Reduction of a Symmetrical Matrix to Tridiagonal Form on GPUs

By

Shuotian Chen

Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign

Adviser:

Professor Volodymyr Kindratenko

Abstract

Many eigenvalue and eigenvector algorithms begin with reducing the input matrix into a tridiagonal form. A tridiagonal matrix is a matrix that has non-zero elements only on its main diagonal, and the two diagonals directly adjacent to it. Reducing a matrix to a tridiagonal form is an iterative process which uses Jacobi rotations to reduce matrix elements to zero. The purpose of this research project is to implement an existing algorithm for tridiagonal reduction using CUDA, thus leveraging the parallelism present in GPUs to accelerate the process.

In a serial implementation of the algorithm, at each step only the elements in 2 rows/ columns are modified. Therefore, the CUDA implementation takes the form of a parallel reduction algorithm which will simultaneously apply multiple Jacobi rotations to the matrix, thus zeroing out multiple elements at the same time.

The CUDA implementation of this algorithm was measured to have a performance improvement of roughly an order of magnitude for sufficiently large matrices as compared to the reference serial CPU implementation. This result shows that the parallel algorithm is able to successfully exploit the GPU's parallel architecture and provide a significant improvement to the performance of the original algorithm.

Subject Keywords: Eigenvalue; Eigenvector; Givens Method; GPU; CUDA

1. Introduction 1
2. Review of Existing Tridiagonalization Methods 2
2.1. Givens Method 2
2.2. Householder Method 2
2.3. Lanczos Algorithm 2
3. High Level Algorithm Overview 4
3.1. Jacobi Transformations 4
3.2. Givens Matrix 5
3.3. Eliminating Elements 7
4. Implementation Details 10
4.1. Introduction 10
4.2. Kernel Design 11
4.3. Optimizations 14
5. Results 17
5.1. Experimental Setup 17
5.2. Comparison Against CPU Based Implementation 18
5.3. Effect of CUDA Block Size on Performance 20
5.4. Effect of Matrix Sparsity on Performance 22
6. Conclusion 23
References 24
Appendix A Program Implementation 25
A.1. CPU Wrapper 25
A.2. GPU Kernel Functions 27

1. Introduction

An efficient strategy for finding eigenvalues and eigenvectors is to reduce a matrix to a tridiagonal form before starting an iterative procedure. The reason a tridiagonal form is chosen instead of a purely diagonal form is because tridiagonalization requires only a finite number of steps, while the Jacobi method used to reduce a matrix to diagonal form requires iteration to convergence.

This work explores the feasibility of implementing the Givens method of tridiagonalization on the GPU. This allows the algorithm to leverage the massively parallel architecture of modern day GPUs to be able to simultaneously perform a large number of simple floating point operations.

2. Review of Existing Tridiagonalization Methods

2.1. Givens Method

The Givens method [1] uses Jacobi transformations to reduce the original matrix A to tridiagonal form. While the original Jacobi method reduces A to diagonal form by iterating to convergence, the Givens method instead requires only a finite number of steps by reducing A to tridiagonal form. Each iteration of the Givens method uses a Jacobi transformation to eliminate one of the non-tridiagonal elements. A total of $n^2/2$ transformations are required to reduce A to tridiagonal form.

2.2. Householder Method

The Householder method [1] reduces the original matrix A to a tridiagonal matrix in a similar manner as the Givens method. The Householder matrix P will eliminate all but the first element of a row or column when applied to A. Reducing the set of elements in A being considered each iteration will ensure that the element that is not eliminated in a row or column is always the tridiagonal element. The Householder method will take a total of $n - 2$ iterations to reduce A to tridiagonal form.

2.3. Lanczos Algorithm

The Lanczos algorithm [2] is an iterative algorithm for finding eigenvalues and eigenvectors using the power method. The original matrix A is multiplied to a randomized vector at each step of the iteration. At the end of n iterations, where n is the dimensions of the A, A is transformed into tridiagonal matrix T_{nn}, which is similar to A.

However, the Lanczos algorithm is not very numerically stable. Numerical instability results from inaccuracy in floating point arithmetic and results in a loss of orthogonality

for the vector. Implementations of the algorithm have to resolve issues with the loss of orthogonality to generate the correct eigenvalues and eigenvectors.

3. High Level Algorithm Overview

3.1. Jacobi Transformations

The Jacobi transformations form the basis of the algorithm. Each transformation, also known as a Jacobi rotation, is a plane rotation designed to eliminate one of the non-diagonal elements.

The Jacobi rotation is a matrix $\boldsymbol{P_{pq}}$ of the form shown in Figure 3.1.

$$P_{pq} = \begin{bmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & c & \cdots & s & & & \\ & & & \vdots & 1 & \vdots & & & \\ & & & -s & \cdots & c & & & \\ & & & & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{bmatrix}$$

Figure 3.1. Jacobi Rotation Matrix

The matrix has the following properties:

- All non-diagonal elements are zeroes except for two elements s and $-s$.
- All diagonal elements are ones except for two elements c in rows p and q.

Given a matrix **A**, the Jacobi rotation, $\boldsymbol{P_{pq}}$, is applied to the matrix in the following manner:

$$A' = P_{pq}^T \cdot A \cdot P_{pq} \tag{3.1}$$

The matrix multiplication $P_{pq}^T \cdot A$ will modify rows p and q of **A**, while $A \cdot P_{pq}$ will modify columns p and q of **A**.

3.2. Givens Matrix

Section 3.1 provided a conceptual overview of the Jacobi rotations and how they are applied to a matrix. We will now proceed to explain how Jacobi rotations can be used to eliminated elements in a matrix using specific values of *c* and *s*.

As seen in section 3.1, the matrix P^T_{pq} will modify the rows *p* and *q* of the input matrix **A**. By choosing the right values of *c* and *s*, we are able to modify the elements in rows *p* and *q* such that an element in row *q* is eliminated. Given a column position *j*, and two elements *a* = *p[j]* and *b* = *q[j]*, we will choose the following values of *c* and *s* to zero out element *b*.

$$c = \frac{a}{\sqrt{a^2 + b^2}} \tag{3.2}$$

$$s = \frac{b}{\sqrt{a^2 + b^2}} \tag{3.3}$$

We then create the Givens Matrix, **g**, in the following manner:

$$g[0] = c \tag{3.4}$$

$$g[1] = s \tag{3.5}$$

$$g[2] = -s \tag{3.6}$$

$$g[3] = c \tag{3.7}$$

Next, we apply the Givens Matrix to all the elements in the rows *p* and *q*. For each element in the rows *p* and *q*, we apply the Givens Matrix as follows:

$$p[i] = g[0] \bullet p[i] + g[1] \bullet q[i] \tag{3.8}$$

$$q[i] = g[2] \bullet p[i] + g[3] \bullet q[i] \tag{3.9}$$

For the original elements a and b, these are their new values:

$$\begin{aligned} a &= g[0] \bullet a + g[1] \bullet b \\ &= ca + sb \\ &= \frac{a}{\sqrt{a^2+b^2}} a + \frac{b}{\sqrt{a^2+b^2}} b \\ &= \frac{a^2+b^2}{\sqrt{a^2+b^2}} \end{aligned} \tag{3.10}$$

$$\begin{aligned} b &= g[2] \bullet a + g[3] \bullet b \\ &= -sa + cb \\ &= -\frac{b}{\sqrt{a^2+b^2}} a + \frac{a}{\sqrt{a^2+b^2}} b \\ &= \frac{-ab+ab}{\sqrt{a^2+b^2}} \\ &= 0 \end{aligned} \tag{3.11}$$

As such, we are able to use the Givens matrix to zero out element b while modifying the rest of rows p and q. From the above result, we are able to come up with the basis for our algorithm to transform the matrix into a tridiagonal form.

3.3. Eliminating Elements

3.3.1. Numerical Demonstration

Figure 3.2 presents a demonstration of the process used to form a tridiagonal matrix from a symmetrical matrix.

At each iteration of the algorithm, we will select the next column/row j to process from left to right for the columns and top to bottom for the row.

We will then iteratively form pairs of rows and pairs of columns, then compute the Givens matrix based on the jth elements of these rows/columns

The Givens matrix is then applied to each element in the two rows/columns following the algorithm in the previous section. This will zero out the jth element in the second row/column in each pair.

$$\begin{pmatrix} 1 & 0.99393 & 0.76016 & 0.12046 \\ 0.99393 & 1 & 0.09791 & 0.57483 \\ 0.76106 & 0.09791 & 1 & 0.37725 \\ 0.12046 & 0.57483 & 0.37725 & 1 \end{pmatrix} \rightarrow \quad \text{(Original Matrix)}$$

$$\begin{pmatrix} 1 & 1.2513 & 0 & 0.1205 \\ 1.2513 & 1.0945 & 0.0256 & 0.6858 \\ 0 & 0.0256 & 0.9055 & -0.0496 \\ 0.1205 & 0.6858 & -0.0496 & 1 \end{pmatrix} \rightarrow \quad \text{(Eliminated elements [2][0] and [0][2])}$$

$$\begin{pmatrix} 1 & 1.2571 & 0 & 0 \\ 1.2571 & 1.2244 & 0.0208 & 0.6642 \\ 0 & 0.0208 & 0.9055 & -0.0518 \\ 0 & 0.6642 & -0.0518 & 0.8700 \end{pmatrix} \rightarrow \quad \text{(Eliminated elements [3][0] and [0][3])}$$

$$\begin{pmatrix} 1 & 1.2571 & 0 & 0 \\ 1.2571 & 1.2244 & 0.6645 & 0 \\ 0 & 0.6645 & 0.8668 & 0.0506 \\ 0 & 0 & 0.0506 & 0.9087 \end{pmatrix} \quad \text{(Eliminated elements [3][1] and [1][3])}$$

Figure 3.2.Numerical Demonstration of Tridiagonalization Process

3.3.2. Numerical Demonstration of Concurrent Operations

In the parallel implementation of the algorithm as shown in Figure 3.3, we will utilize the properties of the Jacobi transformation to simultaneously zero out multiple elements in a single iteration.

As we have seen, eliminating an element from the matrix will only modify the values of two rows/columns. This means that we can perform multiple Jacobi Transformations at the same time as long as our pairs of rows and columns do not overlap.

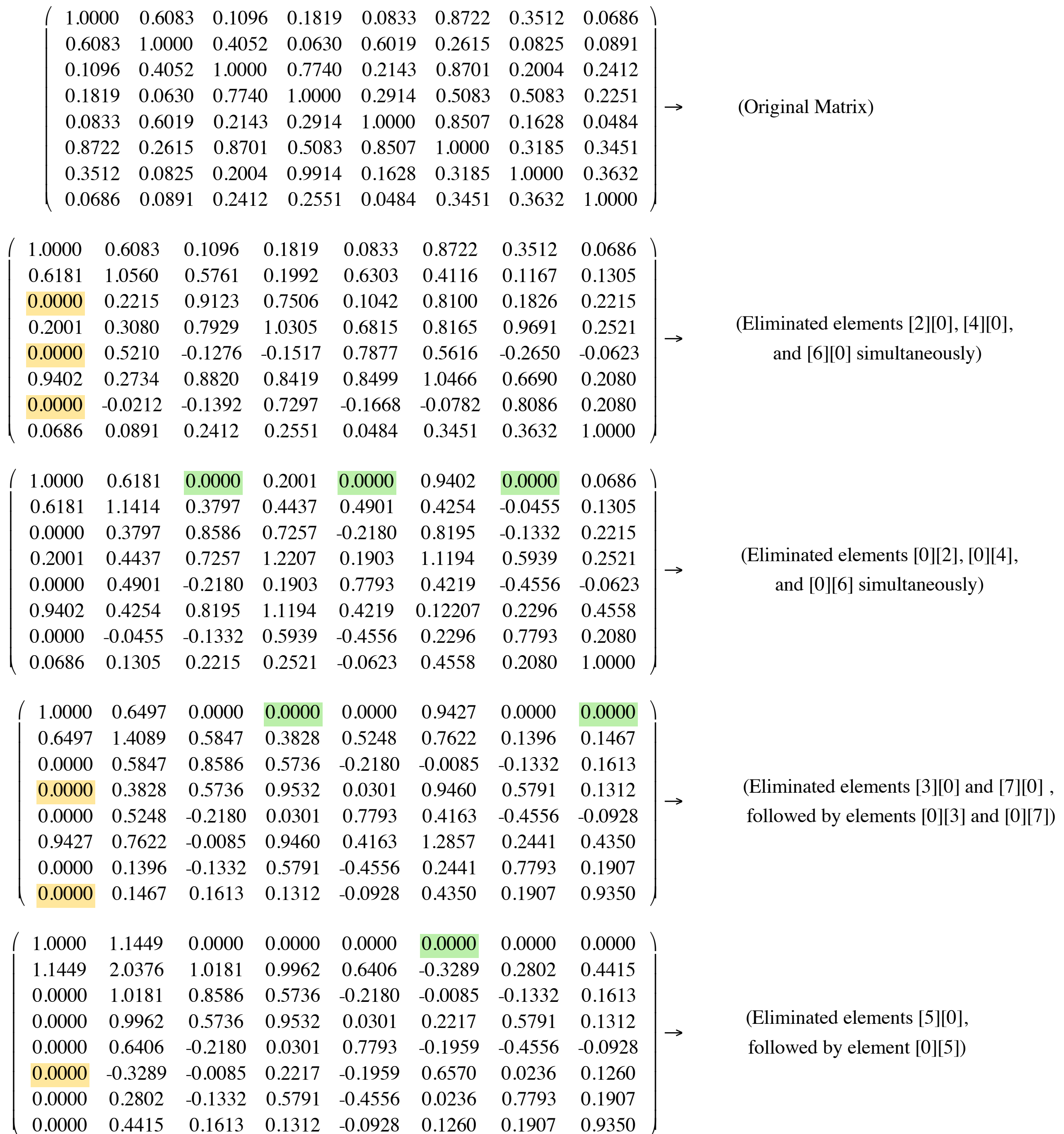
We use a parallel reduction algorithm with multiple iterations to eliminate elements for a set of columns and rows.

In the first iteration, we pair each row up with the row immediately below it, e.g., rows 1 and 2, rows 3 and 4,…. This means that for a column with n elements, we will have $(n-1)/2$ pairs of rows. The *n-1* comes from the fact that the top element in each column is a tridiagonal element and should not be eliminated. The first element of every other row, e.g., 2, 4, 6,… will be eliminated.

In the next iteration, we will then pair each row with the row that is 2 rows below it, e.g., rows 1 and 3, rows 5 and 7,… since rows 2, 4, 6,… were eliminated in the previous iteration. This means that we have half the number of pairs as compared to the previous iteration. This iteration will eliminate the bottom row of each pair, i.e., rows, 3, 7, 11,….

For each iteration, after eliminating the relevant elements in the column, we will also need to apply the same process to the corresponding row. We will consider the iteration to be complete once we have eliminated all non-tridiagonal elements from both one column and one row.

On the completion of one iteration, we will have achieved the desired tridiagonal form for one row and one column. Since all the non-tridiagonal elements are zeros, further rotations will not affect these elements. We are therefore able to consider a smaller subset of the matrix for the next iteration.



$$\begin{pmatrix} 1.0000 & 0.6083 & 0.1096 & 0.1819 & 0.0833 & 0.8722 & 0.3512 & 0.0686 \\ 0.6083 & 1.0000 & 0.4052 & 0.0630 & 0.6019 & 0.2615 & 0.0825 & 0.0891 \\ 0.1096 & 0.4052 & 1.0000 & 0.7740 & 0.2143 & 0.8701 & 0.2004 & 0.2412 \\ 0.1819 & 0.0630 & 0.7740 & 1.0000 & 0.2914 & 0.5083 & 0.5083 & 0.2251 \\ 0.0833 & 0.6019 & 0.2143 & 0.2914 & 1.0000 & 0.8507 & 0.1628 & 0.0484 \\ 0.8722 & 0.2615 & 0.8701 & 0.5083 & 0.8507 & 1.0000 & 0.3185 & 0.3451 \\ 0.3512 & 0.0825 & 0.2004 & 0.9914 & 0.1628 & 0.3185 & 1.0000 & 0.3632 \\ 0.0686 & 0.0891 & 0.2412 & 0.2551 & 0.0484 & 0.3451 & 0.3632 & 1.0000 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 1.0000 & 0.6083 & 0.1096 & 0.1819 & 0.0833 & 0.8722 & 0.3512 & 0.0686 \\ 0.6181 & 1.0560 & 0.5761 & 0.1992 & 0.6303 & 0.4116 & 0.1167 & 0.1305 \\ 0.0000 & 0.2215 & 0.9123 & 0.7506 & 0.1042 & 0.8100 & 0.1826 & 0.2215 \\ 0.2001 & 0.3080 & 0.7929 & 1.0305 & 0.6815 & 0.8165 & 0.9691 & 0.2521 \\ 0.0000 & 0.5210 & -0.1276 & -0.1517 & 0.7877 & 0.5616 & -0.2650 & -0.0623 \\ 0.9402 & 0.2734 & 0.8820 & 0.8419 & 0.8499 & 1.0466 & 0.6690 & 0.2080 \\ 0.0000 & -0.0212 & -0.1392 & 0.7297 & -0.1668 & -0.0782 & 0.8086 & 0.2080 \\ 0.0686 & 0.0891 & 0.2412 & 0.2551 & 0.0484 & 0.3451 & 0.3632 & 1.0000 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 1.0000 & 0.6181 & 0.0000 & 0.2001 & 0.0000 & 0.9402 & 0.0000 & 0.0686 \\ 0.6181 & 1.1414 & 0.3797 & 0.4437 & 0.4901 & 0.4254 & -0.0455 & 0.1305 \\ 0.0000 & 0.3797 & 0.8586 & 0.7257 & -0.2180 & 0.8195 & -0.1332 & 0.2215 \\ 0.2001 & 0.4437 & 0.7257 & 1.2207 & 0.1903 & 1.1194 & 0.5939 & 0.2521 \\ 0.0000 & 0.4901 & -0.2180 & 0.1903 & 0.7793 & 0.4219 & -0.4556 & -0.0623 \\ 0.9402 & 0.4254 & 0.8195 & 1.1194 & 0.4219 & 0.12207 & 0.2296 & 0.4558 \\ 0.0000 & -0.0455 & -0.1332 & 0.5939 & -0.4556 & 0.2296 & 0.7793 & 0.2080 \\ 0.0686 & 0.1305 & 0.2215 & 0.2521 & -0.0623 & 0.4558 & 0.2080 & 1.0000 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 1.0000 & 0.6497 & 0.0000 & 0.0000 & 0.0000 & 0.9427 & 0.0000 & 0.0000 \\ 0.6497 & 1.4089 & 0.5847 & 0.3828 & 0.5248 & 0.7622 & 0.1396 & 0.1467 \\ 0.0000 & 0.5847 & 0.8586 & 0.5736 & -0.2180 & -0.0085 & -0.1332 & 0.1613 \\ 0.0000 & 0.3828 & 0.5736 & 0.9532 & 0.0301 & 0.9460 & 0.5791 & 0.1312 \\ 0.0000 & 0.5248 & -0.2180 & 0.0301 & 0.7793 & 0.4163 & -0.4556 & -0.0928 \\ 0.9427 & 0.7622 & -0.0085 & 0.9460 & 0.4163 & 1.2857 & 0.2441 & 0.4350 \\ 0.0000 & 0.1396 & -0.1332 & 0.5791 & -0.4556 & 0.2441 & 0.7793 & 0.1907 \\ 0.0000 & 0.1467 & 0.1613 & 0.1312 & -0.0928 & 0.4350 & 0.1907 & 0.9350 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 1.0000 & 1.1449 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 1.1449 & 2.0376 & 1.0181 & 0.9962 & 0.6406 & -0.3289 & 0.2802 & 0.4415 \\ 0.0000 & 1.0181 & 0.8586 & 0.5736 & -0.2180 & -0.0085 & -0.1332 & 0.1613 \\ 0.0000 & 0.9962 & 0.5736 & 0.9532 & 0.0301 & 0.2217 & 0.5791 & 0.1312 \\ 0.0000 & 0.6406 & -0.2180 & 0.0301 & 0.7793 & -0.1959 & -0.4556 & -0.0928 \\ 0.0000 & -0.3289 & -0.0085 & 0.2217 & -0.1959 & 0.6570 & 0.0236 & 0.1260 \\ 0.0000 & 0.2802 & -0.1332 & 0.5791 & -0.4556 & 0.0236 & 0.7793 & 0.1907 \\ 0.0000 & 0.4415 & 0.1613 & 0.1312 & -0.0928 & 0.1260 & 0.1907 & 0.9350 \end{pmatrix} \rightarrow$$

Figure 3.3. Numerical Demonstration of Concurrent Tridiagonalization Process

4. Implementation Details

4.1. Introduction

The algorithm described in section 3.3.2 was implemented on the GPU. Based on the parallel reduction algorithm, we divide the matrix into pairs of rows or columns, then eliminated one element from each pair using the Givens matrix. The elimination process is done in parallel.

4.2. Kernel Design

Our GPU implementation follows the algorithm described in section 3.3.2. Due to the lack of a global synchronization mechanism in the CUDA model, it was necessary to divide up the algorithm into multiple kernels and iterate over them.

The overall algorithm, as seen in Figure 4.1, has a nested loop structure. The outer loop cycles through columns from left to right and rows from top to bottom. For each iteration of the outer loop, we will have successfully eliminated the non-tridiagonal elements for one row and one column. The inner loop implements the reduction process using the step variable. Each iteration of the inner loop increments the step variable to increase the distance for the pairs of rows/columns. Within the inner loop, the two GPU kernels `tridiagKernelRow` and `tridiagKernelCol` are called.

```
tridiagMain(matrix, size)
{
  for j from 0 to (size-2)
    step = 2
    maxStep = size - j

    while (step <= maxStep)
      tridiagKernelRow(matrix, size, j, step)
      tridiagKernelCol(matrix, size, j, step)

      step = step * 2
}
```

Figure 4.1. Algorithm Outline
A general outline of the algorithm implemented on the GPU as described in Chapter 3

The CUDA global functions represented by Figures 4.2 and 4.3 are invoked on the GPU. Each thread block is responsible for executing the algorithm on a pair of rows/columns. Each block will first calculate the rows or columns that it is mapped to, then determine whether any operation is required on this pair of rows/columns. An operation is required if the element to be zeroed out is currently non-zero. If an operation is required, the appropriate CUDA device function applyGivensTwoRows or applyGivensTwoCols will be invoked.

```
tridiagKernelRow(matrix, size, j, step)
  row1 = blockYPosition + j
  row2 = blockYPosition + j + step/2

  if (row2 < size && blockRequiredForCurrentStep == true)
    element1 = matrix[j][row1]
    element2 = matrix[j][row2]
    givensMatrix = calculateGivensMatrix(element1, element2)

    if (element2 != 0.0)
      applyGivensTwoRows(matrix, size, givensMatrix, row1, row2, j)
```

Figure 4.2. CUDA Global Kernel Function for Rows
Function computes the Givens matrix for pairs of rows and applies the Givens matrix to each pair if necessary

```
tridiagKernelCol(matrix, size, j, step)
  col1 = blockYPosition + j
  col2 = blockYPosition + j + step/2

  if (col2 < size && blockRequiredForCurrentStep == true)
    element1 = matrix[col1][j]
    element2 = matrix[col2][j]
    givensMatrix = calculateGivensMatrix(element1, element2)

    if (element2 != 0.0)
      applyGivensTwoCols(matrix, size, givensMatrix, col1, col2, j)
```

Figure 4.3. CUDA Global Kernel Function for Columns
Function computes the Givens matrix for pairs of columns and applies the Givens matrix to each pair if necessary

The CUDA device functions shown in Figure 4.4 and 4.5 will apply the Givens matrix to the rows/columns that are passed in as parameters. They will calculate the new values for each matrix element inside the selected rows/columns and write them to the matrix.

```
applyGivensTwoRows(matrix, givensMatrix, size, row1, row2, j)
  i = threadNumber

  if (i < size-j)
    index1 = row1 * n + i + j
    index2 = row2 * n + i + j

    a = givensMatrix[0] * matrix[index1] + givensMatrix[1] * matrix[index2]
    b = givensMatrix[2] * matrix[index1] + givensMatrix[3] * matrix[index2]

    matrix[index1] = a
    matrix[index2] = b
```

Figure 4.4. CUDA Device Function for Rows
Function applies the Givens matrix supplied as parameter to a pair of rows

```
applyGivensTwoCols(matrix, givensMatrix, size, col1, col2, j)
  i = threadNumber

  if (i < size-j)
    index1 = (i + j) * n + col1
    index2 = (i + j) * n + col2

    a = givensMatrix[0] * matrix[index1] + givensMatrix[1] * matrix[index2]
    b = givensMatrix[2] * matrix[index1] + givensMatrix[3] * matrix[index2]

    matrix[index1] = a
    matrix[index2] = b
```

Figure 4.5. CUDA Device Function for Columns
Function applies the Givens matrix supplied as parameter to a pair of columns

The CUDA threads and thread blocks are assigned in the following manner: Each thread block is assigned to a pair of rows/columns. A thread is responsible for applying the Givens matrix to the two elements with the same index in each pair of rows/columns. The number of threads in the block is set by the environmental variable BLOCK_SIZE. If there are more elements in a matrix row/column than threads in a thread block, then multiple thread blocks will be assigned to handle the same row/column.

4.3. Optimizations

4.3.1. Understanding Code Performance

Nvidia CUDA Profiler can be used to determine the distribution of time spent in each part of the program. It can also reveal the number of kernel launches during the program.

With the CUDA Profiler, we identified that for a 5000 x 5000 matrix, there are roughly 50000 kernel invocations. With each invocation taking ~30μs, we are looking at a total kernel launch time of ~1.5s. This figure is negligible in comparison to the total time taken for the algorithm to complete.

We have also identified regions in the program where most time is spent. For smaller matrices, this was the `cudaMalloc` and memory copying stage. For larger matrices however, most of the computation time was spent in the `triDiagKernelRow` and `triDiagKernelCol` kernels. This was in line with expectations because it meant that we were spending the most time in the sections of the program which required the most work.

4.3.2. Shared Memory

Shared memory is on-chip memory that provides much faster access speeds than global memory. Each thread block can be allocated its own shared memory. Shared memory is frequently used to store data that is frequently accessed by all the threads of the thread block.

In this case, we used shared memory as a form of cache for the Givens matrix. Based on the kernel design, shared memory is an ideal optimization to store the Givens matrix because each thread block shares an identical Givens matrix, and every thread in the

thread block will access the Givens matrix to determine the new value of the elements in the row/column. Using shared memory allows us to compute the Givens matrix only once per thread block and reduce the demand on global memory.

4.3.3. Block Size

The block size, or number of threads in a thread block, can have a significant impact on the performance of the program. Each multiprocessor on an Nvidia GPU has various limitations in terms of the maximum number of threads and thread blocks it can accept. Choosing the optimal block size will maximize the number of threads that are assigned to each multiprocessor. Further explanation on this optimization is provided in section 5.3.

4.3.4. Dynamic Kernel Scaling

The number of elements to be processed in each iteration decreases because they are eliminated. Rather than assign CUDA threads to elements that have already been eliminated, we will dynamically reduce the size of the kernel for each iteration so that we can reduce the number of threads without work, thereby freeing up GPU resources for the threads that have actual work to perform.

4.3.5. Reducing __syncthreads() calls

The __syncthreads() CUDA function provides a block level synchronization mechanism in a CUDA kernel. Synchronization in the kernel is expensive because faster threads will stall at a synchronization point to wait for slower threads to catch up. Therefore, reducing the number of __syncthreads() calls will have a positive impact on performance.

Originally, when applying the Givens matrix to a pair of rows, we assigned a single thread to each individual matrix element being altered as seen in Figure 4.6. However,

this required a __syncthreads() call at the end of the relevant device functions. We then made the modification such that each thread will handle two matrix elements as seen in Figure 4.7. This allowed us to remove the __syncthreads() call. The gain from removing this __syncthreads() call was able to outweigh the reduction in parallelism by having each thread performing twice the work.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 4.6. Original Device Function with __syncthreads() call

1	2	3	4
1	2	3	4
5	6	7	8
5	6	7	8

Figure 4.7. Optimized Device Function with no __syncthreads() call and each thread doing twice the work

5. Results

5.1. Experimental Setup

5.1.1. Hardware

	Model	Count
CPU	Intel Xeon X5680 @ 3.33GHz	2 x 6 cores
Motherboard	Tyan FT77 (B7015)	N/A
Hard Disk	300GB	N/A
Memory	12GB	N/A
GPU	Nvidia Tesla C2050 Nvidia Tesla M2090	7 1

Table 5.1. Hardware Setup

Table 5.1 shows our hardware setup for testing our implemented algorithm. Although we have multiple CPU cores, it is important to note that the reference CPU implementation is single threaded and therefore only runs on a single core. Similarly even though we have multiple GPUs installed in the system, our GPU implementation is restricted to executing on only one of these GPUs.

Software

OS	Red Hat Enterprise Linux Server 6.5 Kernel Version 2.6.32
CUDA Version	6.5

Table 5.2. Software Setup

Table 5.2 shows the software setup. We are using the latest version of CUDA as of Dec 2014 so that we have all the newest software features available in the GPU implementation

5.2. Comparison Against CPU Based Implementation

Figure 5.1 shows a comparison between the GPU implementation and the single-threaded CPU implementation, as well as the theoretical maximum for a multi-threaded CPU implementation based on our single-threaded CPU implementation.

The CPU implementation is faster than the GPU implementation for matrices up to 256 x 256 in size. This can be attributed to the GPU's memory copy as well as kernel invocation overheads.

However, the GPU implementation becomes faster than the CPU implementation for matrices sized 512 x 512 and above. In general we can expect a performance improvement of roughly an order of magnitude (~10x). For the following discussion, we will only be interested in the portion of the graph where the GPU implementation is faster than the CPU implementation.

The speedup factor for the GPU implementation as compared to the CPU implementation initially increases, but then starts to hold constant for most matrix sizes beyond 512 x 512. This is in line with our expectations. Since a GPU does not have an infinite amount of resources available, increasing the amount of parallelism with bigger matrices will eventually saturate the GPU's resources. At this point, the increase in computation time with larger matrices will scale by the same factor as the CPU implementation.

Another factor affecting the speedup is the formulation of the algorithm. Multiple dependencies are encountered when progressing through the steps of the algorithm. For example, we cannot calculate the new values of the matrix elements when applying the Givens matrix to columns unless we have already completed the step of applying the Givens matrix to rows. Since CUDA lacks a global synchronization mechanism, the only way to resolve these synchronization issues is to separate the steps into multiple GPU

kernels and launch them sequentially. This increases in overall computation time as the number of kernel launches will scale exponentially with a linear increase in matrix size.

This explanation also provides reason to believe that it is very difficult for a multi-threaded CPU implementation to reach the theoretical maximum we have plotted above. By assuming that a two-threaded algorithm can perform exactly twice as fast, we will be neglecting to factor in many of the dependency problems that we have encountered in our GPU implementation.

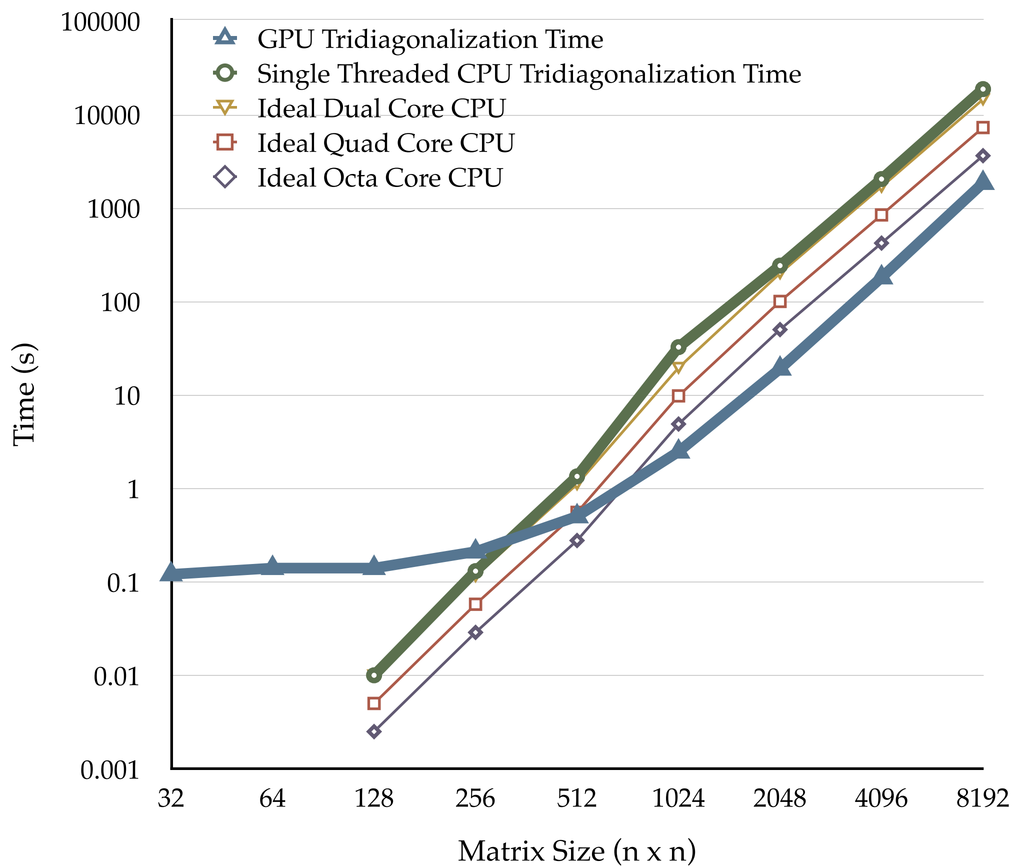


Figure 5.1. Comparison Against CPU Implementation

5.3. Effect of CUDA Block Size on Performance

To understand how the number of threads per block impacts performance, we have to look at the occupancy of each multiprocessor. The multiprocessor occupancy is a function based on the number of active warps per multiprocessor. For a device with compute capability 2.0, Table 5.4 shows the limitations on threads, warps and blocks on a multiprocessor [3].

Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Maximum Thread Block Size	1024

Table 5.3. NVIDIA GPU Multiprocessor Limitations for Compute Capability 2.0

To ensure the optimal performance, we need to maximize the multiprocessor occupancy given the above limitations. This generally means choosing a block size such that we achieve that maximum number of threads per multiprocessor.

In our test results presented in Figure 5.2 and 5.3, we achieve the best performance with between 192 to 256 threads per block.

Starting at 192 threads per block with a total of 8 blocks, we can achieve the maximum of 1536 threads per multiprocessor. Because of the limit of 8 blocks per multiprocessor, using any less than 192 threads per block will not allow the kernel to achieve the maximum multiprocessor occupancy.

The other thread block sizes we have chosen besides 1024 will also be able to achieve the maximum multiprocessor occupancy. However, the performance difference can also be attributed to other factors. Choosing a smaller number of threads per block will re-

sult in an overall higher number of blocks. This may create more options for the CUDA scheduler to allocate blocks to the different multiprocessors on the GPU.

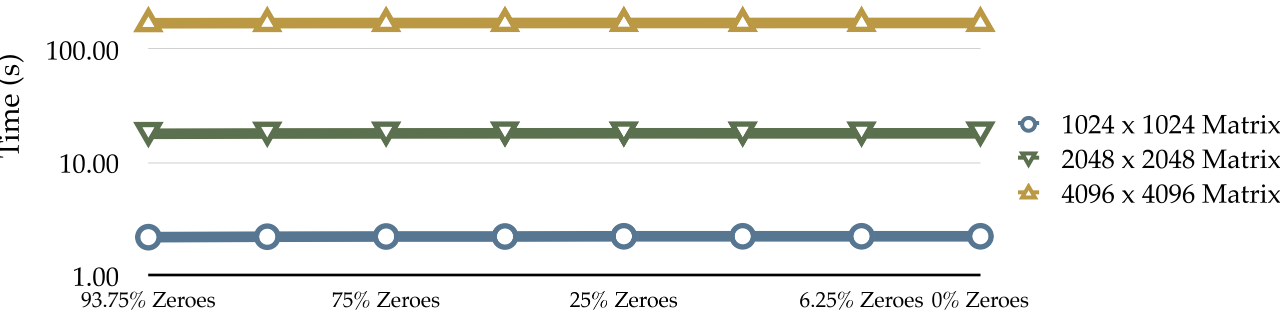


Figure 5.4. Effect of Matrix Sparsity

5.4. Effect of Matrix Sparsity on Performance

From our test results in Figure 5.7, we can see that matrix sparsity has little impact on our overall performance. Matrix sparsity could possibly impact performance due to branch divergence, where different program execution paths are taken based on whether or not an element is zero.

Branch divergence occurs when different threads in a warp take a different execution path. However, the kernel was designed in such a way that in most cases, threads in a block will perform the same function. Each block is responsible for applying the Givens matrix to a pair of rows/columns. Based on whether the element being zeroed out is already zero, the block will either apply the Givens matrix or terminate. This means that threads in a block will perform as a cohesive unit that minimizes branch divergence.

While it may be beneficial that our program performs consistently throughout a wide range of matrix sparsities, there are also drawbacks. When comparing the total amount of work done, clearly a more dense matrix requires more work than a spare matrix. This means that we may actually be doing unnecessary work for sparse matrices. It is possible that further optimizations could be applied to the program to improve its performance when operating on sparse matrices.

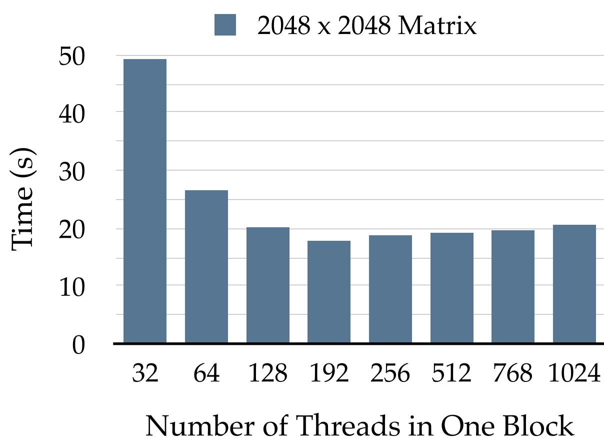


Figure 5.2. Effect of CUDA Block Size,
2048 x 2048 Matrix

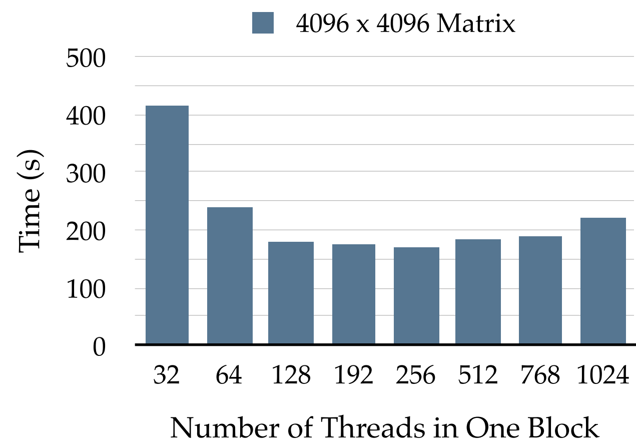


Figure 5.3. Effect of CUDA Block Size,
4096 x 4096 Matrix

6. Conclusion

Through this work, we have successfully implemented a parallel formulation of the Givens method for matrix tridiagonalization using CUDA. We have also verified its correctness by comparing the tridiagonalization results against existing CPU based implementations.

This parallel formulation resulted in a significant speedup of roughly 10x for non-trivial matrices as compared to a serial CPU implementation. CUDA optimizations also contributed to the speedup factor.

As with any project that aims to parallelize an existing algorithm, it was important to make sure that the parallel formulation could actually fully exploit the GPU's capabilities. This was done through reducing the number of dependencies between matrix elements and designing the CUDA kernel such we always do the maximum amount of work in parallel at each stage.

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge, UK: Press Syndicate of the University of Cambridge, 1992.

[2] C. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators," *Journal of Research of the National Bureau of Standards*, vol. 45, no. 4, October 1950, pp. 255-282.

[3] CUDA C Programming Guide, web page. Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed October 2014.

Appendix A Program Implementation

A.1. CPU Wrapper

```
void tridiagGPU(double *m, int n, double *d, double *e)
{
  cudaError_t cuda_ret;
  cuda_ret = cudaSetDevice(5);

  //Allocate device memory and copy matrix over to device
  double *m_d, *d_d, *e_d, *x_d;
  cudaMalloc((void**)&m_d, n*n*sizeof(double));
  cudaMalloc((void**)&d_d, n*sizeof(double));
  cudaMalloc((void**)&e_d, n*sizeof(double));
  cudaMalloc((void**)&x_d, n*n*sizeof(double));

  cudaMemcpy(m_d, m, n*n*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemset(d_d, 0, n*sizeof(double));
  cudaMemset(e_d, 0, n*sizeof(double));
  cudaMemset(x_d, 0, n*n*sizeof(double));


  // Set Grid and Block Dimensions
  dim3 dimBlock;
  dimBlock.x = BLOCK_SIZE;
  dimBlock.y = 1;
  dimBlock.z = 1;

  dim3 rowDimGrid;
  rowDimGrid.x = (int)(n/BLOCK_SIZE) + 1;
  rowDimGrid.y = (int)(n/2) + 1;
  rowDimGrid.z = 1;

  dim3 colDimGrid;
  colDimGrid.x = (int)(n/BLOCK_SIZE) + 1;
  colDimGrid.y = (int)(n/2) + 1;
  colDimGrid.z = 2;

  dim3 copyDimBlock;
  copyDimBlock.x = BLOCK_SIZE;
  copyDimBlock.y = 1;
  copyDimBlock.z = 1;

  dim3 copyDimGrid;
  copyDimGrid.x = (int)(n/BLOCK_SIZE) + 1;
  copyDimGrid.y = 1;
  copyDimGrid.z = 1;

  double *saveRow_d;
  double *saveCol_d;

  cudaMalloc((void**)&saveRow_d, n*sizeof(double));
  cudaMalloc((void**)&saveCol_d, n*sizeof(double));

  double *g_stored;
  cudaMalloc((void**)&g_stored, n*4*sizeof(double));

  setDiagToOne<<<copyDiagDimGrid, copyDiagDimBlock>>>(x_d, n);


  for (int j = 0; j < n - 2; j++) {
    int step = 2;
    int maxStep = n - j;
    maxStep = CPUget_nextpowerof2(maxStep);
```

```
    while (step <= maxStep) {
      copyCol<<<copyDimGrid, copyDimBlock>>>(m_d, saveCol_d, n, j);

      triDiagKernelRow<<<dimGrid, dimBlock>>>(m_d, n, d_d, e_d, x_d, j, step, saveCol_d, g_s-
tored);

      copyRow<<<copyDimGrid, copyDimBlock>>>(m_d, saveRow_d, n, j);

      triDiagKernelCol<<<colDimGrid, dimBlock>>>(m_d, n, d_d, e_d, x_d, j, step, saveRow_d, g_s-
tored);


      rowDimGrid.x = (int)((n-(j+1))/BLOCK_SIZE) + 1;
      rowDimGrid.y = (int)((n-(j+1))/2) + 1;

      step *= 2;
    }
  }

  copyDiag<<<copyDimGrid, copyDimBlock>>>(m_d, d_d, e_d, n);


  // Copy device memory back to host memory
  cuda_ret = cudaMemcpy(m, x_d, n*n*sizeof(double), cudaMemcpyDeviceToHost);
  cuda_ret = cudaMemcpy(d, d_d, n*sizeof(double), cudaMemcpyDeviceToHost);
  cuda_ret = cudaMemcpy(e, e_d, n*sizeof(double), cudaMemcpyDeviceToHost);

  cudaFree(m_d);
  cudaFree(d_d);
  cudaFree(e_d);
  cudaFree(x_d);
}
```

A.2. GPU Kernel Functions

```
__device__
void givens_GPU(double *g, double *g_shared, double a, double b, int i) {
  if (threadIdx.x == 0 && threadIdx.y == 0) {
    double r = sqrt(a*a + b*b);
    double rr = 1.0 / r;
    double c = a * rr;
    double s = b * rr;

    g[i * 4 + 0] = c;
    g[i * 4 + 1] = s;
    g[i * 4 + 2] = -s;
    g[i * 4 + 3] = c;

    g_shared[0] = c;
    g_shared[1] = s;
    g_shared[2] = -s;
    g_shared[3] = c;
  }
}
```

```
__device__
void apply_givens_2_rows_GPU(double *m, int n, double *g, int r1, int r2, int j) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < (n-j)) {
    int element1 = r1 * n + i + j;
    int element2 = r2 * n + i + j;

    double a = g[0] * m[element1] + g[1] * m[element2];
    double b = g[2] * m[element1] + g[3] * m[element2];

    m[element1] = a;
    m[element2] = b;

  }
}
```

```
__device__
void apply_givens_2_cols_GPU(double *m, int n, double *g, int c1, int c2, int j) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < (n-j)) {
    int element1 = (i + j) * n + c1;
    int element2 = (i + j) * n + c2;

    double a = g[0] * m[element1] + g[1] * m[element2];
    double b = g[2] * m[element1] + g[3] * m[element2];

    m[element1] = a;
    m[element2] = b;

  }
}
```

```
__global__
void triDiagKernelRow(double *m, int n, double *d, double *e, double *x, int j, int step, double
*originalCol, double *g_stored) {
  __shared__ double g[4];

  if ((blockIdx.y * 2 + 1 + j + step/2) < n && (blockIdx.y)%(step/2) == 0) {
    int row1 = blockIdx.y * 2 + 1 + j;
    int row2 = blockIdx.y * 2 + 1 + j + step/2;

    if (originalCol[row2] != 0.0) {
      givens_GPU(g_stored, g, originalCol[row1], originalCol[row2], row1);
      __syncthreads();
      apply_givens_2_rows_GPU(m, n, g, row1, row2, j);
    }
  }
}
```

```
__global__
void triDiagKernelCol(double *m, int n, double *d, double *e, double *x, int j, int step, double
*originalRow, double *g_stored) {
  __shared__ double g[4];

  if ((blockIdx.y * 2 + 1 + j + step/2) < n && (blockIdx.y)%(step/2) == 0) {
    int col1 = blockIdx.y * 2 + 1 + j;
    int col2 = blockIdx.y * 2 + 1 + j + step/2;

    if (originalRow[col2] != 0.0) {
      if (threadIdx.x == 0 && threadIdx.y == 0) {
        g[0] = g_stored[col1 * 4 + 0];
        g[1] = g_stored[col1 * 4 + 1];
        g[2] = g_stored[col1 * 4 + 2];
        g[3] = g_stored[col1 * 4 + 3];
      }
      __syncthreads();

      if (blockIdx.z == 0) {
        apply_givens_2_cols_GPU(m, n, g, col1, col2, j);
      }
      if (blockIdx.z == 1) {
        apply_givens_2_cols_GPU(x, n, g, col1, col2, 0);
      }
    }
  }
}
```

```
__global__
void copyRow(double *src, double *dest, int n, int row) {
  if ((blockIdx.x * blockDim.x + threadIdx.x) < n) {
    dest[blockIdx.x * blockDim.x + threadIdx.x] = src[row * n + (blockIdx.x * blockDim.x + threa-
dIdx.x)];
  }
}

__global__
void copyCol(double *src, double *dest, int n, int col) {
  if ((blockIdx.x * blockDim.x + threadIdx.x) < n) {
    dest[blockIdx.x * blockDim.x + threadIdx.x] = src[(blockIdx.x * blockDim.x + threadIdx.x) * n
+ col];
  }
}
```

```
__global__
void copyDiag(double *m, double *d, double *e, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i == 0) {
    d[0] = m[0];
  }

  if (i < n - 1) {
    d[i+1] = m[(i+1) * n + (i+1)];
    e[i+1] = m[(i+1) * n + i];
  }
}
```

```
__global__
void setDiagToOne(double *x, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    x[i * n + i] = 1;
  }
}
```