assign the GPU card.

$$idGPU = (idProcess) \mod 2.$$

Where 2 is the number of GPU cards.

The line 9, refers to the row partitioning of the matrices R and Q. Thus, each MPI process corresponds to sub-matrices $R, Q \in \mathbb{R}^{(n/np)\times n}$ with the same number of lines. That is, each MPI process is sent $(n * n)/np$ elements to be processed. For this task, `MPI_Scatter` function is used. The `MPI_Scatter` function is used to divide an array of data into equal parts between the different processes.

Line 21 of the Algorithm 3 gathers the partial results of each MPI process to form the resulting R^T and Q matrices. To do this, the `MPI_Gather` function is used.

In the following sections, other lines of Algorithm 3 will be described in more detail. These lines are: 12, 13 and 13.

Algorithm 3 QR Factorization using Givens Rotations for Multi GPUs

Require: $A \in \mathbb{R}^{n\times n}$, a symmetric square matrix ;np = total number of process.
Ensure: $R^T \in \mathbb{R}^{n\times n}$ transposed upper triangular matrix; $Q \in \mathbb{R}^{n\times n}$ orthogonal matrix.
1: Start MPI.
2: np = total number of process.
3: $idProcess$ = process number.
4: **if** $idProcess == 0$ **then**
5: $R \leftarrow A$.
6: $Q \leftarrow I$.
7: **end if**
8: Associate a GPU card with the MPI process.
9: MPI Scatter of R and Q to all processes.
10: To copy the sub-matrices of R and Q to the memory of the GPUs.
11: **for** $i = 0, 1, \ldots, n-2$. **do**
12: **if** Row i belongs to the R sub matrix of the process $idProcess$ **then**
13: To get the i row from R sub matrix of the $idProcess$ process.
14: **end if**
15: $idProcess$ process sends the row obtained to the other MPI processes.
16: Wait until all processes have received the row.
17: Copy the row to the memory of the GPUs card.
18: To apply Givens rotations to sub-matrices R and Q.
19: **end for**
20: Copy the R and Q sub-matrices from the GPUs card to the Host.
21: Join submatrices to get R^T y Q.
22: Finalize MPI.

Lines 12 and 13 show how to obtain the necessary row of a specific process. And the line 18 Givens rotations are applied to the matrix partition of each process.

A. To Get the i-th Row of the Process that has the Sub Matrix

As discussed in the previous sections, it is required to avoid synchronization between processes to achieve performance. To avoid this, a copy the row to which the rotations are applied to each GPU card is made. It is important to extract the line that is being processed. It must be inferred which process is the one that contains the line to be processed. The Algorithm 4 is responsible for obtaining the row and sending them to the MPI processes. This algorithm is used in lines 12 and 15 of Algorithm 3.

In Algorithm 4, the i is the row that is send to all processes, k is the partition that has the row, d is the number of rows that each process has, n is the number of rows that the original matrix has, np is the number of MPI processes, this value is equal to the number of partitions that were made from the matrix, and $idProcess$ is the identifier of each process.

This process runs n times, as seen in the Algorithm 3, where j and k are initialized to 0 before starting the `for` cycle, it is possible yo obtain the row at the time it is required and assures us to go through all the rows.

Algorithm 4 To Get the i-th Row of R Sub Matrix from MPI Process.

Require: i, i-th row; k is the process with the i row; $d = n/np$m number of rows; $idProcess$, process identifier.
1: **if** $idProcess == k$ **then**
2: To get i row of sub matrix R from $idProcess$.
3: **end if**
4: To send the row from $idProcess$ to the other process.
5: $i = i + 1$
6: **if** $i == d$ **then**
7: $i = 0$
8: $k = k + 1$.
9: **end if**

B. To Apply Givens Rotations to Row i with Multi-GPUs

Because rotations are applied to row i of A matrix, this operation affects rows $i + 1$ to $n - 1$ of this matrix. Since each process has this row, the sub-matrices of R and Q can be modified.

The algorithm is similar to Algorithm 2, although it has some changes to be able to apply the rotations to each partition. In Algorithm 5 shows the steps that must be applied to each sub matrix of R and Q to get the partition with the rotations applied.

Algorithm 5 To Compute Givens Rotations Per Row on Each GPU Card

Require: $R, Q \in \mathbb{R}^{(n/np)\times n}$, sub matrices; $L \in \mathbb{R}^n$, is the i-row; fil, is the row identifier; n, matrix column size; np, number of process.
Ensure: $R, Q \in \mathbb{R}^{(n/np)\times n}$, sub matrices with Givens rotations applied;
1: $i =$ thread identifier.
2: $j = n$.
3: **while** l_j sea 0 **do**
4: $j = j - 1$
5: **end while**
6: $tempj = j$
7: **while** $i < n/np$ **do**
8: $\mu'_i = \sqrt{{l_{j-1}}^2 + {l_j}^2}$.
9: $c = \frac{l_{j-1}}{\mu'_i}$ y $s = \frac{l_j}{\mu'_i}$.
10: **while** $j > fil$ **do**
11: $\mu_i = r_{i,j-1}$ y $\nu_i = r_{i,j}$.
12: $\alpha_i = q_{i,j-1}$ y $\beta_i = q_{i,j}$.
13: $r_{i,j-1} = c\mu_i + s\nu_i$ y $r_{i,j} = -s\mu_i + c\nu_i$.
14: $q_{i,j-1} = c\alpha_i + s\beta_i$ y $q_{i,j} = -s\alpha_i + c\beta_i$.
15: $j = j - 1$.
16: $a = \mu'_i$.
17: $\mu'_i = \sqrt{{l_{j-1}}^2 + a^2}$.
18: $c = \frac{l_{j-1}}{\mu'_i}$ y $s = \frac{a}{\mu'_i}$.
19: **end while**
20: $j = tempj$
21: increase i to block size
22: **end while**

Every CUDA thread work with a specific row. If the number of threads is less than the number of lines of the sub-matrices, then the thread identifier is increased with the total of CUDA threads that have been requested per MPI process. Because CUDA threads traverse the sub matrix per row, using a moderate number of threads is not recommended to avoid the sparse cache problem. It was observed that an adequate number of CUDA threads is 128 threads per streaming multiprocessor in GPU cards.

A CUDA kernel runs the Algorithm 5. This kernel runs on every GPU card associated with an MPI process. The key for this algorithm to work is to have the row to which the rotations in memory will be applied, since without it the algorithm would not work, or it only work on the GPU card where the row data is kept. Fig. 4 shows how the algorithm works on the sub matrices.

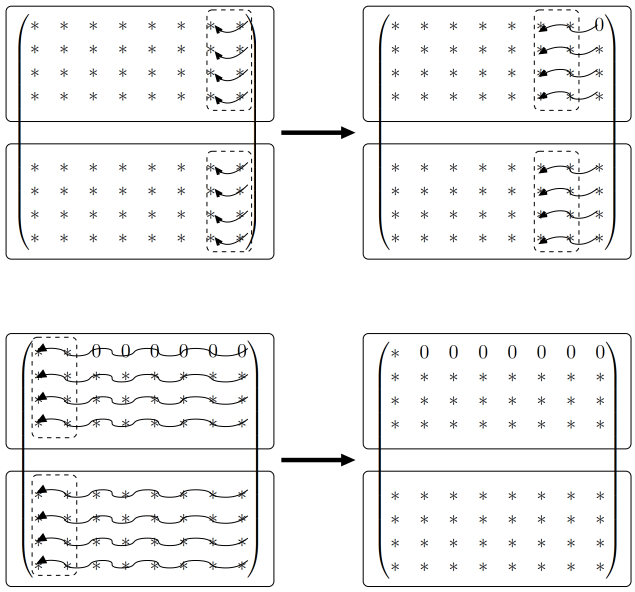

Fig. 4. Kernel Running on 2 GPU Cards.

It is known that although the cards run the same kernel, it does not imply that they end at the same time. So MPI processes must wait for all cards to finish executing the kernel that applies rotation to row i before moving on to row $i + 1$. This is accomplished by synchronizing the MPI processes with a call to the `MPI_Barrier` function.

As in the method for a GPU card, in the method of Givens multi GPUs, each card must make $n - 1$ calls to the kernel. Fig. 5 shows how the whole method works using two GPU cards.

Fig. 5 shows that after half the process, it seems that one of the cards, the one with the upper part of the matrix, no longer performs any operation; but this is not the case, although it does not work on the partition of R, on the partition of Q it does, since each call to the kernel changes the partition of Q. Fig. 6 shows how even after the middle of the process the two cards continue to work.

Like a parallel method that works on a single card, for the multi-GPU method, if the number of rows in each partition is greater than the number of threads, then the threads start traversing to completely cover the partitions. Thus, it is not

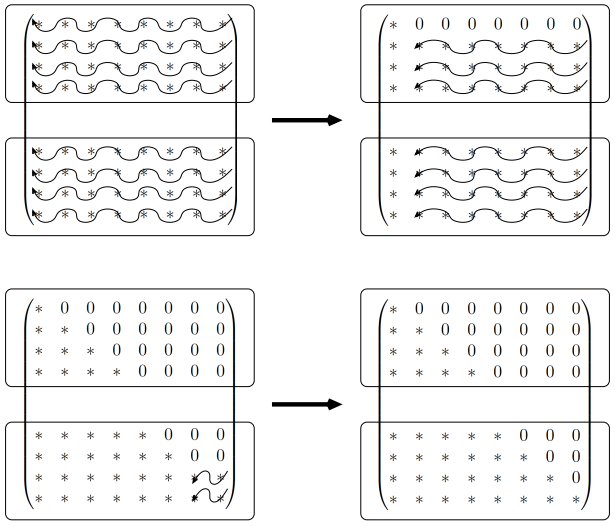

Fig. 5. Givens Method for 2 GPU Cards Working in the R Matrix.

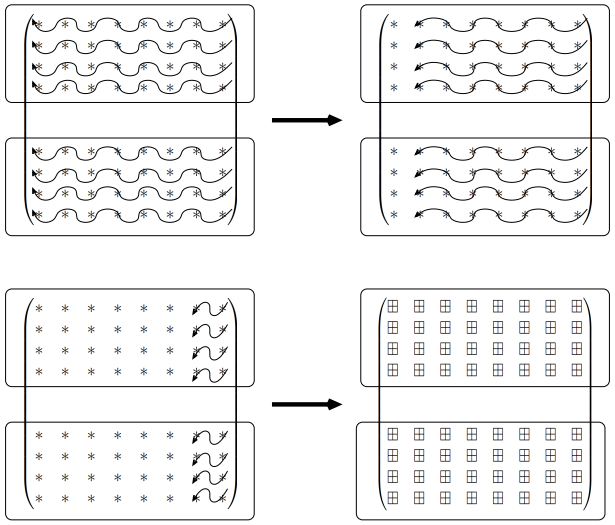

Fig. 6. Givens Method for 2 GPU Cards Working in the Q matrix.

necessary to have the same number of threads as rows per partition, this takes on importance since it can be the case that the two cards are different, so there is no need to change the number of threads in the program execution. In addition to avoid scattered cache problems, it is recommended to have 128 threads per block.

VI. Tests

To test the performance of the multi GPU parallelization of the QR factorization, it was compared against Intel Math Kernel Library (MKL) functions. These comparisons were made with single-GPU and multi-GPU experiments.

The hardware used for testing was as follows:

- Server with two **Intel Xeon X5675** [29] to run MKL. Each process has::
  - Number of CPU core: 6. Clock Speed: 3.06 GHz. Memory bandwidth: 32 GB/s.
- Three different GPU cards were used for testing with CUDA:
  - 2 **NVIDIA Tesla K20X** card [30]:
    - Memory size: 6 GB. CUDA Cores: 2688. Clock speed per core: 732 MHz. Memory Bandwidth: 250 GB/s.
  - 1 **NVIDIA Tesla C2070** [31]:
    - Memory size: 6 GB. CUDA Cores: 448. Clocl speed per core: 1.15 GHz. Memory Bandwidth: 144 GB/s.
  - 1 **NVIDIA GeForce GTX 460** [32]:
    - Memory size: 1 GB. CUDA Cores: 336. Clock speed per core: 1.53 GHz. Memory Bandwidth: 115.2 GB/s. Cuenta con *overclocking*

A. Experiments with a Single GPU

The parallel implementation of the QR factorization for multi GPU was executed in each one of the cards. The 2 MKL functions were used: `?geqrf`[1], which allows to get the matrix R^T and the function `?orgqr`, which is used after the function `?geqrf`, to get the matrix Q^T. In CUDA and MKL tests, single and double precision were used.

Three different types of test matrices were used: tridiagonal, pentadiagonal, and heptadiagonal. The results of the three types of matrices were very similar, since the algorithm considers symmetric dense matrices, so only the results of the tests with heptadiagonal matrices will be exposed. One of the objectives of this work is to be able to process large matrices. The experiments were carried out with single precision square matrices of size 5000×5000 to 20000×20000, and 5000×5000 to 16000×16000 for double precision square matrices.

Fig. 7 shows the difference in execution times of the different tests that were carried out for simple precision. And the Table I shows the data in more detail for your study. These first experiments are for QR factorization with a single GPU card. In general, it is appreciated that the MKL functions for QR factorization are very competitive up to sizes between 9000×9000 or 10000×10000.

Table I shows the execution times. Because the MKL experiments with 12 threads have better performances, these times will be taken as a reference for CUDA experiments. The K20X card, which is the one with the lowest performance among GPU cards. The test on the K20X has a maximum acceleration of 1.2x over MKL with 12 threads when working with an matrix size of 20000×20000. The GTX 460 card could only process matrices up to 10000×10000 in single precision because it only has 1MB of memory, however, when processing this matrix size it achieves a 1.36x acceleration compared to MKL. The C2070 card begins to show better performance compared to MKL from the matrix size of 10000×10000 with a 1.2x acceleration and achieves an acceleration of 2.8x at the size 20000×20000, which is the maximum it supports.

[1]Where the symbol `?` changes for a `s` if it is a single precision or a `d` if it is a double precision.

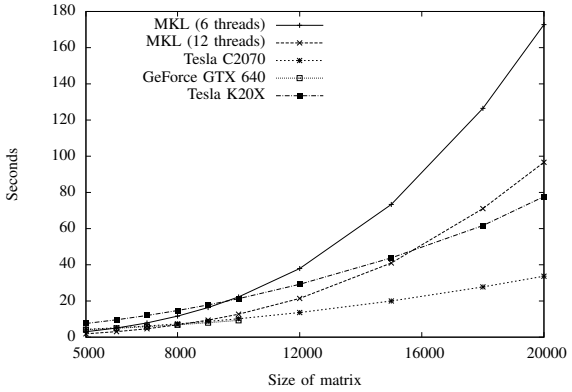

Fig. 7. Execution Times of QR Factorization for Single Precision Matrices.

TABLE I. EXECUTION TIMES OF QR FACTORIZATION FOR SINGLE PRECISION MATRICES.

	Execution times (seconds)				
Size	**MKL 6 threads**	**MKL 12 threads**	**C2070**	**GTX 640**	**K20X**
5000	3.005	1.854	4.315	4.127	7.567
6000	5.057	3.054	5.181	4.893	9.583
7000	7.887	4.664	6.175	5.775	11.967
8000	11.621	6.795	7.345	6.823	14.708
9000	16.361	9.428	8.689	7.991	17.812
10000	22.249	12.701	10.153	9.289	21.251
12000	37.928	21.379	13.598	-	29.217
15000	73.342	40.967	19.986	-	43.838
18000	126.449	71.037	27.771	-	61.680
20000	172.728	96.641	33.677	-	77.676

Fig. 8 shows the execution times of the different tests with double precision with MKL and CUDA with a single GPU card. Like the simple precision case, MKL experiments with 12 threads perform better than execution with 6 threads, so comparisons of the CUDA implementation will be made with the results of this experiment.

Table II shown the performance of CUDA programs with respect to the MKL version. The maximum size for GTX 640 card es 8000×8000, this card begins to show an acceleration of 1.1x in the size of 6000×6000 and reaches 1.7x in the

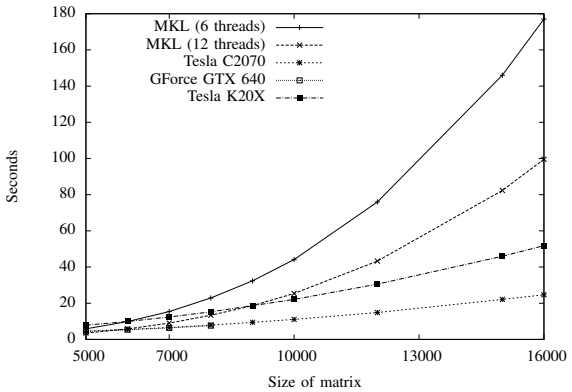

Fig. 8. Execution Times of QR Factorization for Double Precision Matrices.

TABLE II. EXECUTION TIMES OF QR FACTORIZATION FOR DOUBLE PRECISION MATRICES.

	Execution times (seconds)				
Size	**MKL 6 threads**	**MKL 12 threads**	**C2070**	**GTX 640**	**K20X**
5000	5.856	3.471	4.532	4.405	7.886
6000	9.863	5.783	5.507	5.333	9.893
7000	15.407	9.001	6.592	6.354	12.401
8000	22.894	13.338	7.882	7.599	15.271
9000	32.359	18.678	9.449	-	18.640
10000	44.139	25.375	11.043	-	22.124
12000	75.976	43.294	14.861	-	30.562
15000	146.037	82.342	22.118	-	45.949
16000	177.032	99.588	24.683	-	51.798

TABLE III. EXECUTION TIME OF QR FACTORIZATION WITH 2 K20X CARDS FOR SINGLE PRECISION SQUARE MATRICES.

	Execution times (seconds)		
Size	**MKL 6 threads**	**MKL 12 threads**	**2 K20X**
10000	22.24998125	12.70196825	38.5489915
15000	73.34226025	40.967238	84.951077
20000	172.7288738	96.6412415	150.7447375
22000	229.2751283	129.0128638	182.637326
24000	296.6546478	165.9274763	217.4389965
26000	376.5570043	210.0910878	254.3220125

maximum size of the card. The C2070 card presents better performance than MKL from 7000×7000 matrix size, with 1.3x, and reaches 4.14x acceleration compared to MKL. The K20X starts its acceleration in the size of $10000x10000$ with 1.15x and reaches almost 2x speedup. Again, the C2070 performs better than the K20.

Finally, a very peculiar behavior is observed in Table I and II tables, since the execution times are maintained for matrix sizes from 5000 to 1000 for double and single precision cases.

B. Experiments with Multi-GPU

Two experiments were carried out on the scalable implementation of GPU cards: One using 2 Tesla K20X cards and the other using a Tesla C2070 card and a GTX 640. The results of these experiments are compared against MKL of 6 and 12 threads. The first configuration of cards is homogeneous, since they are two Tesla K20 cards. Because K20 cards have 6GB of memory, its possible work with 26000×26000 single precision square matrix for the experiments. The second configuration is heterogeneous, since they are cards with different characteristics. In the case of the experiments with the C2070 and the GTX 460x, it will only be possible to carry out experiments on stable matrices of sizes 6000 to 18000. This limitation in the size of the matrix is due to the memory restriction of the GTX 460 card, since it only has 1GB of memory.

It start with the experiments of the 2 K20X, in the Fig. 9 and Table III it can be see the times for MKL and the implementation proposed in this work to perform the QR factorization. In this the MKL experiments for 12 threads have better performance that the multi-GPU program. However, the multi GPU program is capable of processing a larger matrix than its single card version.

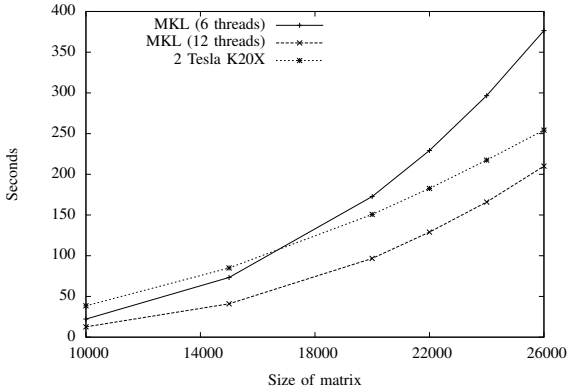

Fig. 9. Execution Time of QR Factorization with 2 K20X Gpus Cards for Single Precision Matrices.

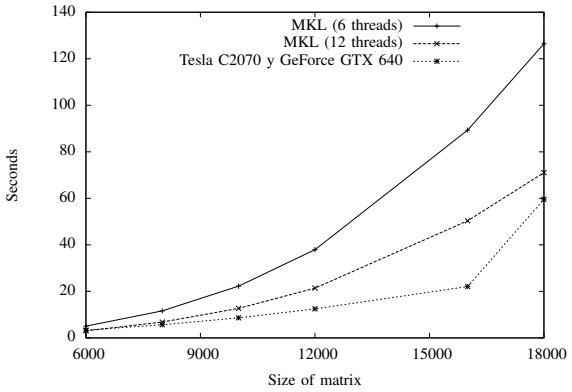

Fig. 10. Execution Times for QR Factorization with Multi GPU using C2070 and GTX 460 Cards for Single Precision Matrices.

Unfortunately for the configuration with the K20X cards the same behavior is maintained in single and double precision. In both cases the performance of the mutli-GPU version is lower than that of MKL with 12 threads.

For multi-GPU experiments with the C2070 and GTX 640, we started showing the results for single precision matrices. In this case, the maximum size of the matrix used to process was 18000×18000, due to the memory limitation of the GTX 460 card. Fig. 10 presents the times of the MKL experiments and the multi GPU implementation. It is interesting to see that the same multi-GPU program has better performance than running for 2 K20X and better performance than MKL functions with 12 threads.

From the values in the Table IV, it is observed that the multi GPU program with the C2070 and GTX 640 cards, starts with similar times to the program with MKL and 12 threads, and accelerates until reaching 1.2x acceleration for the maximum size of the test matrix.

Fig. 11 shows the run times for experiments in the case of matrices with double precision. For these experiments the maximum size of the test matrices was 12000×12000. It is observed that the performance of the version of the QR multi GPU factorization is superior to the version of MKL

TABLE IV. EXECUTION TIMES FOR QR FACTORIZATION WITH MULTI GPU USING C2070 AND GTX 460 CARDS FOR SINGLE PRECISION MATRICES.

	Execution times (seconds)		
Size	**MKL 6 threads**	**MKL 12 threads**	**C2070 y GTX 640**
6000	5.05746	3.05409	3.343184
8000	11.62196075	6.79531825	5.625851
10000	22.24998125	12.70196825	8.625267
12000	37.928559	21.37907675	12.4785045
16000	89.31594025	50.3303555	22.0926865
18000	126.449971	71.03795625	59.6142455

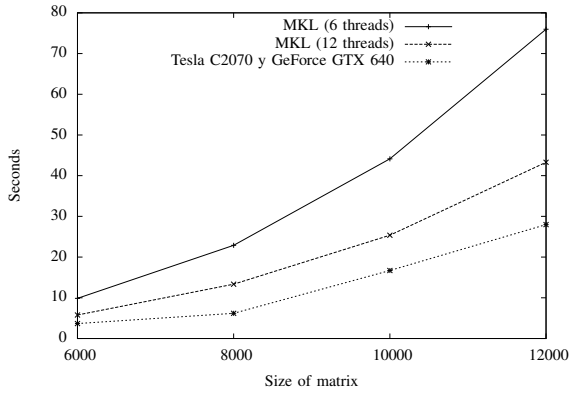

Fig. 11. Execution times for QR factorization with multi GPU using C2070 and GTX 460 cards for double precision matrices.

with 12 threads. Table V shows that the multi GPU program that uses the C2070 cards and the GTX 640, has better performance since the first experiments, with a performance of 1.6x compared to the MKL program with 12 threads. For the experiment with the maximum size of the matrices, the acceleration reached 1.6x over the version of MKL with 12 threads.

VII. DISCUSSION

After carrying out the experiments and comparing the results, its possible analyze the behavior of the programs, including the technologies that were used in this work. This gives a set of lessons learned.

First it can be see that the performance of the Tesla K20X cards was lower than that of the Fermi Tesla C2070 card. When analyzing the execution of the CUDA program on these cards, it is appreciated that the communication time between the Host and Device (both to send data to the GPU and to receive data from the GPU) increased with the K20X cards.

TABLE V. EXECUTION TIMES FOR QR FACTORIZATION WITH MULTI GPU USING C2070 AND GTX 460 CARDS FOR DOUBLE PRECISION MATRICES.

	Execution times (seconds)		
Size	**MKL 6 threads**	**MKL 12 threads**	**C2070 y GTX 640**
6000	9.86363625	5.7834095	3.6662875
8000	22.8946235	13.33819875	6.1803205
10000	44.13997175	25.37548475	16.696302
12000	75.97651125	43.29401675	27.9818585

The kernel performance is superior in the K20X, however, the main bottleneck when using these cards is to pass data between the host and the device.

It is observed that the performance behavior for double precision and single precision data processing is similar with CUDA. So the limitation of the cards is their memory size.

In this paper, symmetric matrices were deliberately used to study the partitioning of rows by matrices to apply Givens rotations. This type of partitioning allows the distribution of data to be very transparent in MPI and thus avoid a bottleneck in communication between MPI processes. However, this results in CUDA threads having to traverse the matrix by row, which generates a sparse cache problem. That is, the cache of the Streaming multiprocessor of the Nvidia GPUs is 46KB. By having multiple threads that must traverse the rows in reverse, there is no guarantee that the data for a CUDA thread will be available in the L1 cache. Which generates a poor performance in the execution of the kernel.In the case of the CUDA program with a card, the path of the CUDA threads is made per column, and thus each thread processes an element of the i-th row and a compact cache is generated, so there is a guarantee of the CUDA threads have the data with which they will operate in the L1 cache memory.

The effect of the use of the compact and dispersed cache is very noticeable in the experiments that were carried out. If the execution times of the CUDA programs of a card are compared with the CUDA multi GPU program, Table II and Table V, it is noted that the performance of the programs with a card is superior to that of the multi GPU version. At first glance one might think that it is due to the synchronization between the processes to send the i-th row to all the processes. But it has been observed that the execution of the multi GPU program with a single GPU consumes more than twice the time of the execution of the program designed for a single GPU.

A new version of the multi GPU program for QR factorization using Givens rotations must have a matrix partitioning per column. This approach would allow CUDA threads to work with compact cache memory, which would give CUDA kernel execution a good performance. Fortunately, there are ways to do a matrix scatter per column in MPI.

VIII. Conclusions

This work presents the parallel implementation in CUDA for Givens factorization. This implementation is scalable to work with multiple GPUs when combined with MPI and CUDA. This work explore some strategies for working with large volumes of data combining MPI with CUDA. These strategies focus on partitioning the matrix to be factored. It can be seen that the program with escalation has less performance than the program for a card. But the performance is due to the way CUDA threads work with sub-matrices derived from row partitioning.

Communication between CPU memory and GPU card affects the performance of CUDA programs. The C2070 card, although it is from a previous generation to the K20X, presents better results than the K20X, because the data transfer times are shorter.

The main contribution of this work is that, for the QR factorization algorithm through Gives rotations, the relationship between matrix partitioning and the scattered or compact cache problems that occur at the GPU level by the how the CUDA kernel should process the sub array. In addition, it is proven that it is feasible to establish a combination of MPI and CUDA to have scalable algorithms that process large volumes of data on nodes that do not include the latest memory management technology from NVIDIA such as GPU Direct.

Acknowledgment

The authors thank financial support given by the Mexican National Council of Science and Technology (CONACyT), as well as IPN for SIP Research Project number 20201079. The authors acknowledge both, the Center for Research and Advance Studies of the National Polytechnic Institute (CINVESTAV- IPN) and the Section of Research and Graduate Studies (SEPI) of ESCOM-IPN, for encouragement and facilities provided to accomplish this publication.

References

[1] B. Alipourfard and J. X. Gao, "Solving all regression models for learning gaussian networks using givens rotations," *arXiv preprint arXiv:1901.07643*, 2019.

[2] Y. Gan, B. Hu, W. Liu, S. Wang, G. Zhang, X. Feng, and D. Wen, "Endmember extraction from hyperspectral imagery based on qr factorisation using givens rotations," *IET Image Processing*, vol. 13, no. 2, pp. 332–343, 2018.

[3] L. Marcellino and G. Navarra, "A gpu-accelerated svd algorithm, based on qr factorization and givens rotations, for dwi denoising," in *2016 12th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*. IEEE, 2016, pp. 699–704.

[4] C. Luo, K. Zhang, S. Salinas, and P. Li, "Efficient privacy-preserving outsourcing of large-scale qr factorization," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 917–924.

[5] P. Czarnul, J. Proficz, and K. Drypczewski, "Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems," *Scientific Programming*, vol. 2020, 2020.

[6] A. A. Serrano-Rubio, A. Meneses-Viveros, G. B. Morales-Luna, and M. Paredes-López, "Generic methodology for the design of parallel algorithms based on pattern languages," in *International Conference on Supercomputing in Mexico*. Springer, 2018, pp. 35–48.

[7] I. Burylov, M. Chuvelev, B. Greer, G. Henry, S. Kuznetsov, and B. Sabanin, "Intel performance libraries: Multi-core-ready software for numeric-intensive computation." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[8] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi?* Springer, 2014, pp. 167–188.

[9] A. Kalinkin, A. Anders, and R. Anders, "Intel® math kernel library parallel direct sparse solver for clusters," in *EAGE Workshop on High Performance Computing for Upstream*. European Association of Geoscientists & Engineers, 2014, pp. cp–426.

[10] Intel, "Math kernel library (mkl)," 2016. [Online]. Available: http://software.intel.com/en-us/intel-mkl/

[11] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[12] A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra, "Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015, pp. 1–6.

[13] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jungel, and S. Selberherr, "Viennacl—linear algebra library for multi-and many-core architectures," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S412–S439, 2016.

[14] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016.

[15] S. Eliuk, C. Upright, and A. Skjellum, "dmath: A scalable linear algebra and math library for heterogeneous gp-gpu architectures," *arXiv preprint arXiv:1604.01416*, 2016.

[16] J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov, "Portable hpc programming on intel many-integrated-core hardware with magma port to xeon phi," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 571–581.

[17] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. YarKhan, M. Abalenkovs, N. Bagherpour *et al.*, "Plasma: Parallel linear algebra software for multicore using openmp," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 2, pp. 1–35, 2019.

[18] A. H. Sameh and D. J. Kuck, "On stable parallel linear system solvers," *Journal of the ACM (JACM)*, vol. 25, no. 1, pp. 81–91, 1978.

[19] ——, "A parallel qr algorithm for symmetric tridiagonal matrices," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 147–153, 1977.

[20] M. Cosnard, J.-M. Muller, and Y. Robert, "Parallel qr decomposition of a rectangular matrix," *Numerische Mathematik*, vol. 48, no. 2, pp. 239–249, 1986.

[21] I. C. Ipsen, "A parallel qr method using fast givens' rotations." YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, Tech. Rep., 1984.

[22] H. Xu and W. Alexander, "Parallel qr factorization on a block data flow architecture," in *The 24th Southeastern Symposium on System Theory and The 3rd Annual Symposium on Communications, Signal Processing Expert Systems, and ASIC VLSI Design*. IEEE, 1992, pp. 332–336.

[23] J. Dongarra, M. Faverge, T. Herault, M. Jacquelin, J. Langou, and Y. Robert, "Hierarchical qr factorization algorithms for multi-core clusters," *Parallel Computing*, vol. 39, no. 4-5, pp. 212–232, 2013.

[24] T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa, and Y. Yamamoto, "Choleskyqr2: a simple and communication-avoiding algorithm for computing a tall-skinny qr factorization on a large-scale parallel system," in *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. IEEE, 2014, pp. 31–38.

[25] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse qr factorization on the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 2, pp. 1–29, 2017.

[26] R. Andrew and N. Dingle, "Implementing qr factorization updating algorithms on gpus," *Parallel Computing*, vol. 40, no. 7, pp. 161–172, 2014.

[27] C. Coti, "Scalable, robust, fault-tolerant parallel qr factorization," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. IEEE, 2016, pp. 626–633.

[28] A. Buttari, S. Hauberg, and C. Kodsi, "Parallel qr factorization of block-tridiagonal matrices," 2019.

[29] I. Corporation, "Intel xeon processor x5680." [Online]. Available: http://ark.intel.com/ products/47916/.

[30] N. Corporation, "Tesla k20x gpu accelerator," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf.

[31] ——, "Tesla c2075 and tesla c2070 computing processor board," 2010.

[32] ——, "Nvidia geforce gtx 460," 2011.