Transient Error Resilient Hessenberg Reduction on GPU-based Hybrid Architectures

Yulu Jia
University of Tennessee
Knoxville

Piotr Luszczek
University of Tennessee
Knoxville

Jack J. Dongarra
University of Tennessee
Knoxville, Oak Ridge National Laboratory and University of Manchester

1. ABSTRACT

Graphics Processing Units (GPUs) are gaining wide spread usage in the field of scientific computing owing to the performance boost GPUs bring to computation intensive applications. The typical configuration is to integrate GPUs and CPUs in the same system where the CPUs handle the control flow and part of the computation workload, and the GPUs serve as accelerators carry out the bulk of the data parallel compute workload. In this paper we design and implement a soft error resilient Hessenberg reduction algorithm on GPU based hybrid platforms. Our design employs algorithm based fault tolerance technique, diskless checkpointing and reverse computation. We detect and correct soft errors on-line without delaying the detection and correction to the end of the factorization. By utilizing idle time of the CPUs and overlapping both host side and GPU side workloads we minimize the observed overhead. Experiment results validated our design philosophy. Our algorithm introduces less than 2% performance overhead compared to the non-fault tolerant hybrid Hessenberg reduction algorithm.

2. INTRODUCTION

A transient error is an error in a signal or data element which is transient and caused by factors other than permanent component failures. Reasons for transient errors range from alpha particles from package decay, cosmic rays and thermal neutrons. Cosmic rays were shown to be the most prevailing source for transient errors among these sources [25]. Transient errors may happen in communication links, digital logic and other places but the most common situation is errors in semiconductor storage.

Both GPUs and traditional CPUs and their associated memory are prone to suffer from transient errors. CPU designs continue to follow Moore's law 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. For example, the Intel Ivy Bridge processors are fabricated using the 22 nanometer process technology. Smaller feature size combined with lower voltage to maintain transistor states make it easier to change the transistor state. The critical charge Q_{crit} which is the lowest electron charge needed to change the logical level decreases when 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 more and more popularity in the scientific computing community due to the sizable acceleration they provide to computation intensive applications. The source of the huge acceleration is the large amount of data processing transistors inside the GPGPUs. The number of transistors per unit area in GPGPUs continues to grow according to Moore's law. The latest NVIDIA GeForce GTX Titan GPGUP is fabricated using the 28 nanometer process and has 7.1 billion transtors on a 551 mm^2 die. The same reasons that make conventional CPUs suffer from transient errors are also present in the case of GPUs.

Transient errors are becoming a challenge in real world applications. Both CPU main memory and GPU memory are DRAMs (Dynamic Random-access Memory). Baumann [2] has reported that the soft error rate (SER) of DRAM is between 1k FIT/chip - 10K FIT/chip range, and stays at the same level over 7 generations of DRAMs. FIT(failure in time) is the measurement unit of SER, one FIT is one soft error in 10^9 device hours. Michalak et al. [16] reported that the ASC Q supercomputer at Los Alamos National Laboratory experience an average of 51.7 soft errors per week over a period of 7 weeks from September 2004 to October 2004. Haque et al. [10] 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 higher than 1×10^{-5} and a large population with a mean of 2×10^{-5}. Jacob et at. [12, page 33] reported that at 130 nm process generation DRAM has an SER of 1000 FIT and SRAM 100, 000 FIT.

Soft error and its impact in linear algebra have attracted much attention of researchers recently. Du et al. [6, 7] proposed an algorithm to tolerate soft errors in the High Performance LINPACK Benchmark (HPL) [5]. 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 an post processing approach in which the erroneous result is corrected through post processing after the regular factorization. Bronevetsky and Supinski [3] 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. In [20] Shantharam et al.

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. In [21] Shantharam et al. 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 happens. Chen and Abraham [4] designed a concurrent error detection scheme for transient errors in the computation of eigenvalues on systolic processor arrays using the QR algorithm [9, 22] (not to be confused with the QR factorization). Kim et al. [13] designed a fault tolerant Hessenberg reduction algorithm to recover from processor failures on networks of workstations (NOWs).

In this paper we design and implement a soft error resilient hybrid Hessenberg reduction algorithm on GPU enabled hybrid architectures. We take advantage of diskless checkpointing, ABFT and reverse computation techniques to achieve soft error tolerance while introducing very little overhead into the non fault tolerant Hessenberg reduction algorithm. We further minimized the observable overhead by carefully overlapping workloads on the host side and the GPU side. Our fault tolerant algorithm is able to detect and correct one or more simultaneous soft errors. Once the algorithm has corrected the simultaneous errors, it continues as normal and is ready to detect and correct subsequent soft errors. Unlike the post-processing scheme for LU and QR in [6, 7, 8], our algorithm detects soft errors on the fly at the end of each iteration. After the algorithm has detected the errors, it corrects them right away so the errors don't propagate and contaminate other matrix elements. While the post-processing scheme can only correct 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 errors if the error positions don't form a rectangle. Our fault tolerant Hessenberg algorithm also can detect and correct as many subsequent soft errors as they occur.

The remainder of the paper is organized as follows: in Section 3 we survey related work, then in Section 4 we explain the Hessenberg reduction algorithm and its implementation in the MAGMA framework. Section 5 describes our soft error resilient hybrid Hessenberg reduction algorithm in detail. Section 7 presents experiment results of the algorithm and provides a theoretical analysis for the performance. Section 8 summarizes our work.

3. RELATED WORK

There have been many research efforts in soft error resilient dense linear algebra operations. Abraham et al. [11] first brought forward the idea of algorithm based fault tolerance (ABFT) for matrix computations such as matrix-matrix addition, matrix-matrix multiplication, scalar product and LU-decomposition in multiple processor systems. They encoded the input matrix to add data redundancy with checksums. The checksum relationship between the matrix data and the checksum data is preserved throughout the matrix operation in consideration. Error detection and location are achieved by examining the checksum relationship at the end of the matrix operation. To correct the error a rollback is performed and the matrix data are restored to the step right before the error has happened. Then the erroneous element could be recovered using the checksum and the fault-free elements. The approach they proposed is a post-processing one in that the error detection, location and correction happen after the matrix operation has finished. Luk and Park [15, 14] extended and improved the ABFT approach to tolerate soft errors in the LU factorization, QR factorization and Gaussian elimination with pairwise pivoting. The improved approach also works in a post-processing manner. After the matrix operation is finished, the error detection procedure is performed. If a soft error has been detected, the error is projected back as a rank-1 perturbation to the original matrix data. The correct result is obtained through a matrix factorization update process to the erroneous result. The advantage of this error model is that no rollback is required, and the time point when the soft error happened is irrelevant to the recovery process. No matter when the error happened, it is always projected as a rank-1 perturbation to the original matrix which produces the same result as the actual error. The drawback of this error model is that it can only correct one soft error during the whole factorization.

Plank et al. [13] 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 since 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 [4] 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 are taken into consideration to make the detection of errors very efficient.

There are also research efforts on soft error resilience in sparse linear algebra. Shantharam et al. [21] designed a soft error resilient preconditioned conjugate gradient method for sparse linear systems. They adapted the classic ABFT technique for sparse matrices. Their method removed one SpMV operation and replaced it with a dot product. The SpMV operation is the most computation intensive routine in an iterative method. By removing this SpMV operation they are able to reduce the extra computation and reduce the overhead incurred by the irregular memory accesses in the SpMV operation.

In [17] Plank et al. 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 Matric Algebra on GPU and Multicore Architectures project (MAGMA) [24] is a dense linear algebra library for hybrid architectures with GPUs. The library provides equivalent functionalities to LAPACK [1] 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 computation 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 LAPCK 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.

4. HESSENBERG REDUCTION ON GPU ENABLED HYBRID ARCHITECTURES

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

4.1 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 P_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 $Q_{i-1}^{-1} \cdots Q_1^{-1} A Q_1 \cdots Q_{i-1}$.

4.2 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 [18] greatly reduced the arithmetic intensity by grouping nb Householder reflectors and apply the group to A at the same time.

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

where I is the identity matrix, V is an $N \times nb$ matrix composed of the Householder vectors, T is an $nb \times nb$ upper triangular matrix. This representation of U_1 is called the *compact WY representation* [19]. 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 its 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.

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**

4.3 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 [23]. The input matrix A is stored in LAPACK layout, matrix elements are stored contiguously in column major format. The matrix is logically divide into block columns, each block column is of size $N \times nb$. The matrix entries below the first subdiagonal are overwritten with the final Q matrix. The upper part of the matrix is overwritten with the final H matrix when the factorization completes. The hybrid algorithm keeps a copy of the matrix in the GPU memory, all the updates to the trailing matrix are performed by the GPU in its own memory. The panel factorization is assigned to the CPU, 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 5 is an asynchronous data transfer, 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 computation, using asynchronous data transfer here 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 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. They reside on the host side and will not be modified again. The red rectangular is the trailing matrix which will be factorized and updated in this iteration. The first nb columns of the red part is 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 red 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, 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.

Algorithm 2 Hybrid Hessenberg Reduction

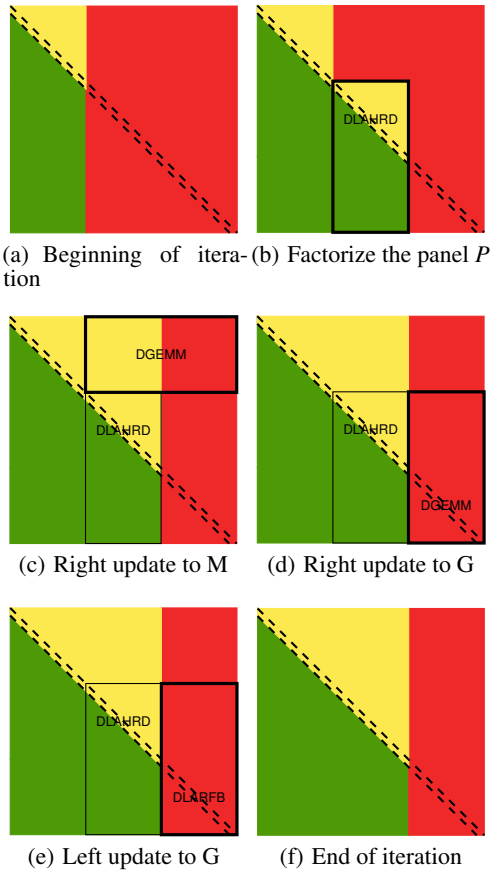
1: Transfer matrix: A on the host $\rightarrow$ d_A on the GPU
2: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
3: Send the lower part of the next panel P_{next} to the host.
4: **MAGMA_DLAHR2**, return $V, T and Y$ where $Y = [P, G]VT$
5: **DGEMM**: $M = M - MVTV^\top$
6: Send the leftmost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

5. SOFT ERROR RESILIENT HESSENBERG REDUCTION ALGORITHM

something here.

5.1 Failure Model

In this work we consider soft errors which are temporary faults in the data matrix, the factorization is oblivious to the error and continues as usual. Without loss of generality, we assume only one error happens at a single time point. Later we will provide an analysis on the case where more than one soft error happen simultaneously and how our soft error resilient algorithm applies to that case. We allow



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

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

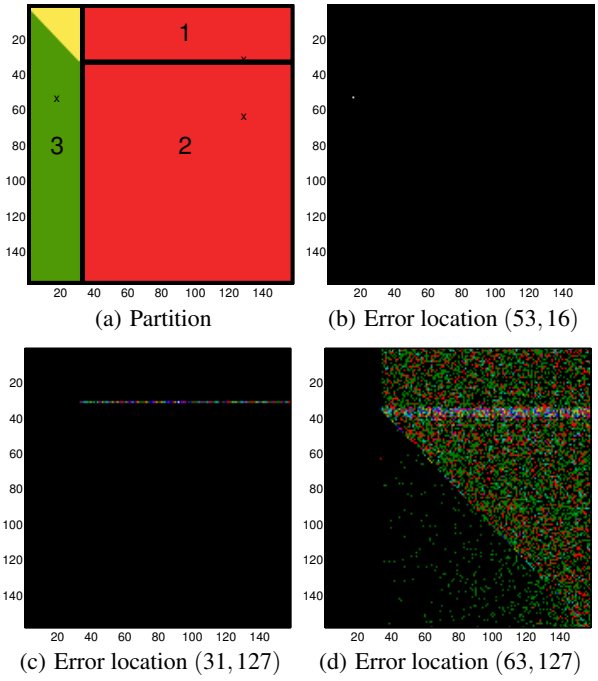
(e) Left update to G (f) End of iteration

Figure 1: One iteration of DGEHRD

the situation where multiple non simultaneous soft errors happen. Errors which happen in the same iteration are considered simultaneous errors.

In the MAGMA Hessenberg reduction algorithm, both the CPU and GPU carry out computation. The CPU is responsible for the panel factorization, the GPU is responsible for 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 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 ABFT technique and diskless checkpointing technique. The algorithm also 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 an soft error when it happens in three different location. In this example, the matrix size N is set to 158, 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 started yet. 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 color means the difference is 0. Other colors mean the difference is bigger than 0. 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 propgates rowwise, and polluted 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.



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

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

Figure 2: Propagation pattern of errors at different locations

5.2 Encoding the Input Matrix

To recover from an error we need redundancy 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 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.

We define the follow 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 appended with A_{r_chk} on the right side (*re* for *rowwise encoded*). A_{ce} is the original appended with A_{c_chk} at the bottom (*ce*

Figure 3: The encoded initial matrix

for *columnwise encoded*). A_{fe} is the original matrix appended with both A_{r_chk} and A_{c_chk} (*fe* for *fully encoded*).

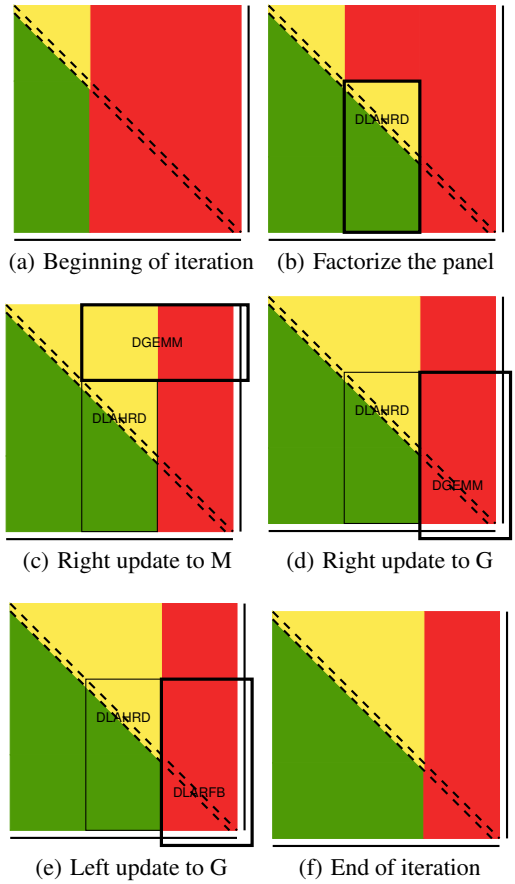
5.3 The Fault Tolerant Algorithm

In this subsection we present and explain two soft error tolerant Hessenberg reduction algorithms. They differ in the way to detect soft errors. e is an all one vector: $e = (1, 1, \cdots, 1, 1)^\top$. Algorithm 3 and Algorithm 4 are the pseudocodes for the two fault tolerant algorithms.

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_{next} to the host.
5: **MAGMA_DLAHR2**, return $V, TandY$ 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**

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), matrix M_{re} is the matrix marked by the black box in the figure. Line 10 applies the right update to matrix G. This corresponds to Figure 4(d). Line 11 applies the left update from the



(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

Figure 4: One iteration of FT_DGEHRD

panel to matrix G, 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 subsection.

Line 12 through line 16 check for the existence of soft errors. 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 the rounding error into consideration, we check the difference against a threshold. If the difference exceeds the threshold we consider an error has happened. 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 won't be destroyed until the next panel factorization. The algorithm then enters the recovery procedure.

Algorithm 4 is different from Algorithm 3 in error detection. In line 3 the algorithm computes the summation S_{orig} of the diagonal

elements of the input matrix A. In line 13 the algorithm computes the summation S_{now} of the diagonal elements of the finished part of matrix H and the trailing matrix. In the Hessenberg reduction, a similarity transformation is applied to A in every iteration. Similarity transformations to a matrix preserves the eigenvalues of the matrix, which means the summation of the diagonal elements of the matrix is invariant. explain this better Algorithm 4 uses this property to check for soft errors. If S_{now} and S_{orig} are equal within the limit of rounding error, we say no error has happened. Otherwise the algorithm enters the recovery procedure.

5.4 The Checksum Relationship

In this subsection 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. We use Algorithm 3 as an example in the proof. Algorithm 4 updates the matrix and the checksums in exactly the same way as Algorithm 3 so the proof is also correct for Algorithm 4.

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.

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

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

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

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

□

5.5 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 an soft error in H, we need both a checksum row and a checksum column to determine the 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, 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 1 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, our arrangement hides the time cost of the checksum generation.

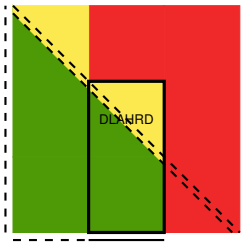


Figure 5: Maintaining the checksums for Q

when done explaining the algorithm, prove the checksum relationship invariable. There is still something missing in the algorithm. Computing the checksum of the panel and sending it to the GPU.

5.6 Recovery

Once we have detected an 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}, 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 formua $A(i,j) = A_{c\_chk}(j) - \sum_{k=1}^{k\leq n, k\neq i} A(k,j)$

Since an 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, only that it is carried out once instead of once per iteration.

6. 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, 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:

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

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

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

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

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

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

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

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

$$\begin{aligned} & FLOP_{extra} \\ &= FLOP_{init} + FLOP_{chkV} + FLOP_{r\_chk} + FLOP_{c\_chk} + FLOP_{common} \\ &= O(N^2) \end{aligned}$$

The computation complexity of Hessenberg reduction is $FLOP_{orig} \sim 10/3N^3$, so the overhead of the fault tolerant Hessenberg reduction in terms of floating point operation percentage is:

$$\begin{aligned} Overhead &= \frac{FLOP_{extra}}{FLOP_{extra}} \\ &= \frac{O(N^2)}{10/3N^3} \\ &= \frac{3}{10} O(N^{-1}) \end{aligned}$$

When N increases the overhead tends to: 0

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

$$FLOP_{D\_alg1} = \sum_{i=0}^{N/nb-1} 2(N+N-1) = O(N^2)$$

The computation cost to detect the error in Algorithm 4 only requires one dot product operation since it only computes the summation of the diagonal elements. The cost is given by:

$$FLOP_{D\_alg2} = \sum_{i=0}^{N/nb-1} (N+N-1) = O(N^2)$$

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 the yellow part and the red part in Figure 2(a). The cost is given by:

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

Table 1: Detailed specification of the test platform.

	CPU	GPU
Process model	Intel Xeon E5-2670	NVIDIA Tesla K20c
Clock frequency	2.6 GHz	705.5 MHz
Memory	62 GB	4799.6 MB
Peak DP	10.4 Gflop/s	1.17 Tflop/s
BLAS/LAPACK	Intel MKL 11.0	CUBLAS 3.2
OS	Red Hat 4.4.6-4	-
Compiler	gcc version 4.4.6	nvcc 5.0 V0.2.1221

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 algorithms performs a roll back by a 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 happed in the j-th iteration, 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.

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:

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

7. EXPERIMENTS

In this section we present performance results of our fault tolerant algorithm of a series of experiments. The test platform we use consists an Intel Sandy Bridge-EP CPU and a NVIDIA Kepler GPU. The specifications of the test platform are listed in Table 1 in detail.

7.1 Performance Study

We show and analyze the performance of our algorithm when the soft error occurs in different regions of the data matrix, and at different time points of the factorization.

Figure 6 shows the overhead in terms of performance in the case where the soft error occurs in area 1 (see Figure 2(a)). Figure 6(a) shows the case when no errors happens. We can see that the performance overhead is less than 1% for most of the matrix sizes, and the percentage of the overhead keeps decreasing as the matrix size increases. The overhead in this case includes the computations on the checksums, the transfer of newly generated checksums for Q to the GPU, and the error detection in each iteration. Figure 6(b) shows the case where the error occurs in the beginning of the factorization (the second iteration). The performance overhead in this case includes every cost in Figure 6(b), and also includes a reverse update to the trailing matrix, and a repetition of the faulty iteration. Among all these costs, the most expensive step is the panel factorization when repeating the faulty iteration. The reason is that the panel factorization involves lots of GEMV operations on the GPU. The GPU is fast for data parallel tasks, GEMV is memory bound and is very slow compared to the trailing matrix update (DGEMM). So this extra panel factorization incurs a big amount overhead compared to other overhead contributors. Moreover, the size of the panel which the algorithm re-factorizes is the largest panel (of size $N \times nb$). For these reasons, we observe the biggest overhead when the error happens in the first iteration. Figure 6(c) shows the case where the error occurs in the middle of the factorization (in iteration $(N/nb)/2$). The overhead contributors in this graph are the same as in Figure 6(b), the difference is in this graph the repeated iteration has a smaller panel and a smaller trailing matrix. The overhead is reduced significantly because of this. We can see that at the matrix size 10112×10112 the overhead is 1.1% as compared to 2.1% in Figure 6(b). Figure 6(d) shows the case where the error occurs in the end of the factorization (the second last iteration). The overhead in this graph drops to 0.47% at matrix size 10112×10112.

Figure 7 shows the performance overhead of the fault tolerant algorithm when the soft error occurs in area 2 (see Figure 2(a)). Figure 7(b), Figure 7(c), and Figure 7(d) show the cases where the error happens in the beginning, middle and end of the factorization respectively. Similar to Figure 6, we observe the trend in which the later the error occurs, the smaller the relative performance overhead is. Also the relative performance overhead keeps decreasing as the matrix size increases, the penalty is 0.61% at matrix size 10112×10112.

Figure 8 shows the performance overhead of the fault tolerant algorithm when the soft error occurs in area 3 (see Figure 2(a)). Figure 8(b), Figure 8(c), and Figure 8(d) show the cases where the error happens in the beginning, middle and end of the factorization respectively. We can see that the performance overhead in this case is very small. Actually, the performance overhead with one error happens is close to the case without any failures. There are two reasons for this phenomenon. Firstly, the error detection and correction are only carried out once at the end of the factorization.Secondly, after an error is detected, only a dot product operation is required in order to correct the error. In contrast, an error in either area 1 or 2 requires a reverse update, and a repeated panel factorization and trailing matrix update (includes a left update and a right update). We also observe that the time point when the error happens does not affect the observed overhead. No matter when the error happened during the factorization, they are only treated at the end with the same procedure. Therefore they incur the same amount of overhead.

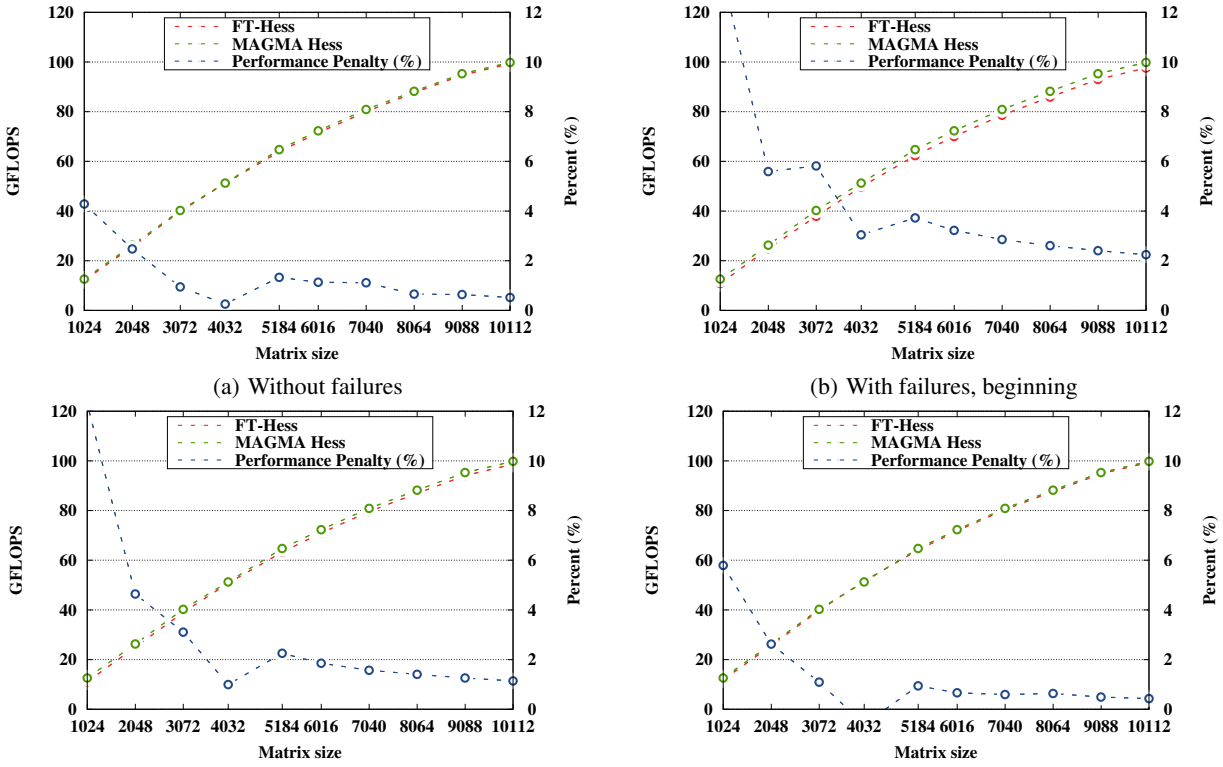
7.2 Numerical Stability

In this subsection 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, N is the matrix dimension. Table 2



(a) Without failures

(b) With failures, beginning

(c) With failures, middle

(d) With failures, end

Figure 6: Overhead of FT-Hess without failures and with one failure. A1

through Table 4 show the comparison of the residuals as obtained from our fault tolerant algorithm with one soft error and the original MAGMA non-fault tolerant algorithm without soft errors.

Table 2 shows the case in which the error occurs in area 1 of the matrix. The first column of residuals are obtained from the MAGMA routine, the second column of residuals are obtained when the soft error happens in the beginning of the factorization, the third column of residuals are obtained when the error occurs in the middle of the factorization, the fourth column of residuals are obtained when the soft error occurs at the end of the factorization. We can see that for every matrix size the residuals from both algorithms are on the same order. They only vary slightly in magnitude. In some cases the fault tolerant algorithm even has a smaller residual than the fault free original algorithm. This phenomenon is an evidence that our fault tolerant Hessenberg reduction algorithm can successfully correct soft errors without degrading the stability of the original algorithm.

Table 3 shows the residuals from the fault tolerant algorithm in the cases where one soft error occurs and is corrected. In this table the soft error happens in area 2 of the matrix (see Figure 2(a)). The second column of residuals shows the case in which the soft error happens in the beginning of the factorization, the third column shows the case in which the soft error occurs in the middle of the factorization. The fourth column of residuals shows the case where the soft error happens near the end of the factorization. The residuals shown in this table are also only different in magnitude from the fault free original algorithm. This is consistent with the observation we made from Table 2, and further confirms our claim that our fault tolerant Hessenberg reduction algorithm does not introduce numerical degradation.

Table 4 shows the residuals from the fault tolerant Hessenberg algorithm when the soft error happens in the left part of the matrix i.e., Q. In this case the final residuals obtained are higher than their counterparts in the MAGMA routine, but they are still within the acceptable range. We suspect that this is caused by rounding errors when calculating the checksum of a large number of elements. Further investigation is needed to interpret and mitigate this effect.

8. CONCLUSION

In this paper we design and implemented a soft error tolerant hybrid Hessenberg reduction algorithm. Our algorithm can detect and correct soft errors which occurred during the course of the factorization. Our fault tolerant algorithm performs error detection and correction on-line and completely prevents the error from propagating. Our algorithm combines the strength of ABFT and diskless checkpointing to maintain data redundancy during the factorization. In the case of an soft error, our algorithm carries out a reverse computation to roll back the program data back to a consistent state and then correct the soft error. The overhead of our approach is very low since it mainly utilizes extra computation to detect and correct the error, the amount of extra memory access is minimized. The performance overhead of our fault tolerant algorithm compared to the non-fault tolerant MAGMA Hessenberg reduction reaches 0.56% when no errors happens, and reaches 0.61% when one error happens. Our fault tolerant algorithm can detect and correct more than one consecutive errors. Our methodology is general enough so that we can apply it to other two-sided factorizations. In the future we

Table 2: Numerical Stability A1 top

Matrix Size	MAGMA Hess	FT-Hess B	FT-Hess M	FT-Hess E
1022	6.252980×10^{-18}	6.369269×10^{-18}	6.243481×10^{-18}	6.240962×10^{-18}
2046	2.629116×10^{-18}	2.630569×10^{-18}	2.644486×10^{-18}	2.625012×10^{-18}
3070	8.008891×10^{-18}	8.014784×10^{-18}	8.010240×10^{-18}	8.008317×10^{-18}
4030	8.478453×10^{-18}	8.465195×10^{-18}	8.474643×10^{-18}	8.476308×10^{-18}
5182	1.201234×10^{-17}	1.201695×10^{-17}	1.201397×10^{-17}	1.201272×10^{-17}
6014	1.589200×10^{-17}	1.588111×10^{-17}	1.589051×10^{-17}	1.589241×10^{-17}
7038	1.957300×10^{-17}	1.957639×10^{-17}	1.957776×10^{-17}	1.957325×10^{-17}
8062	3.765650×10^{-18}	3.763755×10^{-18}	3.766010×10^{-18}	3.764728×10^{-18}
9086	6.374555×10^{-18}	6.379111×10^{-18}	6.368058×10^{-18}	6.374630×10^{-18}
10110	1.753614×10^{-17}	1.753520×10^{-17}	1.753726×10^{-17}	1.753650×10^{-17}

Table 3: Numerical Stability A2 bottom

Matrix Size	MAGMA Hess	FT-Hess B	FT-Hess M	FT-Hess E
1022	6.252980×10^{-18}	6.276436×10^{-18}	6.252014×10^{-18}	6.254011×10^{-18}
2046	2.629116×10^{-18}	2.655281×10^{-18}	2.650204×10^{-18}	2.627617×10^{-18}
3070	8.008891×10^{-18}	8.002311×10^{-18}	7.998726×10^{-18}	8.006659×10^{-18}
4030	8.478453×10^{-18}	8.469741×10^{-18}	8.474768×10^{-18}	8.479033×10^{-18}
5182	1.201234×10^{-17}	1.202468×10^{-17}	1.200824×10^{-17}	1.201148×10^{-17}
6014	1.589200×10^{-17}	1.588116×10^{-17}	1.589105×10^{-17}	1.589267×10^{-17}
7038	1.957300×10^{-17}	1.958012×10^{-17}	1.957135×10^{-17}	1.957146×10^{-17}
8062	3.765650×10^{-18}	3.757575×10^{-18}	3.769064×10^{-18}	3.765698×10^{-18}
9086	6.374555×10^{-18}	6.381444×10^{-18}	6.373613×10^{-18}	6.374615×10^{-18}
10110	1.753614×10^{-17}	1.753192×10^{-17}	1.753503×10^{-17}	1.753686×10^{-17}

Table 4: Numerical Stability A3 left

Matrix Size	MAGMA Hess	FT-Hess B	FT-Hess M	FT-Hess E
1022	6.252980×10^{-18}	3.978038×10^{-16}	3.978038×10^{-16}	3.978038×10^{-16}
2046	2.629116×10^{-18}	1.604774×10^{-15}	1.604774×10^{-15}	1.604774×10^{-15}
3070	8.008891×10^{-18}	1.957680×10^{-15}	1.957680×10^{-15}	1.957680×10^{-15}
4030	8.478453×10^{-18}	1.947312×10^{-14}	1.947312×10^{-14}	1.947312×10^{-14}
5182	1.201234×10^{-17}	2.516603×10^{-15}	2.516603×10^{-15}	2.516603×10^{-15}
6014	1.589200×10^{-17}	4.336840×10^{-15}	4.336840×10^{-15}	4.336840×10^{-15}
7038	1.957300×10^{-17}	2.615821×10^{-14}	2.615821×10^{-14}	2.615821×10^{-14}
8062	3.765650×10^{-18}	8.987489×10^{-15}	8.987489×10^{-15}	8.987489×10^{-15}
9086	6.374555×10^{-18}	2.261822×10^{-14}	2.261822×10^{-14}	2.261822×10^{-14}
10110	1.753614×10^{-17}	2.430251×10^{-14}	2.430251×10^{-14}	2.430251×10^{-14}

will provide soft error resilience for the rest of the hybrid two-sided factorizations in MAGMA.

Acknowledgments

9. REFERENCES

[1] 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*. SIAM, Philadelphia, PA, Third edition, 1999.

[2] R. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, 2005.

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

[4] 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*, volume 0696, pages 228–237, April 1986.

[5] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[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*, pages 272–280, 2011.

[7] P. Du, P. Luszczek, and J. Dongarra. High performance dense linear system solver with resilience to multiple soft errors. *Procedia Computer Science*, 9(0):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*, ScalA '11, pages 11–14, New York, NY, USA, 2011. ACM.

[9] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The

John Hopkins University Press, 4th edition, December 27 2012. ISBN-10: 1421407949, ISBN-13: 978-1421407944.

[10] 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*, pages 691–696, 2010.

[11] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

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

[13] 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*, pages 70–77, Annapolis, MD, October 1996.

[14] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *J. Parallel Distrib. Comput.*, 5(2):172–184, April 1988.

[15] F. T. Luk and H. Park. Fault-tolerant matrix triangularizations on systolic arrays. *IEEE Trans. Comput.*, 37(11):1434–1438, November 1988.

[16] 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*, 5(3):329–335, 2005.

[17] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, October 1998.

[18] G. Quintana-Ortí and R. v. d. Geijn. Improving the performance of reduction to hessenberg form. *ACM Trans. Math. Softw.*, 32(2):180–194, June 2006.

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

[20] 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*, ICS '11, pages 152–161, New York, NY, USA, 2011. ACM.

[21] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 69–78, New York, NY, USA, 2012. ACM.

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

[23] S. Tomov and J. Dongarra. Accelerating the reduction to upper hessenberg form through hybrid gpu-based computing. Technical Report UT-CS-09-642, University of Tennessee Knoxville, 2009.

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

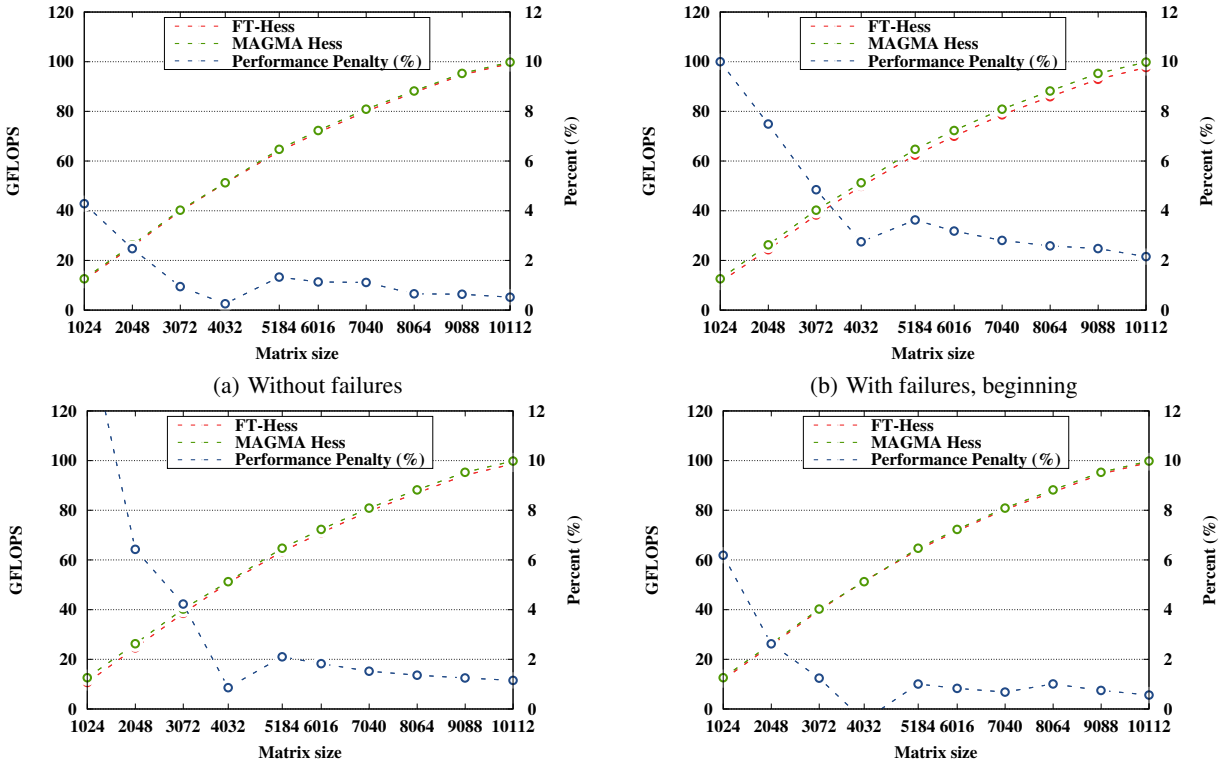
[25] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

APPENDIX

A. SUPPLEMENT

Algorithm 4 Fault Tolerant Hybrid Hessenberg Reduction 2

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



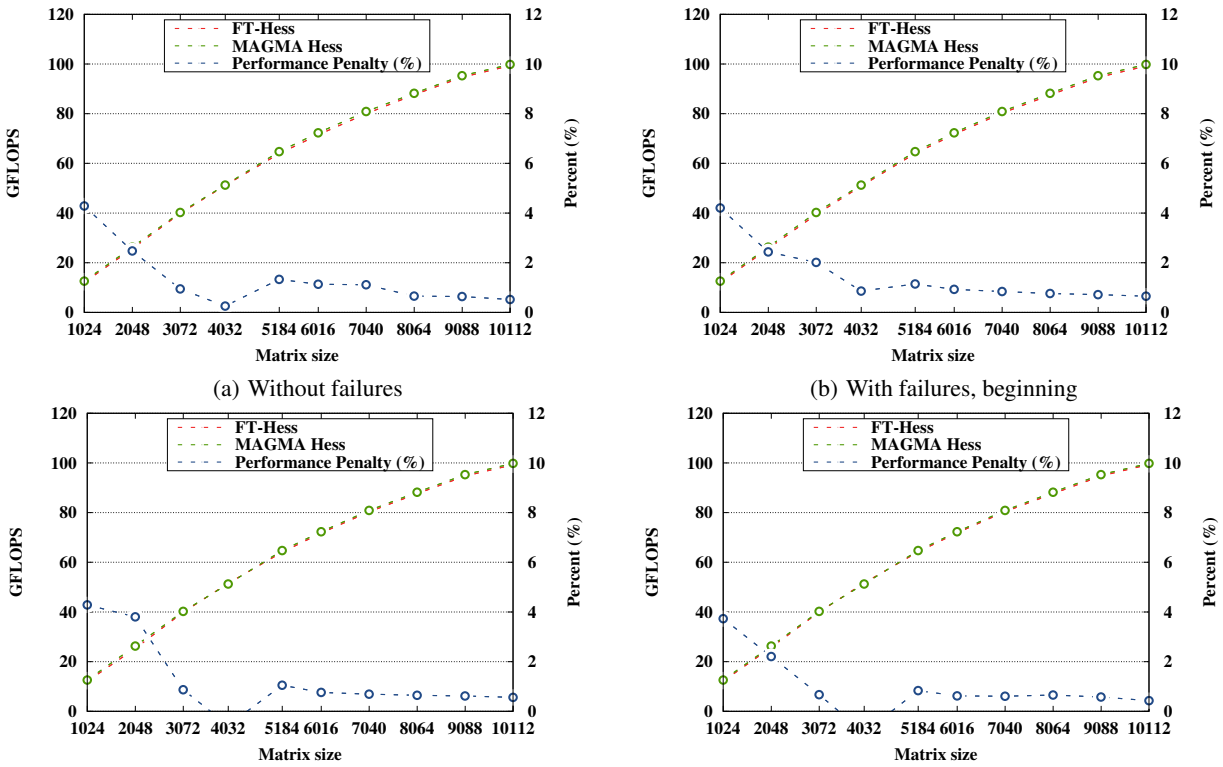
(a) Without failures

(b) With failures, beginning

(c) With failures, middle

(d) With failures, end

Figure 7: Overhead of FT-Hess without failures and with one failure. A2



(a) Without failures

(b) With failures, beginning

(c) With failures, middle

(d) With failures, end

Figure 8: Overhead of FT-Hess without failures and with one failure. A3