2016 IEEE International Parallel and Distributed Processing Symposium Workshops

Hessenberg Reduction with Transient Error Resilience on GPU-Based Hybrid Architectures

Yulu Jia,* Piotr Luszczek* Jack Dongarra*†‡
*University of Tennessee, USA †Oak Ridge National Laboratory, USA ‡University of Manchester, UK

***Abstract*—Graphics Processing Units (GPUs) have been seeing widespread adoption in the field of scientific computing, owing to the performance gains provided on computation-intensive applications. In this paper, we present the design and implementation of a Hessenberg reduction algorithm immune to simultaneous soft-errors, capable of taking advantage of hybrid GPU-CPU platforms. These soft-errors are detected and corrected on the fly, preventing the propagation of the error to the rest of the data. Our design is at the intersection between several fault tolerant techniques and employs the algorithm-based fault tolerance technique, diskless checkpointing, and reverse computation to achieve its goal. By utilizing the idle time of the CPUs, and by overlapping both host-side and GPU-side workloads, we minimize the resilience overhead. Experimental results have validated our design decisions as our algorithm introduced less than 2% performance overhead compared to the optimized, but fault-prone, hybrid Hessenberg reduction.**

I. Introduction

A transient error is an error in a signal or data element which is temporary, and caused by factors other than permanent component failures. Many phenomena have been blamed for transient errors, ranging from alpha particles from package decay, to cosmic rays and thermal neutrons. Cosmic rays were shown to be the most prevalent source of transient errors among these sources [1]. While transient errors may happen at different levels in the hardware hierarchy, such as communication links or digital logic, the most common situation is in the semiconductor storage.

Both GPUs and traditional CPUs, and their associated memory, are prone to transient errors. CPU designs increasingly scale the number of cores and the memory hierarchies in order to provide more processing ability. Along with increasing transistor density, newer CPU designs also adopt faster clock frequency and lower voltage. More transistors per unit area means the size of each transistor gets smaller. A smaller feature size, combined with lower voltage to maintain transistor states, makes the transistor state easier to change, and therefore more vulnerable to external factors that might change the state. The critical charge Q_{crit}, which is the lowest electron charge needed to change the logical level, decreases as the chip feature size decreases. Higher transistor density also causes higher heat density which brings more thermal neutrons which contribute to transient errors as well. General Purpose Graphics Processing Units (GPGPUs) are gaining popularity in the scientific computing community due to the sizable acceleration they provide to computation intensive applications. A significant percentage of the acceleration is due to the large amount of data processing transistors inside the GPGPUs, where the number of transistors follow a even more drastic increase than in the CPU. As the evolution of the conventional processors and accelerators follows similar trends, the presence and frequency of transient errors have comparable progression and identical effect, becoming a disturbance to application developers.

Transient errors are also becoming a challenge for the applications. Both CPU main memory and GPU memory are DRAMs (Dynamic Random-access Memory). Baumann [2] reported that the soft error rate (SER)[1] of DRAM is between 1k FIT/chip to 10K FIT/chip range, and stays at the same level over 7 generations of DRAMs. Similarly, Jacob et al. [3] reported that at the 130 nm process SRAM memory exhibits a 100k FIT/chip. Michalak et al. [4] reported that the ASC Q supercomputer at Los Alamos National Laboratory experienced an average of 51.7 soft errors per week over a period of 7 weeks from September 2004 to October 2004. More recently, Haque et al. [5] assessed the probability of soft errors in NVIDIA GPUs using a benchmark called MemtestG80. They ran the test on 50000 GPUs and found that about 60% of the GPUs have a soft error probability (per test iteration) higher than 10^{-5} and a large population with a mean of 2×10^{-5}. ECC memory can protect data from being corrupted, but ECC incurs high storage overhead. It is beneficial to explore alternative methods to protect application data which as low storage overhead.

It goes without saying that science is based on facts and on experiments that can be replicated and results that can be trusted and verified. A single soft error can have a major impact on the outcome of any computation as it can drastically alter the results, and thus the understanding of the analyzed phenomenon. In the extremely volatile execution environments we will encounter in the very near future, it is critical that the pillar of scientific applications, the notion of trust in the scientific outcome, is not undermined. This requires the data and the result to be carefully validated to ensure it matches the experiment, and it has not been altered during the computational phase. Ensuring this property is a difficult task if we are bound to generic methodologies. Fortunately, some of the most widely used algorithms have inherently properties that can be advantageously exploited in fulfilling this need.

In this paper, we design and implement a soft error resilient Hessenberg reduction algorithm for GPU enabled hybrid architectures. We take advantage of diskless checkpointing,

[1]The measurement unit of (SER) is Failure in time (FIT), and one FIT is one soft error in 10^9 device hours.

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.34

ABFT, and reverse computation techniques to achieve soft error resilience while introducing very little overhead compared to the non fault tolerant Hessenberg reduction. We further minimize the overhead by carefully overlapping workloads on the host side and the GPU side. Unlike the post-processing scheme for LU and QR in [6], [7], [8], our algorithm detects soft errors at the end of each iteration. Once detected, the errors are corrected right away, preventing the errors from propagating and contaminating other matrix elements. While the above mentioned post-processing scheme can only correct up to two soft errors total during the course of the entire LU or QR factorization, our fault tolerant Hessenberg algorithm can detect and correct more than one simultaneous soft error, assuming that the error positions in the matrix do not form a rectangle. Once the algorithm has corrected the simultaneous errors, it continues as normal and is ready to detect and correct subsequent soft errors as they occur.

The remainder of the paper is organized as follows: in Section II we survey related work, then in Section III we explain the Hessenberg reduction algorithm and its implementation in the MAGMA framework. Section IV describes our soft error resilient hybrid Hessenberg reduction algorithm in detail. Section V gives a formal analysis on the performance overhead of the fault tolerant algorithm. Section VI presents the experiment results of the algorithm and provides a theoretical analysis for the performance. Section VII summarizes our work.

II. Related work

Plank et al. [9] presented a fault tolerant technique based on checksum and reverse computation for matrix computations on networks of workstations (NOWs). Their scheme tackles node failures instead of soft errors. A checksum of each processor's local matrix data is stored in main memory and regenerated periodically. When a node failure happens, the live processors reverse the computations that occurred after the failure so that the matrix data and the checksum are consistent with each other. Then the lost data on the failed processor are recovered using the checksum and the data on the live processors. Chen and Abraham [10] devised methods to detect and locate faulty processors in the computation of eigenvalues and singular values on systolic arrays. Their methods take the special properties of eigenvalue computation and singular value computation into consideration to make the detection of errors very efficient.

While the field of fault tolerance was dominated for years by solutions to address hard errors, with the increase in the number of computing components, the impact of soft errors has attracted significant attention, especially in linear algebra. Based on the ABFT idea [11], [12], [13], Du et al. [6], [7] proposed an algorithm to tolerate soft errors in the High Performance LINPACK Benchmark (HPL) [14]. Their approach can compute the correct solution vector to $Ax = b$ in the presence of one or two soft errors over the course of the factorization. Du et al. [8] also designed a scheme to tolerate soft errors in the QR factorization on hybrid systems with GPGPUs. At most, one soft error can be tolerated in this fault tolerant hybrid QR algorithm. Both the HPL fault tolerant scheme and QR fault tolerant scheme adopt a post processing approach in which the erroneous result is corrected through post processing after the regular factorization. Bronevetsky and Supinski [15] studied the impact of soft errors on iterative linear algebra methods. They found that iterative methods are vulnerable to soft errors as well and exhibit poor soft error detection abilities. Shantharam et al. [16] analyzed the propagation pattern of soft errors in iterative methods by modeling the iterative process with a sequence of sparse matrix-vector multiplication (SpMV) operations. Shantharam et al. [17] proposed a soft error tolerant preconditioned conjugate gradient algorithm for sparse linear systems. Their method adapted the algorithm based fault tolerance technique to sparse linear systems and achieved an overhead of 11.3% when no soft error occurs. Chen and Abraham [10] designed a concurrent error detection scheme for transient errors in the computation of eigenvalues on systolic processor arrays using the QR algorithm [18], [19] (not to be confused with the QR factorization). Cao et al. [20] designed a soft error resilient task-based runtime with three options to achieve fault tolerance.

Plank et al. [21] first introduced the idea of diskless checkpointing which eliminates the disk access bottleneck in the traditional checkpointing technique. In the traditional checkpointing technique, checkpoints are stored to secondary stable memory, usually in the form of hard drives. Since disk accesses are very slow compared to floating point computation, frequently writing checkpoints to disk incurs a big overhead. With diskless checkpoint, the checkpoints are stored in main memory instead of hard disk. Main memory access is much faster than hard drive access, so diskless checkpointing can greatly reduce the memory access overhead.

The Matrix Algebra on GPU and Multicore Architectures project (MAGMA) [22] is a dense linear algebra library for hybrid architectures with GPUs. The library provides equivalent functionalities to LAPACK [23] and uses block algorithms similar to those of LAPACK. By scheduling workloads with different characteristics to CPUs and GPUs, the hybrid algorithms are able to take advantage of both computational units and gain considerable acceleration over their LAPACK counterparts. The hybrid Hessenberg reduction algorithm in MAGMA also utilizes both CPUs and GPUs in a hybrid system. This hybrid algorithm is adapted from the LAPACK algorithm in order to separate workloads which are more suitable for GPUs from workloads that are suitable for CPUs. Details of this hybrid algorithm will be explained in the next section.

III. Hessenberg reduction accelerated with GPU

In this section, we describe the Hessenberg reduction algorithm and its variation as implemented in MAGMA.

A. The Unblocked Hessenberg Reduction

A square matrix H in which all entries below the first subdiagonal are zeros is said to be in upper Hessenberg matrix form. Reduction of a square matrix A to the Hessenberg form H is an important intermediate step in the Hessenberg QR algorithm which is used to compute the eigenvalues of A. Given a square matrix A, we apply a sequence of orthogonal similarity transformations Q_i to A:

$$H = Q_n^{-1} Q_{n-1}^{-1} \cdots Q_2^{-1} Q_1^{-1} A Q_1 Q_2 \cdots Q_{n-1} Q_n$$

let $Q = Q_1 Q_2 \cdots Q_{n-1} Q_n$, we have:

$$H = Q^{-1} A Q = Q^T A Q.$$

Q_i is chosen to be the Householder reflector, which eliminates the elements below the first subdiagonal in the i-th column of $Q_{i-1}^{-1} \cdots Q_1^{-1} A Q_1 \cdots Q_{i-1}$.

B. The Blocked Hessenberg Reduction

The speed of the unblocked Hessenberg reduction algorithm on modern computers is constrained by the latency of memory accesses. The blocked Hessenberg reduction algorithm [24] greatly increased the arithmetic intensity by grouping nb Householder reflectors and applying the group to A at the same time.

$$U_1 = Q_1 Q_2 \cdots Q_{nb} = I - VTV^T$$

where I is the identity matrix, V is an $N \times nb$ matrix composed of the Householder vectors, and T is an $nb \times nb$ upper triangular matrix. This representation of U_1 is called the *compact WY representation* [25]. This representation requires less storage to store U_1 and enables the use of matrix-matrix multiplications in the factorization. Matrix-matrix multiplications are desirable because of their high arithmetic intensity and efficient implementation on modern computers with hierarchical memory systems. Algorithm 1 shows the blocked Hessenberg reduction algorithm as implemented in the LAPACK **DGEHRD** routine. $trail(A)$ means the trailing submatrix in that iteration.

Algorithm 1 Blocked Hessenberg Reduction

1: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
2: **DLAHRD**, return V, T and Y where $Y = AVT$
3: **DGEMM**: $trail(A) = trail(A) - YV^\top$
4: **DLARFB**: $trail(A) = trail(A) - VT^\top V^\top trail(A)$
5: **end for**

C. Hessenberg Reduction in MAGMA

The hybrid Hessenberg reduction algorithm in MAGMA is an adapted version of Algorithm 1. Algorithm 2 shows the pseudocode for the hybrid Hessenberg reduction algorithm [26]. The input matrix A is stored in LAPACK layout, and matrix elements are stored contiguously in column major format. The matrix is logically divided into block columns, each block column is of size $N \times nb$. Upon completion, the matrix entries below the first subdiagonal are overwritten with the final Q matrix and the upper part of the matrix is overwritten with the final H matrix. The hybrid algorithm executes all the updates to the trailing matrix on the GPU. The panel factorization is assigned to the CPU, and the next panel to be factorized is transfered back to the host when both the right update and left update from the previous panel have been applied to it. Line 6 is an asynchronous data transfer, and control is returned to the CPU immediately after the data transfer is issued so that the CPU can initiate the next computation kernel. GPUs are able to do computation in parallel with communication, and using asynchronous data transfer hides the time cost to transfer the upper part of the current panel back to the CPU when it is updated and will not be modified again. The two lines in Algorithm 2, shown in red, are overlapped with each other.

Figure 1 visually illustrates one iteration of Algorithm 2; the computation routine called in each step and the data it operates on are pointed out with a black box. Figure 1(a) shows the state at the beginning of this iteration. The matrix elements in the yellow triangle and in the green trapezoid are the final results of the Q matrix and the H matrix, and they reside on the host side and will not be modified again. The red rectangle is the trailing matrix which will be factorized and updated in this iteration. The first nb columns of the red part are called a *panel* which will be factorized next. Figure 1(b) shows the panel factorization **DLAHRD** which factorizes the lower part of the current panel. The yellow upper triangular matrix is updated and contains the final results of H. The green trapezoid contains the Householder vectors which are the final results in the Q matrix. Upon completion of **DLAHRD**, both the yellow triangle and the green trapezoid are on the host side. Figure 1(c) shows the right update on M. M is the part of the matrix marked by the black box which consists of the upper part of the current panel and the upper part of the trailing matrix. This step corresponds to line 5 of Algorithm 2. Upon completion of this step, the $nb \times nb$ square matrix in yellow contains the final results of H, and it will not be modified again. This square matrix is sent back to the host side with an asynchronous data transfer. Figure 1(d) shows the right update to G. The G matrix is the lower part of the trailing matrix marked by the black box. In Figure 1(e) the left update to G is applied through the **DLARFB** call. After the **DLARFB** call, the matrix A has a smaller trailing matrix to be factorized in the next iteration. Figure 1(f) shows the state of the matrix at the end of this iteration. The rectangular matrix in red is the trailing matrix.

IV. Soft error resilient Hessenberg reduction algorithm

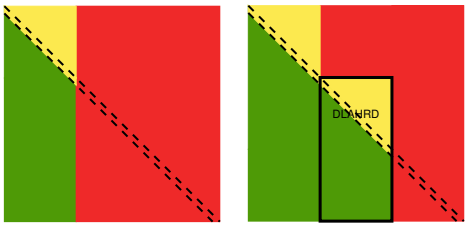
A. Failure Model

In this work, we consider soft errors, which are temporary faults in the data matrix, where the factorization is oblivious to the error and continues as usual. Without loss of generality, we assume only one error happens at a single point in time.

In the MAGMA Hessenberg reduction algorithm, both the CPU and GPU carry out computation. The CPU is responsible for the panel factorization, and the GPU is responsible for

Algorithm 2 Hybrid Hessenberg Reduction

1: Transfer matrix: A on the host $\rightarrow$ d_A on the GPU
2: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
3: Send the lower part of the next panel P to the host.
4: **MAGMA_DLAHR2**, return V, T *and* Y where $Y = [P, G]VT$
5: **DGEMM**: $M = M - MVTV^\top$
6: Send the leftmost nb columns of M to the host asynchronously.
7: **DGEMM**: $G = G - YV^\top$
8: **DLARFB**: $trail(A) = trail(A) - VT^\top V^\top trail(A)$
9: **end for**



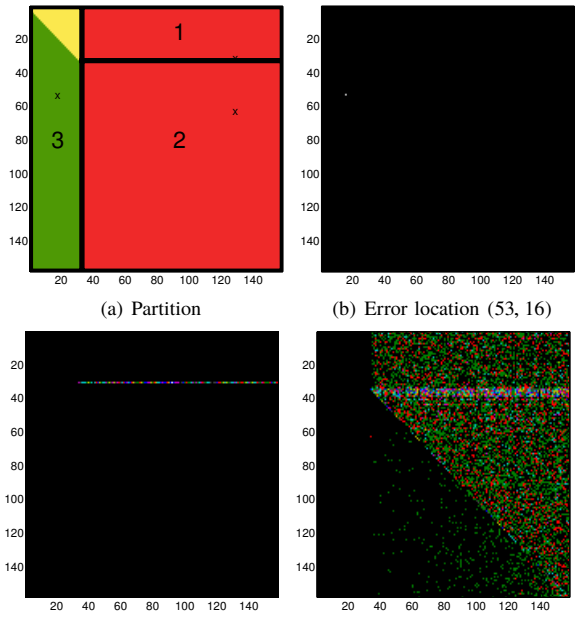
(a) Beginning of iteration (b) Factorize the panel P

(c) Right update to M (d) Right update to G

(e) Left update to G (f) End of iteration

Fig. 1. One iteration of DGEHRD

the trailing matrix update. Both the CPU memory and GPU memory contain part of the final result or intermediate data that are used to compute the final result. The lower triangular matrix to the left of the current panel on the host side contains part of the final result of the Q matrix. The upper triangular matrix to the left of the current panel on the host side contains the final result of the H matrix. On the GPU, the rectangular matrix to the right of the current panel contains intermediate data that will be used to compute Q and H. Soft errors in either one of these parts will cause the factorization to give an incorrect result. We need to detect and correct soft errors in both the CPU memory and the GPU memory. The algorithm we propose in this work combines the advantage of the ABFT technique and the diskless checkpointing technique. The algorithm also



(a) Partition (b) Error location (53, 16)

(c) Error location (31, 127) (d) Error location (63, 127)

Fig. 2. Propagation pattern of errors at different locations

uses reverse computation to roll back the program data to a previous state.

Depending on the location of the soft error, an error has different impacts on the result of the factorization. Figure 2 shows the impact of a soft error when it happens in three different locations. In this example, the matrix size N is set to 158, and the block size is 32. In all three figures, the soft error is injected when the first iteration has finished, and the second iteration has not yet started. Figure 2(a) is the partitioning of the matrix. Each of the following three figures shows the heat map of the difference matrix between the error-free result and the result when an error has happened during the factorization. Black means the difference is 0. Other colors mean the difference is bigger than 0, with each color representing a magnitude range. In Figure 2(b), the error occurs at location $(53, 16)$. This location is marked by an x in region 3 on the left in Figure 2(a). This error does not propagate as the factorization proceeds. We can see that in the final result of the factorization there is still only one incorrect element (shown as the white dot in the upper left part of the matrix). In Figure 2(c), the error happens at location $(31, 127)$. This location is marked by an x in region 1 shown in Figure 2(a). This soft error propagates row-wise, and pollutes the entire row in H when the factorization completes. In Figure 2(d), the error occurs at location $(63, 127)$. This location is marked by an x in region 2, shown in Figure 2(a). An error in this region causes the most damage among the three scenarios. When the factorization completes, almost all the elements after column 32 in H are polluted, and many elements after column 32 in Q are polluted.

B. Encoding the Input Matrix

To recover from an error we need redundant information. We add redundancy to the input matrix by appending an extra column at the right side of the matrix, and an extra row at the bottom of the matrix. An element in the extra column is the summation of all the elements in the same row in the input matrix. Similarly, an element in the extra row is the summation of all the elements in the same column of the original matrix. Figure 3 shows the initial state of the encoded input matrix.

Fig. 3. The encoded initial matrix

We define the following notations: A_{r_chk} is the column of row checksums on the right side of the original matrix; A_{c_chk} is the row of column checksums at the bottom of the original matrix. A_{re} is the original matrix appended with A_{r_chk} on the right side (*re* for *rowwise encoded*). A_{ce} is the original matrix appended with A_{c_chk} at the bottom (*ce* for *columnwise encoded*). A_{fe} is the original matrix appended with both A_{r_chk} and A_{c_chk} (*fe* for *fully encoded*).

C. The Fault Tolerant Algorithm

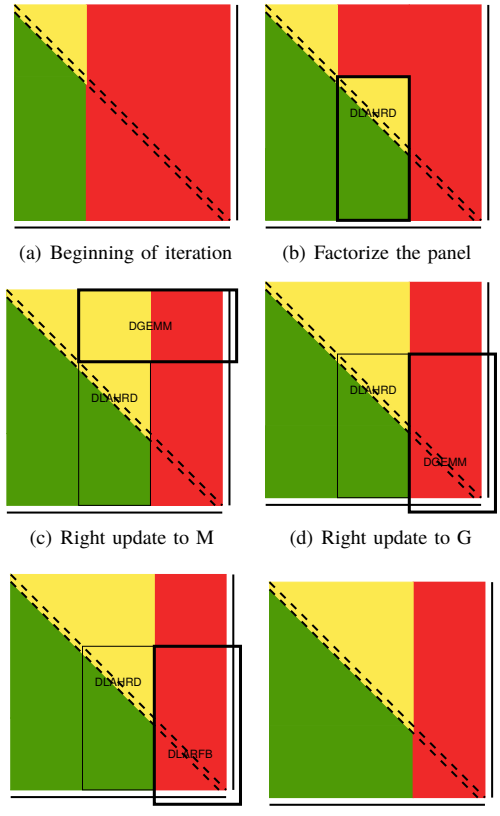
In this section, we present and explain our soft error tolerant Hessenberg reduction algorithm. e is an all one vector: $e = (1, 1, \cdots, 1, 1)^\top$. Algorithm 3 is the pseudocode for the fault tolerant algorithm.

The input matrix resides on the host side when the algorithm begins; in Algorithm 3 line 1 sends the input matrix to the GPU. Line 2 encodes the input matrix to obtain A_{fe}. Starting from line 3 the algorithm enters a **for** loop, this **for** loop iterates over the block columns of A. In each loop the algorithm first sends the lower part (the part marked by the black box in Figure 4(b)) of the next panel to the CPU from the GPU in line 4. In line 6 and line 7 the algorithm computes the column checksums for matrix Y and matrix V. This procedure requires two GEMV operations on the GPU. Line 8 applies the right update to matrix M_{re}. This line corresponds to Figure 4(c), and matrix M_{re} is the matrix marked by the black box in the figure. Line 9 and line 10 (in red text) overlap with each other. Line 10 applies the right update to matrix G. This corresponds to Figure 4(d). Line 11 applies the left update from the panel to matrix G, and this operation is illustrated in Figure 4(e).

We prove that, after line 11 in Algorithm 3, the column of row checksums and the row of column checksums are still valid for the yellow part and the red part in Figure 4(f). The proof is presented in the next section.

Algorithm 3 Fault Tolerant Hybrid Hessenberg Reduction

1: Transfer matrix: A on the host $\rightarrow$ d_A on the GPU
2: Encode the input matrix, expand it with a checksum column and a checksum row.
3: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
4: Send the lower part of the next panel P to the host.
5: **MAGMA_DLAHR2**, return V, T and Y where $Y = [P, G]VT$
6: Obtain Y_{ce} by computing the column checksums of Y: $Y_{chk\_c} = trail(A)_{chk\_c} \cdot V$
7: Obtain V_{ce} by computing the column checksums of V: $V_{chk\_c} = e^\top \cdot V$
8: **DGEMM**: $M_{re} = M_{re} - MVTV_{ce}^\top$
9: Send the leftmost nb columns of M to the host asynchronously.
10: **DGEMM**: $G_{fe} = G_{fe} - Y_{ce}V_{ce}^\top$
11: **DLARFB**: $trail(A)_{fe} = trail(A)_{fe} - V_{ce}T^\top V^\top trail(A)$
12: Compute $S_{re} = \sum A_{re}(i)$ and $S_{ce} = \sum A_{ce}(i)$
13: **if** $|S_{re} - S_{ce}| > threshold$ **then**
14: Reverse the last left update and right update.
15: Correct the error.
16: **end if**
17: **end for**



(a) Beginning of iteration (b) Factorize the panel

(c) Right update to M (d) Right update to G

(e) Left update to G (f) End of iteration

Fig. 4. One iteration of FT_DGEHRD

Line 12 through line 16 check for the existence of a soft error. The algorithm corrects the error if there is any.

Line 12 computes the summations of the checksum row and the checksum column. Since they contain checksums of the same matrix data along different directions, the summation of each vector should equal each other. Taking rounding errors into consideration, we check the difference against a threshold. If the difference exceeds the threshold, we consider an error has happened. The threshold should be big enough to tolerate roundoff errors, at the same time it should be small enough to avoid false negatives. A proper choice of the threshold is a value larger than the machine epsilon by 2 to 3 orders of magnitude. At this point the soft error in the matrix element has propagated to both the checksum column and the checksum row, the checksums are not valid any more. Line 14 reverses the last left update and the last right update so that the checksum column and the checksum row, together with the matrix data, are restored to their states at the end of the previous iteration. The checksum relationship is made valid again. The reverse computation is possible because the intermediate data used to apply the last last left update and right update are still available at the end of the iteration. They will not be destroyed until the next panel factorization. The algorithm then enters the recovery procedure.

D. The Checksum Relationship

In this section, we prove the following theorem:

Theorem 1. *The checksum column on the right of matrix A and the checksum row at the bottom of matrix A are valid at the end of each iteration.*

Proof. 1) The checksum column and the checksum row are valid after line 2 since they are newly computed.

2) The checksum column and the checksum row are valid after the right update to the trailing matrix.

$$
\begin{aligned}
A_{fe} &= A_{fe} - \begin{bmatrix} A \\ e^\top A \end{bmatrix} VT \begin{bmatrix} V \\ e^\top V \end{bmatrix}^\top \\
&= \begin{bmatrix} A & Ae \\ e^\top A & 0 \end{bmatrix} - \begin{bmatrix} AVTV^\top & AVTV^\top e \\ e^\top AVTV^\top & e^\top AVTV^\top e \end{bmatrix} \\
&= \begin{bmatrix} (A - AVTV^\top) & (A - AVTV^\top)e \\ e^\top (A - AVTV^\top) & * \end{bmatrix}
\end{aligned}
$$

3) The checksum column and the checksum row are valid after the left update to the checksum.

$$
\begin{aligned}
A_{fe} &= A_{fe} - \begin{bmatrix} V \\ e^\top V \end{bmatrix} T^\top V^\top \begin{bmatrix} A & Ae \end{bmatrix} \\
&= \begin{bmatrix} A & Ae \\ e^\top A & 0 \end{bmatrix} - \begin{bmatrix} VT^\top V^\top A & VT^\top V^\top Ae \\ e^\top VT^\top V^\top A & e^\top VT^\top V^\top Ae \end{bmatrix} \\
&= \begin{bmatrix} (A - VT^\top V^\top A) & (A - VT^\top V^\top A)e \\ e^\top (A - VT^\top V^\top A) & * \end{bmatrix}
\end{aligned}
$$

4) According to Mathematical Induction, the checksum row and the checksum column are valid at the end of each iteration.

□

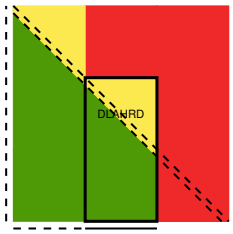


Fig. 5. Maintaining the checksums for Q

E. Protecting Q

The Q matrix contains the Householder vectors which were used to apply the similarity transformations to A. These Householder vectors are not protected by the checksums that encode the H matrix, we should provide protection for Q through other schemes. These Householder vectors are generated on the host side and stay there until the entire factorization finishes. They are not modified after they are generated. Moreover, they are not even read after the iteration in which they were generated finishes. Hence, it suffices to maintain a checksum for each row in order to correct an error. But just like the situation in detecting a soft error in H, we need both a checksum row and a checksum column to determine both the error column index j and error row index i. We keep the checksums for Q on the host. Q_{r_chk} is the rowwise checksum vector, and Q_{c_chk} is the columnwise checksum vector.

Figure 5 shows the process for generating and updating the checksums for the Q matrix. The dashed line on the left of the matrix is the column of row checksums for Q. When a new panel factorization is finished as the one shown in Figure 5, we compute the row checksums for the newly finished panel. Then the partial checksums for the panel are applied to the dashed line on the left so that the dashed line protects the entire green part. The dashed line at the bottom of the matrix is the row of column checksums for Q. This vector is computed segment by segment. When a new panel factorization is done on an nb wide panel, an nb long segment of the column checksums is also generated. The solid line segment at the bottom of the panel in Figure 5 is the newly generated column checksum segment for Q. This segment is never changed once generated.

Our algorithm overlaps the checksum generation for Q with the update to the trailing matrix on the GPU. The checksum generation involves two GEMV operations. GEMV is a level 2 BLAS operation which is a memory bound operation. We choose to perform the checksum generation on the CPU while the GPU is updating the trailing matrix. The CPU is idle in the non-fault tolerant MAGMA Hessenberg reduction algorithm, and our arrangement hides the time cost of the checksum generation.

F. Recovery

Once we have detected a soft error, we first determine the row index and the column index of the soft error before we can correct the error. We recalculate a checksum column A'_{r_chk} and a checksum row A'_{c_chk} of the current matrix (the yellow part and the red part in Figure 4(f)). Then we compare A'_{r_chk}

and A_{r_chk}, and the error row index i can be determined if $A'_{r\_chk}(i) \neq A_{r\_chk}(i)$. Similarly, the error column index j can be determined by comparing A'_{c_chk} and A_{c_chk}.

The erroneous element can be corrected using the formula $A(i,j) = A_{r\_chk}(i) - \sum_{k=1}^{k\leq n, k\neq j} A(i,k)$ or the formula $A(i,j) = A_{c\_chk}(j) - \sum_{k=1}^{k\leq n, k\neq i} A(k,j)$.

Since a soft error in the Q matrix does not propagate, we only examine the checksum relationship once, at the end of the factorization. The error detection and correction scheme is similar to those for the H matrix, except that it is carried out once at the end of the entire factorization instead of once per iteration.

V. Performance Evaluation

In this section, we give a formal analysis for the overhead of our fault tolerant Hessenberg reduction algorithm. The fault tolerant Hessenberg reduction algorithm performs extra floating point operations and extra data transfers between the host and the GPU in addition to those in the original MAGMA Hessenberg reduction. The fault tolerant algorithm also consumes extra storage to keep data redundancy. So, we evaluate the overhead in terms of extra FLOPS, extra communication, and extra storage. We denote the matrix dimension as N, the block size as nb, and the amount of floating point operations as $FLOP$.

After the algorithm transfers the input matrix to the GPU, the algorithm computes the global row checksums and the column checksums for the input matrix. This involves two DGEMV operations on the GPU: $A_{r\_chk} = Ae$ and $A_{c\_chk} = e^\top A$. The amount of floating operations:

$$FLOP_{init} = 2N(N+N-1) = 4N^2 - 2N.$$

In every iteration, the algorithm computes column checksums for matrix V. In the i-th iteration, the dimension of matrix V is $(N - nb \cdot i) \cdot nb$. The accumulated FLOP count over the course of the factorization is:

$$FLOP_{chkV} = \sum_{i=0}^{N/nb-1} nb \cdot (N - nb \cdot i + N - nb \cdot i - 1) = O(N^2).$$

The amount of floating point operations applied on the right hand side checksums is:

$$FLOP_{r\_chk} = \sum_{i=0}^{N/nb-1} \{(N - nb \cdot i) \cdot (nb + nb - 1) + N \cdot (nb + nb - 1) + nb \cdot [(N - nb \cdot i) + (N - nb \cdot i) - 1]\} = O(N^2).$$

The amount of floating point operations applied on the bottom checksums is:

$$FLOP_{c\_chk} = \sum_{i=0}^{N/nb-1} [(N - nb \cdot i)(nb + nb - 1) + (N - nb \cdot i)(nb + nb - 1)] = O(N^2).$$

The amount of floating point operations spent on intermediate results used by both row and column checksums is:

$$FLOP_{common} = \sum_{i=0}^{N/nb-1} nb \cdot (nb + nb - 1) = O(N).$$

The computation cost to detect the error in Algorithm 3 requires two dot product operations, one for the summation of the row checksums, and one for the summation of the column checksums. The total cost is given by:

$$FLOP_D = \sum_{i=0}^{N/nb-1} 2(N + N - 1) = O(N^2).$$

Adding all these together we get the total amount of extra floating point operations performed by the fault tolerant algorithm:

$$FLOP_{extra} = FLOP_{init} + FLOP_{chkV} + FLOP_{r\_chk} + FLOP_{c\_chk} + FLOP_{common} + FLOP_D = O(N^2).$$

The computation complexity of the Hessenberg reduction is $FLOP_{orig} \sim 10/3N^3$, so when there is no errors, the overhead of the fault tolerant Hessenberg reduction in terms of FLOP percentage is:

$$Overhead = \frac{FLOP_{orig}}{FLOP_{extra}} = \frac{O(N^2)}{10/3N^3} = \frac{3}{10}O(N^{-1}).$$

When N increases the overhead tends to: 0.

In order to locate the error, a vector of new row checksums and a vector of new column checksums need to be computed on the matrix consisting of the yellow part and the red part in Figure 2(a). The cost is given by:

$$FLOP_L = 2N(N + N - 1) = 4N^2 - 2N.$$

To correct the error requires a dot product and a subtraction:

$$FLOP_C = N - 2 + 1 = N - 1.$$

After an error has been detected, the algorithm performs a roll back by reverse update, which includes a reverse left update and a reverse right update. Then the pre-factorized panel is retrieved from the buffer, and the entire iteration is repeated after the error correction. The amount of overhead is a function of the size of the trailing matrix. Assume the error occurred in the j-th iteration, and we have:

$$\begin{aligned} FLOP_{redo} &= FLOP_{repeat} + FLOP_{panel} \\ &\approx N \cdot (N - j \cdot nb)(2nb - 1) + \\ &\quad (N - j \cdot nb) \cdot (N - j \cdot nb)(2nb - 1) \\ &\quad + (N - j \cdot nb) \cdot nb \cdot [(N - j \cdot nb) + (N - j \cdot nb) - 1] \\ &\quad + (N - j \cdot nb) \cdot nb \cdot (nb + nb - 1) \\ &= O(N^2). \end{aligned}$$

Compared with the computation cost of the original Hessenberg reduction, the extra FLOP introduced by the fault tolerant algorithm is very low. It tends to 0 when n increases.

TABLE I
DETAILED SPECIFICATION OF THE TEST PLATFORM.

	CPU	GPU
Processor model	Intel Xeon E5-2670	NVIDIA Tesla K40c
Clock frequency	2.6 GHz	745 MHz
Memory	62 GB	11.5 GB
Peak DP	10.4 Gflop/s	1.43 Tflop/s
BLAS/LAPACK	Intel MKL 11.2	CUBLAS 7.0.28
OS	CentOS 6.4	-
Compiler	gcc 4.4.7	nvcc 7.0 V7.0.27

The storage requirement of the fault tolerant Hessenberg reduction algorithm consists of a panel worth of work space for the intermediate result to update the trailing matrix, and four columns worth of space for the checksums:

$$S = nb \cdot N + 4 \cdot N$$

VI. EXPERIMENTS

In this section, we present the performance of our fault tolerant algorithm. Our testbed consists of an Intel Sandy Bridge-EP CPU and an NVIDIA Kepler GPU. The detailed specifications of the test platform are listed in Table I.

A. Performance Study

As shown in Figure 2(a), during the factorization the matrix is partitioned in three areas. We analyze the performance of our algorithm when the soft error occurs in each of the different areas, at different moments of the factorization. The Hessenberg reduction algorithm is application agnostic, different applications may use a different typical matrix size.

Figure 6(a) shows the performance overhead in the case where the soft error occurs in area 1 (see Figure 2(a)). This overhead includes setting up and maintaining the checksums, the reverse update to the trailing matrix, and the re-execution of the faulty iteration. Among all these costs, the most expensive step is the panel factorization when re-executing the faulty iteration. When the error occurs early in the factorization, the size of the panel which the algorithm re-factorizes is larger, and the performance overhead is also larger. The gray area in the figure indicates the range of the overhead depending on the moment when the single fault is introduced in Area 1. We can see that the overhead range remains small for all matrix sizes while the overhead exhibits a decreasing trend as the matrix size grows; at matrix size 10112×10112 the overhead is less than between 0.47% and 2.1% when one error occurs in Area 1.

Figure 6(b) shows the performance overhead of the fault tolerant algorithm when the soft error occurs in area 2 (see Figure 2(a)). Similar to Figure 6(a), the overhead is dependent on the moment when the error occurs. It maintains the same constant range and it exhibits the same decreasing trend as the matrix size grows. The performance overhead is between 0.61% and 2.15% at matrix size 10112×10112.

Figure 6(c) shows the performance overhead of the fault tolerant algorithm when the soft error occurs in Area 3 (see Figure 2(a)). In this case we can see that the performance overhead is smaller, closely following the overhead of the case without failures. There are two reasons for this phenomenon: the error detection and correction are only carried out once at the end of the factorization, and after an error is detected, only a dot product is necessary to correct the error. In contrast, an error in either area 1 or 2 requires a reverse update, a repeated panel factorization, and a trailing matrix update. We also observe that the uncertainty interval of the performance overhead is very small at all matrix sizes. No matter when the error occurred during the factorization, they are treated at the end with the same procedure, with the same minimalistic approach. Therefore they incur the same amount of overhead. Overall, these results indicate that our approach is a practical solution to ensure the correctness of the Hessenberg reduction with minimal overhead, and that this overhead consistently decreases as the size of the matrix increases.

B. Numerical Stability

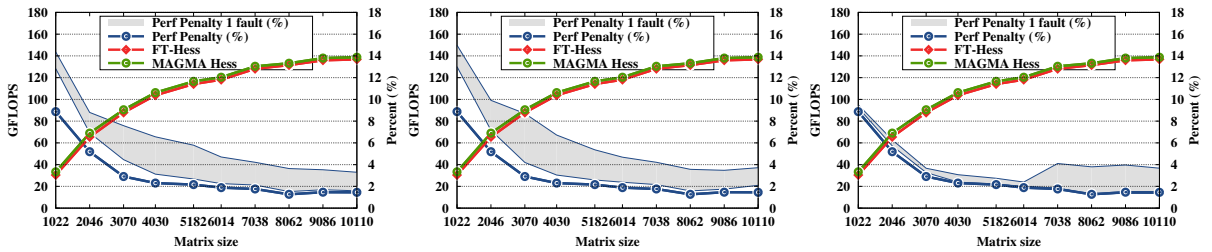
In this section, we investigate the numerical behavior of our fault tolerant Hessenberg algorithm compared with the non-fault tolerant algorithm.

The block Hessenberg reduction algorithm implemented in MAGMA is backward stable. The following residual is used to verify the factorization result:

$$r = \frac{\|A - QHQ^\top\|_1}{N\|A\|_1}$$

where A is the input matrix, and N is the matrix dimension. Table II shows the comparison of the residuals as obtained from the original MAGMA non-fault tolerant algorithm and our fault tolerant algorithm with one soft error.

The three main sections of the table indicate the location of the error, Area 1, Area 2, or Area 3. In each section, the letter appended to the name of the column indicates the moment when the error occurs, B for the beginning of the factorization, M for the middle, and finally, E for the end of the factorization. Finally, in the case of Area 3, all columns were collapsed into a single column as the residuals were identical. We can see that for every matrix size the residuals from Area 1 and Area 2 are on the same order of magnitude, with minimal variations, as the original MAGMA algorithm. In some cases, the fault tolerant algorithm even has a smaller residual than the fault free original algorithm. When the error was introduced on the left part of the matrix (i.e., Q, in Area 3), the final residuals are higher than their counterparts in the MAGMA routine, but they are still within the acceptable range. The extra amount of error compared with the classic algorithm is introduced by the encoding/recovery process. In the encoding phase, N elements (in a row or column) are added together to form one checksum element. In the recovery phase, $N - 1$ elements are subtracted from the checksum element. Both phases are implemented as dot products. We refer interested readers to [27] for a detailed discussion of rounding errors in dot products. Overall, these results are evidence that our fault tolerant Hessenberg reduction algorithm can successfully correct soft errors without degrading the stability of the original algorithm.



(a) Area 1 (b) Area 2 (c) Area 3

Fig. 6. Overhead of FT-Hess. The blue line is the overhead without failures, while the gray area is the uncertainty interval when one single error is introduced in a specific area (as described in Figure 2(a)).

TABLE II
NUMERICAL STABILITY A1, A2, A3

Matrix Size	MAGMA Hess	Area 1 FT-Hess B	Area 1 FT-Hess M	Area 1 FT-Hess E	Area 2 FT-Hess B	Area 2 FT-Hess M	Area 2 FT-Hess E	Area 3 FT-Hess B/M/E
1022	6.2529×10^{-18}	6.2764×10^{-18}	6.2520×10^{-18}	6.2540×10^{-18}	6.2764×10^{-18}	6.2520×10^{-18}	6.2540×10^{-18}	3.9780×10^{-16}
2046	2.6291×10^{-18}	2.6552×10^{-18}	2.6502×10^{-18}	2.6276×10^{-18}	2.6552×10^{-18}	2.6502×10^{-18}	2.6276×10^{-18}	1.6047×10^{-15}
3070	8.0088×10^{-18}	8.0023×10^{-18}	7.9987×10^{-18}	8.0066×10^{-18}	8.0023×10^{-18}	7.9987×10^{-18}	8.0066×10^{-18}	1.9576×10^{-15}
4030	8.4784×10^{-18}	8.4697×10^{-18}	8.4747×10^{-18}	8.4790×10^{-18}	8.4697×10^{-18}	8.4747×10^{-18}	8.4790×10^{-18}	1.9473×10^{-14}
5182	1.2012×10^{-17}	1.2024×10^{-17}	1.2008×10^{-17}	1.2011×10^{-17}	1.2024×10^{-17}	1.2008×10^{-17}	1.2011×10^{-17}	2.5166×10^{-15}
6014	1.5892×10^{-17}	1.5881×10^{-17}	1.5891×10^{-17}	1.5892×10^{-17}	1.5881×10^{-17}	1.5891×10^{-17}	1.5892×10^{-17}	4.3368×10^{-15}
7038	1.9573×10^{-17}	1.9580×10^{-17}	1.9571×10^{-17}	1.9571×10^{-17}	1.9580×10^{-17}	1.9571×10^{-17}	1.9571×10^{-17}	2.6158×10^{-14}
8062	3.7656×10^{-18}	3.7575×10^{-18}	3.7690×10^{-18}	3.7656×10^{-18}	3.7575×10^{-18}	3.7690×10^{-18}	3.7656×10^{-18}	8.9874×10^{-15}
9086	6.3745×10^{-18}	6.3814×10^{-18}	6.3736×10^{-18}	6.3746×10^{-18}	6.3814×10^{-18}	6.3736×10^{-18}	6.3746×10^{-18}	2.2618×10^{-14}
10110	1.7536×10^{-17}	1.7531×10^{-17}	1.7535×10^{-17}	1.7536×10^{-17}	1.7531×10^{-17}	1.7535×10^{-17}	1.7536×10^{-17}	2.4302×10^{-14}

C. Orthogonality of Q

In this section, we verify the orthogonality of matrix Q generated by our fault tolerant algorithm. As explained in Section III-A, we have $H = Q^T A Q$ where Q is an orthogonal matrix. We use the following residual to examine the orthogonality of Q:

$$r = \frac{\|QQ^\top - I\|_1}{N}.$$

I is the identity matrix, N is the matrix dimension. Table III shows the residuals from the non-fault tolerant MAGMA algorithm and residuals from our fault tolerant algorithm when one error occurs in different areas and different stages of the matrix. When the soft-error occurs in Area 1 and Area 2, all residuals are on the order of 10^{-17}, which is the same as the residuals from the MAGMA algorithm. When the soft-error occurs in Area 3, the residual is higher but still comparable to the residuals from MAGMA. So the orthogonality of Q is not damaged after the recovery from an error.

VII. Conclusions and Future Work

In this paper, we presented the design and analysis of a soft error resilient hybrid Hessenberg reduction algorithm, an algorithm capable of taking advantage of current and future hybrid architectures to ensure data correctness during an entire two-sided factorization. This goal is achieved by an attentive combination of the strengths of ABFT and diskless checkpointing to maintain data redundancy during the factorization. From an algorithmic perspective, our algorithm detects the soft errors on-line and corrects them before they have the opportunity to propagate to the rest of the matrix data, minimizing the cost of the recovery process. In the case of a soft error, our algorithm carries out a reverse computation to roll the program data back to a consistent state and then corrects the soft error. The overhead of our approach is very low since it mainly utilizes extra computation to detect and correct the error, and the amount of extra memory necessary for the checksum is minimal. The performance overhead of our fault tolerant algorithm compared to the non-fault tolerant MAGMA Hessenberg reduction reaches 0.56% when no errors occur, and reaches 0.61% when one error occurs. Another important capability of our fault tolerant algorithm is that it can detect and correct more than one consecutive error, making it a potential candidate for highly volatile environments. Moreover, the methodology highlighted in this paper is generic enough to be applicable to the entire spectrum of two-sided factorizations, as well as other similar algorithms. This applicability is on our list of things to explore in the near term as we plan to provide soft error resilience for the rest of the hybrid two-sided factorizations in MAGMA.

VIII. Acknowledgments

The authors would like to thank George Bosilca for discussing and reviewing this paper. The authors would like to thank the NSF for supporting this research through grants 0904952 and 1063019, and DOE INCITE through the Performance End Station PEAC Project – this research used resources of the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

TABLE III
ORTHOGONALITY OF Q A1, A2, A3

Matrix Size	MAGMA Hess	Area 1 FT-Hess B	Area 1 FT-Hess M	Area 1 FT-Hess E	Area 2 FT-Hess B	Area 2 FT-Hess M	Area 2 FT-Hess E	Area 3 FT-Hess
1022	3.65×10^{-17}	3.80×10^{-17}	3.41×10^{-17}	3.36×10^{-17}	3.64×10^{-17}	3.46×10^{-17}	3.35×10^{-17}	6.84×10^{-16}
2046	3.72×10^{-17}	3.61×10^{-17}	3.71×10^{-17}	3.65×10^{-17}	3.53×10^{-17}	3.64×10^{-17}	3.64×10^{-17}	2.76×10^{-15}
3070	3.62×10^{-17}	3.40×10^{-17}	3.57×10^{-17}	3.63×10^{-17}	4.61×10^{-17}	3.63×10^{-17}	3.65×10^{-17}	3.31×10^{-15}
4030	3.75×10^{-17}	3.40×10^{-17}	3.75×10^{-17}	3.75×10^{-17}	3.98×10^{-17}	3.81×10^{-17}	3.77×10^{-17}	3.28×10^{-14}
5182	4.59×10^{-17}	3.78×10^{-17}	3.63×10^{-17}	3.61×10^{-17}	3.92×10^{-17}	3.62×10^{-17}	3.62×10^{-17}	4.19×10^{-15}
6014	3.74×10^{-17}	3.71×10^{-17}	3.63×10^{-17}	3.62×10^{-17}	3.89×10^{-17}	3.60×10^{-17}	3.62×10^{-17}	7.19×10^{-15}
7038	4.10×10^{-17}	4.44×10^{-17}	4.51×10^{-17}	4.50×10^{-17}	4.00×10^{-17}	4.52×10^{-17}	4.51×10^{-17}	4.35×10^{-14}
8062	3.64×10^{-17}	3.31×10^{-17}	3.74×10^{-17}	3.74×10^{-17}	3.58×10^{-17}	3.77×10^{-17}	3.74×10^{-17}	1.49×10^{-14}
9086	3.64×10^{-17}	3.75×10^{-17}	4.22×10^{-17}	4.22×10^{-17}	4.08×10^{-17}	4.18×10^{-17}	4.22×10^{-17}	3.71×10^{-14}
10110	4.36×10^{-17}	4.20×10^{-17}	4.32×10^{-17}	4.29×10^{-17}	4.15×10^{-17}	4.30×10^{-17}	4.29×10^{-17}	4.05×10^{-14}

REFERENCES

[1] J. F. Ziegler and W. A. Lanford, "Effect of Cosmic Rays on Computer Memories," *Science*, vol. 206, no. 4420, pp. 776–788, 1979.

[2] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, 2005.

[3] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010.

[4] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 329–335, 2005.

[5] I. Haque and V. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010, pp. 691–696.

[6] P. Du, P. Luszczek, and J. Dongarra, "High performance dense linear system solver with soft error resilience," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 272–280.

[7] ——, "High performance dense linear system solver with resilience to multiple soft errors," *Procedia Computer Science*, vol. 9, no. 0, pp. 216 – 225, 2012.

[8] P. Du, P. Luszczek, S. Tomov, and J. Dongarra, "Soft error resilient QR factorization for hybrid system with GPGPU," in *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, ser. ScalA '11. New York, NY, USA: ACM, 2011, pp. 11–14.

[9] Y. Kim, J. S. Plank, and J. Dongarra, "Fault Tolerant Matrix Operations Using Checksum and Reverse Computation," in *6th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, October 1996, pp. 70–77.

[10] C.-Y. Chen and J. A. Abraham, "Fault-tolerant systems for the computation of eigenvalues and singular values," in *Advanced Algorithms and Architectures for Signal Processing I*, vol. 0696, April 1986, pp. 228–237.

[11] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, June 1984.

[12] F. T. Luk and H. Park, "Fault-tolerant matrix triangularizations on systolic arrays," *IEEE Trans. Comput.*, vol. 37, no. 11, pp. 1434–1438, November 1988.

[13] ——, "An analysis of algorithm-based fault tolerance techniques," *J. Parallel Distrib. Comput.*, vol. 5, no. 2, pp. 172–184, April 1988.

[14] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, August 10 2003, dOI: 10.1002/cpe.728.

[15] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22nd annual international conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 155–164.

[16] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 152–161.

[17] ——, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 69–78.

[18] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. The John Hopkins University Press, December 27 2012, iSBN-10: 1421407949, ISBN-13: 978-1421407944.

[19] G. W. Stewart, *Matrix Algorithms, Volume II: Eigensystems*, First ed. SIAM: Society for Industrial and Applied Mathematics, August 2001.

[20] C. Cao, G. Bosilca, T. Herault, and J. Dongarra, "Design for a Soft Error Resilient Dynamic Task-based Runtime," in *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, IEEE. Hyderabad, India: IEEE, 05/2015 2015.

[21] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, October 1998.

[22] S. Tomov, R. Nath, P. Du, and J. Dongarra, *MAGMA Users' Guide*, ICL, UTK, November 2009.

[23] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Third ed. Philadelphia, PA: SIAM, 1999.

[24] G. Quintana-Ortí and R. van de Geijn, "Improving the performance of reduction to Hessenberg form," *ACM Trans. Math. Softw.*, vol. 32, no. 2, pp. 180–194, June 2006.

[25] R. Schreiber and C. Van Loan, "A storage efficient WY representation for products of householder transformations," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, 1989.

[26] S. Tomov and J. Dongarra, "Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing," University of Tennessee Knoxville, Tech. Rep. UT-CS-09-642, 2009.

[27] A. M. Castaldo, R. C. Whaley, and A. T. Chronopoulos, "Reducing floating point error in dot product using the superblock family of algorithms," *SIAM J. Scientific Computing*, vol. 31, no. 2, pp. 1156–1174, 2008.