Soft Error Resilient QR Factorization for Hybrid System with GPGPU

Peng Du*, Piotr Luszczek*, Stan Tomov*, Jack Dongarra†
* EECS, University of Tennessee; 1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA
Email: {du, luszczek, tomov}@eecs.utk.edu
† University of Tennessee
Oak Ridge National Laboratory, Oak Ridge, TN, USA; University of Manchester, Manchester, UK
Email: dongarra@eecs.utk.edu

***Abstract*—The general purpose graphics processing units (GPGPU) are increasingly deployed for scientific computing due to their performance advantages over CPUs. What followed is the fact that fault tolerance has become a more serious concern compared to the period when GPGPUs were used exclusively for graphics applications. Using GPUs and CPUs together in a hybrid computing system increases flexibility and performance but also increases the possibility of the computations being affected by soft errors, for example, in the form of bit flips. In this work, we propose a soft error resilient algorithm for QR factorization on such hybrid systems. Our contributions include (1) a checkpointing and recovery mechanism for the left-factor Q whose performance is scalable on hybrid systems; (2) optimized Givens rotation utilities on GPGPUs to efficiently reduce an upper Hessenberg matrix to an upper triangular form for the protection of the right factor R, and (3) a recovery algorithm based on QR update on GPGPUs. Experimental results show that our fault tolerant QR factorization can successfully detect and recover from soft errors in the entire matrix with little overhead on hybrid systems with GPGPUs.**

I. Introduction

Since the introduction of general-purpose computing on graphics processing units (GPGPU), GPUs have quickly become the backbone of the modern high performance computing systems. For instance, China's Tianhe-1A that ranked number one on the November 2010 TOP500 list [27] uses 7,168 NVIDIA Tesla M2050 GPGPUs to achieve 2.57 Pflop/s in the High-Performance LINPACK (HPL) benchmark. While GPUs provide extremely high floating-point processing power, once combined with conventional multi-core CPUs into hybrid systems, performance is further increased for scientific applications [3] by executing the tasks with less parallelism on CPUs, concurrently with tasks that have high parallelism on the GPUs.

As the deployment of the GPGPUs grows rapidly, fault tolerance that has traditionally been relegated to only CPU-based computing systems [32], [16] started to emerge on GPU-based platforms. Traditionally, fault tolerance has been ignored in systems utilizing GPUs because they were originally developed mainly for graphics applications, such as 3D games which favor performance over reliability with bit-wise accuracy. As a result, transient errors, such as soft errors in the form of bit flips caused by cosmic radiation [19], may be tolerated in a vast majority of rendering situations. As technology brings the GPUs into the scientific computing arena, soft errors during computing are no longer tolerable since bit flips affect the result of floating point operation, and, to worsen the situation, in hybrid systems soft errors could propagate between the CPUs and the GPUs corrupting large matrix areas with errors in the factorizations result. Unlike fail-stop failure which brings down the whole system and halts the application execution, soft errors occur silently and cause a "silent data corruption". These errors leave no trace in system logs, and the consequences include incorrect application results, unpredictable code paths taken as a result of errors, and propagation of the initial failure which, all together, waste valuable computing time and resources and make the error detection and recovery a very daunting task.

Since the introduction of NVIDIA's Fermi architecture [28], Error Correcting Code (ECC) has been integrated to protect from soft errors in the global memory of GPUs. While ECC adds overhead to communication I/O and reduces overall computing performance especially for the memory-bound operations, soft errors could still affect other parts of the system such as the caches and arithmetic logic circuits on GPUs. In addition, the CPU host remains as target of soft error as well.

In this work, we set out to provide fault tolerance through soft error resilience to the algorithms featured in our Matrix Algebra on the GPU and Multicore Architect (MAGMA) project [36]. As a demonstration, a single-GPU hybrid QR factorization is chosen to evaluate the capability and performance of the soft error detection and recovery algorithm. Future work will extend the algorithm presented here to multiple-GPU platforms. The rest of this paper is organized as follows: Section II gives a list of the related work items in the field of soft error protection on the GPGPU platforms. Section III introduces the target QR factorization and its implementation in MAGMA. Section IV models soft error in the QR factorization, and Section V details the recovery algorithm including the optimization of primitives for Givens rotations on the GPU. Section VI proposes a scalable double-error protection algorithm for the left factor Q by tracing the MAGMA QR. Section VII shows experimental results that evaluate various aspects of our fault tolerant algorithm and, finally, Section VIII concludes the work and outlines possible future directions.

II. Related Work

For parallel applications, checkpoint-restart (C/R) has been the most commonly used method for fault tolerance [1], where the running state of the application is dumped to reliable storage at certain intervals, either by the message passing middleware automatically or at an explicit request of the user application. C/R requires the least user intervention, but suffers from high checkpointing overhead when writing data to the stable storage.

To reduce overhead, diskless checkpointing [30] is proposed to replace disk storage with system memory for checksum storage. Both C/R and diskless checkpointing need the error information for recovery. Unfortunately, no such information is available for soft errors. In order to detect errors without frequent checks, algorithm based fault tolerance (ABFT) was proposed to remove periodical checkpointing and only perform error checks when the execution being protected is finished [20], [2]. This eliminates checkpointing overhead, and the checksums during computing could reflect the most current status of the data which harbors clues for soft error detection and recovery. ABFT was originally introduced to deal with silent errors in systolic arrays. Matrix data was encoded once before the computation began. Matrix algorithms are carried out along with the encoded checksum in addition to the original matrix data, and the correctness is checked after the matrix operation completes.

ABFT for matrix factorization was explored in the 1980s for a single soft error [22], [23]. It was later extended to multiple errors [29], [15], [5] by adopting methodology from error correcting codes. These methods for systolic arrays offer promising direction, but require modification in both algorithm and implementation, especially when dealing with hybrid systems and applications with GPGPUs, where soft errors could occur on either the host (the CPU and the main memory) or on the GPU. Soft errors on the GPU have been exploited [18], and methods have been developed to detect [34], [37] and recover from such errors [33], [25], [24]. Recently, soft errors in matrix multiplication on a GPUs have also been studied [11].

In the realm of fault tolerant QR, Givens rotations based QR factorization has been studied [26]. However, since Householder QR is widely used in most modern math libraries, in our work we consider a right-looking Householder based QR factorization for a hybrid CPU/GPU system. Our method is based on the error model by Luk et al. [23]. We extended this model by adding protection to the left factor Q. We also provide an optimized recovery algorithm on the GPU.

III. Hybrid QR

In dense linear algebra, the QR factorization decomposes a matrix A into a product $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. QR factorization is often used to solve the linear least squares problem, and is also a central component of the QR iteration method for an eigenvalue problem algorithm.

Several methods exist for computing the QR factorization, such as the Gram-Schmidt process, Householder transformations, and Givens rotations. In today's high performance math libraries, for instance, LAPACK [4], ScaLAPACK [9], and MAGMA, Householder transformations are used to achieve high performance by optimizing the use of the memory hierarchy in modern systems. Given an input matrix A, a Householder matrix Q_1 is used to multiply A:

$$Q_1 A = \begin{bmatrix} r_{11} & r_{12} \cdots r_{1n} \\ 0 & \\ \vdots & A' \\ 0 & \end{bmatrix}$$

This operation annihilates the elements below the diagonal in the first column. The next step is carried out on the trailing matrix A' by a new Householder matrix:

$$Q_2{}' = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & Q_2 & \\ 0 & & & \end{bmatrix}$$

MAGMA uses a block version of the QR factorization by accumulating a few steps of the Householder annihilations. This method is rich in Level 3 BLAS operations and therefore can achieve high performance. Q is stored below the lower diagonal of the input matrix in the form of the WY representation of Householder transformation products [31], [8].

Implementation-wise, the algorithm used by MAGMA is close to the LAPACK QR, except the MAGMA QR is designed and optimized for heterogeneous architectures that consist of a CPU and a GPU. The hybrid QR that we consider, has the input matrix and the result in the GPU's global memory. The computational pattern is similar to the LAPACK's QR – a sequence of a panel factorization followed by a corresponding trailing matrix update. The current panel to be factored is sent to the CPU and factored using LAPACK. The result is copied back to the GPU memory and used on the GPU for the trailing matrix update. The update is split into two – first is an update for the columns that will form the "next" panel, followed by the update for the rest of the trailing matrix. This splitting, known as the *lookahead* technique, is done so that the factorization of the next panel can start before finishing the entire update for which the next panel is part of. This allows overlapping the large update of the trailing matrix and sending the panel to the CPU, its factorization and copy back to the GPU. As a result, for large enough matrices, the overall performance of the algorithm is dictated by the performance of the matrix-matrix multiplications on the GPU. Note that communication is minimized (and overlapped with computation) as on each step the algorithm communicates a panel of size $O(NB \times N)$ and performs operations of size $O(NB \times N^2)$. For further detail on the implementation, one can see the sources available through the MAGMA site.

IV. Soft Error Modeling

MAGMA QR runs with both the GPU and CPU, therefore soft errors on both platforms are considered as a source of contamination. Also, since the result of panel factorization and trailing panel of lookahead commute between the CPU and GPU frequently, soft errors could propagate between the GPU and CPU at any step of the factorization, making error detection a challenging task. To simplify the error analysis and avoid the issue of timing of errors, we adopt an error modeling technique [23]. With this model, the soft error in the right factor can be located by the column of its occurance, and in later section, the correct result of Q and R is rebuilt based on the column number.

A. Error Model

Luk et al. [23] derived a model for both LU and QR using the "ZU" notation where Z represents the left factor and U represents the upper triangular right factor. We return to the "QR" notation for clarity.

Having the initial matrix, $A_0 = A$, the Householder QR is carried out by introducing Householder transforms from the left to get the final triangular form. Let $A_t = Q_{t-1}A_{t-1}$, where Q_{t-1} is the Householder transform matrix at step $t-1$. At step $t-1$, error occurs at random location (i, j) in matrix A as

$$\begin{aligned} \tilde{A}_t &= Q_{t-1}A_{t-1} - \lambda e_i e_j^T \\ &= Q_{t-1}(Q_{t-2}\dots Q_0)A_0 - \lambda e_i e_j^T \end{aligned} \tag{1}$$

e_i, e_j are column vectors with all 0 elements except 1 as the i_{th} and j_{th} row, respectively. And λ is the magnitude of the soft error. The factorization continues in spite of the occurrence of a soft error from step t till the end. View the soft error at step t as the result of perturbation to an erroneous initial matrix:

$$\tilde{A} = A - de_j^T \tag{2}$$

where $d = \lambda(Q_{t-1}\dots Q_0)^{-1}e_i$, then the erroneous process of QR factorization equals to an error-free QR factorization from an erroneous initial matrix $\tilde{A}$. This model is similar to the round-off error analysis in perturbation theory [10].

B. Checksum for R

In MAGMA, the right-looking Householder QR follows LAPACK QR storage, where the right factor R overwrites the upper triangular part of the input matrix, including the diagonals, while the lower triangular part is replaced by Q in the form of vectors that defines elementary reflectors.

During QR factorization, once a panel of Q is produced, its values do not change till the end. It has been shown that Q cannot be protected by appending rows of checksum at the bottom of the input matrix and having QR factorization along with the checksum rows [13]. Since most of the run time of QR factorization is spent in the trailing matrix update, we assume no errors during the process of panel factorization. And we propose to protect the result of panel factorization with diskless checkpointing.

The part of the matrix other than Q is divided into two regions, the already formed R and the trailing matrix A',

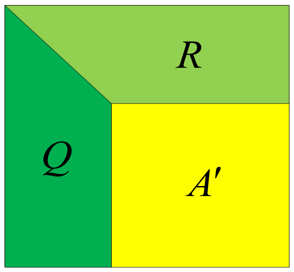

Fig. 1. Different regions of A during factorization

as shown in Figure 1. Each iteration of the trailing update moves a few rows from A' to R, and therefore both A' and R undergo constant changes during the factorization, and cannot be protected by diskless checkpointing without causing large overhead. For R, we adopt the ABFT technique from [2], [21], which was also used in Luk's work [22], [23] to fight soft error in systolic arrays.

To capture one error, for input matrix $A \in \mathbb{R}^{m\times n}$, two generator matrices are used, $e = (1, 1, \dots, 1)^T$ and a random column vector w. $e, w \in \mathbb{R}^{m\times 1}$.

Before factorization, two columns of checksum $(Ae\ Aw)$ are generated and appended on the right of the input matrix as $A_c = (A\ Ae\ Aw)$. Then QR factorization is applied to A_c:

$$(A\ Ae\ Aw) = Q(R\ c\ v) \tag{3}$$

$c, v \in \mathbb{R}^{m\times 1}$ are checksum columns after factorization. Use the error model in (2), the factorization of A with soft error is treated as a soft-error-free QR factorization from $\tilde{A}$:

$$(\tilde{A}\ Ae\ Aw) = \tilde{Q}(\tilde{R}\ \tilde{c}\ \tilde{v}) \tag{4}$$

Therefore,

$$\begin{aligned} \tilde{c} &= \tilde{Q}^{-1}Ae = \tilde{Q}^{-1}(\tilde{A} + de_j^T)e \\ &= \tilde{Q}^{-1}(\tilde{Q}\tilde{R} + de_j^T)e \\ &= \tilde{R}e + \tilde{Q}^{-1}de_j^T e = \tilde{R}e + \tilde{Q}^{-1}d \end{aligned}$$

By the same token,

$$\tilde{v} = \tilde{R}w + w_j\tilde{Q}^{-1}d$$

Hence,

$$\begin{cases} \tilde{r} = \tilde{c} - \tilde{R}e = \tilde{Q}^{-1}d \\ \tilde{s} = \tilde{v} - \tilde{R}w = w_j\tilde{Q}^{-1}d \end{cases} \tag{5}$$

Here $r, s \in \mathbb{R}^{m\times 1}$ are residual vectors. Compare the equation of $\tilde{r}$ and $\tilde{s}$, we have:

$$\tilde{s} = w_j\tilde{r}. \tag{6}$$

w_j is the j_{th} element in the generator vector w, and j is the column where the soft error firstly strikes. In practice, $\tilde{r}$ is computed first to check for the occurrence of soft error. If error is detected, (6) is then used to locate the error column. This process applies to soft error in R and A'.

V. Recovery Algorithm

With the knowledge of error column j, Luk et al. [23] suggested a few possible methods to recover the left and right factors without details and implementation. In this section we complete this work with an algorithm based on QR update to accommodate the storage format of MAGMA QR.

A. Spike-Eliminating Technique

Using the QR notation, the spike reducing technique in [23] starts with the difference of the true initial matrix A and the erroneous initial matrix $\tilde{A}$, obtained in (2).

$$\begin{aligned}
A-\tilde{A} &= (a_{\cdot j}-\tilde{Q}\tilde{R}_{\cdot j})e_j^T \\
A &= \tilde{Q}\tilde{R}+(a_{\cdot j}-\tilde{Q}\tilde{R}_{\cdot j})e_j^T=\tilde{Q}\tilde{R}+\tilde{Q}(\tilde{Q}^T a_{\cdot j}-\tilde{R}_{\cdot j})e_j^T \\
A &= \tilde{Q}(\tilde{R}+pe_j^T) \\
A &= \tilde{Q}\tilde{C},\ C=\tilde{R}+pe_j^T, p=\tilde{Q}^T a_{\cdot j}-\tilde{R}_{\cdot j} \qquad (7)
\end{aligned}$$

C in (7) is an upper triangular matrix with a spike in column j. Since QR requires Q to be an orthogonal matrix, orthogonal transformations are needed to remove non-zeros in the spike.

There are a few choices of algorithm such as Householder transformation and Givens rotation. Householder is more computing intensive and has higher parallelism which is more suitable for the GPU, but it also requires higher amount of extra memory because, while the first Householder transformation removes the spike in column j, the triangular submatrix $(j+1:end, j+1:end)$ becomes a full matrix, and if j is small, this requires an extra buffer almost as large as the data matrix A and since in MAGMA QR the lower triangular is used to store Q, data matrix space cannot be borrowed. Given that the global memory on the GPU is normally used to the limit for matrix data, Householder transformation does not qualify for this high memory demand and therefore we choose Givens rotation as the spike elimination algorithm. Since Givens rotation is memory-bound, implementation on the GPU requires careful design for the best performance. This will be covered in section V-C.

B. QR Update as the Recovery Algorithm

In (2), obtaining the QR factorization of A from $\tilde{A}$ clearly marks it as a QR update problem. QR update is also widely used in applications where repeated updating is required [35], and at the time of this work the available Given rotations implementation for GPU is still primitive and lacks optimization, therefore we devised a general optimization method for the QR update algorithm on GPU and applied to the soft error recovery problem at hand.

The rank-1 update to QR factorization has been described in [17]. We show the algorithm in the context of QR recovery.

Given the erroneous initial matrix and its QR factorization $\tilde{A}=\tilde{Q}\tilde{R}$, the objective is to find the QR factorization of the true initial matrix $A=QR$. Let $u=a_{\cdot j}-\tilde{Q}\tilde{R}_{\cdot j}$, and $v=e_j$,

$$\begin{aligned}
A &= \tilde{A}+uv^T \\
&= \tilde{Q}\tilde{R}+uv^T=\tilde{Q}(\tilde{R}+\tilde{Q}^T uv^T) \\
\therefore\ A &= \tilde{Q}(\tilde{R}+wv^T),\ w=\tilde{Q}^T u=\tilde{Q}^T a_{\cdot j}-\tilde{R}_{\cdot j}
\end{aligned}$$

First, a series of Givens rotations $J^T=J_1^T\cdots J_{n-1}^T$ is used such that $J^T\times w=\pm\|w\|_2 e_1$. The sequence $1,\cdots,n-1$ applied from left to w means the elimination is from bottom up. Since $H=J^T\times\tilde{R}$ is an upper Hessenberg matrix,

$$J^T\times(\tilde{R}+wv^T)=H\pm\|w\|_2 e_1 v^T=\hat{H}$$

is also upper Hessenberg.

To get R from $\hat{H}$, another series of Givens rotations $G^T=G_{n-1}^T\cdots G_1^T$ is used such that $G^T\times\hat{H}=R$. The sequence $n-1,\cdots,1$ means the elimination is from top down.

Combining J and G,

$$Q=\tilde{Q}JG=\tilde{Q}(J_{n-1}\cdots J_1)(G_1\cdots G_{n-1})$$

Algorithm 1 describes the above recovery procedure.

Algorithm 1 QR Recovery Algorithm based on QR-update

```
Require: $\tilde{A}$, $\tilde{Q}$, and $\tilde{R}$
  Obtain $a_{\cdot j}$ and $\tilde{R}_{\cdot j}$
  Calculate $w=\tilde{Q}^T u=\tilde{Q}^T a_{\cdot j}-\tilde{R}_{\cdot j}$
  Zero out $w$ using Givens Rotations as $k_1=J^T\times w=\pm\|w\|_2 e_1$
  Apply $J^T$ to $\tilde{R}$ as $k_2=J^T\tilde{R}$, and store the subdiagonals of $k_2$ into extra storage $Y$
  Perform $\hat{H}=k_2+k_1 e_j^T$
  Zero out subdiagonals of $\hat{H}$ by Givens rotations $G^T\times\hat{H}=R$
```

Along with Algorithm 1, there are some implementation details worth noticing. First, the column j of the original matrix A is required for recovery. For scientific applications that expect soft error with high probability, a mechanism to recover some part of the original matrix is required. Some applications can generate any column of A easily, others need to store the whole matrix A. In our implementation, at the beginning of QR factorization, matrix A on the GPU memory is asynchronously copied to the CPU memory during the first panel factorization for this purpose.

Second, recovery can be performed using the GPU in-situ or on the CPU host with two data transfers, one to load data from the GPU to the CPU and one to store result back. This solution is easier in implementation since LAPACK is equipped with Givens rotation utilities like DLARTG and DLASR, but it suffers from performance impact of data transfer and much lower parallelism of the CPU compared to the GPU. Therefore, we choose to perform the QR recovery on the GPU in place with the matrix data. Since R can only overwrite the upper triangular of A, subdiagonals of k_2 and $\hat{H}$ are kept in a separate 1D buffer Y.

C. Givens Rotation Utilities for the GPU

Givens rotation is at the center of the recovery procedure. Two operations involved are DROTG and DLASR in the LAPACK term. While these operations are readily available for the CPU, on the GPU they pose a significant challenge to be implemented with good performance. We first discuss the two major challenges and then provide our solution.

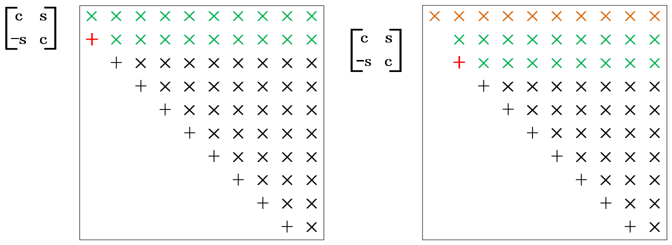
Fig. 2. Reduction from upper Hessenberg to upper triangular

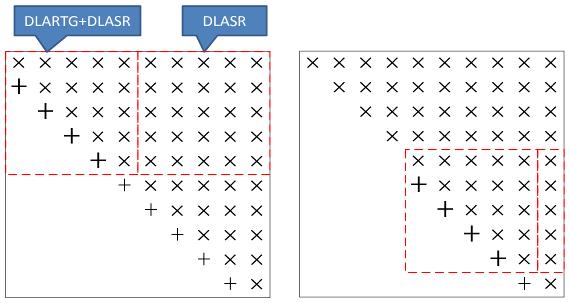

Fig. 3. Reduction from upper Hessenberg to upper triangular (block algorithm)

1) Memory Access Pattern: DROTG generates a plane rotation such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

In this work we use an improved version of DROTG called DLARTG, which is more numerically reliable [7].

DLASR applies a set of plane rotations to a matrix in a certain order, for example one set of plane rotation is applied to a $2 \times N$ matrix,

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1N} \\ x_{21} & \cdots & x_{2N} \end{bmatrix} = \begin{bmatrix} y_{11} & \cdots & y_{1N} \\ y_{21} & \cdots & y_{2N} \end{bmatrix} \quad (8)$$

The FLOP count is $12N$ and the memory operation is $4N+4$, making it a memory-bound operation. While each column of the right hand side $\begin{bmatrix} y_{1j}, & y_{2j} \end{bmatrix}^T$ can be fully parallelized, without data reuse, on the GPU the performance of DLASR is still limited by the memory bandwidth between the GPU global memory and the registers. To worsen the situation, since MAGMA QR uses column-major storage, if each thread calculated one column of the right hand side, the fetching of $[x_{i1}, \cdots, x_{iN}]$ and $[y_{i1}, \cdots, y_{iN}], i = 1,2$ by each thread does not fit the condition of global memory coalescing on the GPU, and each column has to be accessed one at a time.

2) Data Caching: In Algorithm 1, DLARTG and DROTG are used together to firstly create the upper Hessenberg matrix H, and then reduce it to upper triangular. Common in the fused operation are the following Two operation steps:

1) Generate a plane rotation $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ using DLARTG for a vector $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$
2) Apply $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ to a $2 \times N$ matrix as in (8), as in DLASR

Both of these steps are carried out on the GPU. These two steps are consecutive. Figure 2 is an example in the last step of Algorithm 1. The plus signs on the subdiagonal are those elements to be zeroed out, and the red plus signs are the values being eliminated in the current step. Green and red are the elements that participate in the current step. This operation sweeps from top to bottom until an upper triangular matrix is produced.

Take the first two steps for example, the second row of the matrix is updated by the DLASR in the first step and then used as input for the second step. To reduce global memory access that is far more expensive than that of registers and shared memory on the GPU, this row should be cached for the next step rather than read from global memory after being just written there. We use one thread to handle each column of H, and given the size of H, more than one thread blocks is needed for each step. In addition, one thread blocks (one thread per se) performs the DLRTG before all the DLARTG thread blocks could start, hence a synchronization is needed to hold DLASR threads while waiting for the one thread that does DLARTG to finish. To achieve the aforementioned caching using registers, both DLARTG and DLASR functionalities need to reside in one GPU kernel, otherwise the DLASR kernel calls are separated from each other by DLARTG kernel calls, and caching can only be done through shared memory, which is less efficient. The dilemma here is that CUDA offers no lightweight mechanism to synchronize all thread blocks from within threads. Available synchronization mechanisms include global synchronization initiated by host, and synchronization of all threads within a thread block. The atomic operation provides some possibilities but threads that participate in an atomic operation through a variable in global memory are serialized, and therefore suffers a large performance penalty.

3) Algorithm for fused DLARTG and DLASR operation: For dense linear algebra, blocked algorithms have been widely used to achieve high performance on modern computer systems with complex cache hierarchy [12]. To bridge the requirement of caching intermediate rows to reduce global memory access and the difficulty of no lightweight synchronization from within threads, we devised the following algorithm for the fused DLARTG and DLASR operation by having each step work with a block of data rather than only two rows.

Two types of kernels are designed. The first kernel generates a set of plane rotations and use these rotations to reduce an $NB \times NB$ upper Hessenberg submatrix on the diagonal to upper triangular. NB is selected as the maximum number of threads per thread block allowed by the GPU except for edge cases. In our experiment, with a Tesla T20 ('Fermi'), $NB = 1024$.

The second kernel applies this set of plane rotations to all the data on the right of the diagonal $NB \times NB$. Global synchronization on the host is used between these two kernels.

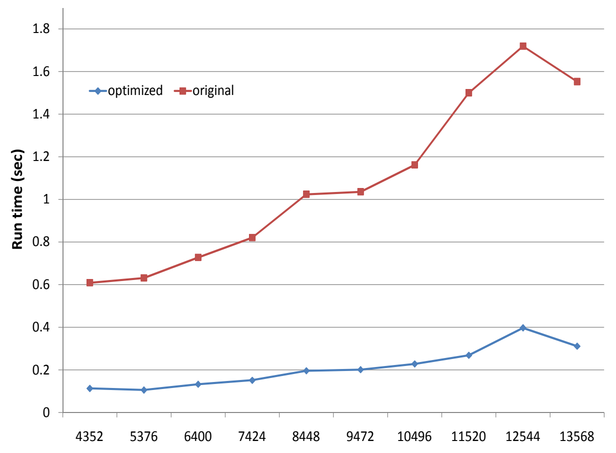

Fig. 4. Performance comparison of the original and optimized DLASR kernel

This algorithm moves down along the diagonal with a step size of NB until an upper triangular matrix is produced. Figure 3 is an example of this algorithm with $NB = 5$. During each iteration, only one thread block is spawned for the first type of kernel and as many thread blocks as needed are spawned for the second kernel.

Within the first kernel, steps proceed as in the unblocked version of fused DLARTG and DLASR. Intermediate rows that are produced by step $i-1$ and will be used in step i are cached in registers to avoid loading from global memory. Thread-block level synchronization is used to separate DLARTG and DLASR. Within the second kernel, steps proceed from the top down, one row each step. Similarly, intermediate rows are cached in registers. The plane rotations are stored in two vectors, respectively, in global memory to pass between the two kernels. In the second kernel, the fetching of current plane rotation pair c and s that is on the critical path of execution is moved to the beginning of kernel execution where NB threads are used to fetch NB plane rotation pairs in a coalesced fashion.

4) Improvement Experiment: Figure 4 is an experiment result of the run time for the reduction of H from upper Hessenberg to upper triangular. The matrix size derives from actual recovery experiment in section VII where the impact of the new reduction algorithm on recovery performance is shown in Figure 6. By using a more efficient memory access pattern and the blocked algorithm, 5x speedup is achieved in the fused DLARTG and DLASR operation.

VI. Protection for Q

This section describes the protection to the left factor Q. Other than the importance of obtaining the correct factorization result, recall that the spike-eliminating algorithm in Algorithm 1 also functions under the assumption that no soft error strikes $\tilde{Q}$, which is the erroneous Q caused ONLY by soft error in R or A'. In MAGMA QR, since Q occupies half of the matrix, it is as susceptible to soft error as other section of the matrix and therefore need to be protected.

A. Checkpointing for Q

In order to provide soft error resilience to Q, we propose to use diskless checkpointing because once a panel is factorized on the CPU, the result is not subject to any further change.

For any column of the factorized panel v, the objective of the checkpointing scheme is to allow recovery from errors that occur to random items in the column. It has been shown in [14] that one soft error in a column of L in LU can be protected with trivial overhead. Here we implement a double soft error correction checkpointing for Q.

Suppose $v = [v_1, v_2, \cdots, v_k]^T$, the vertical checkpointing produces the following three checksums:

$$\begin{cases} v_1 + v_2 + \cdots + v_k = c_1 \\ w_1 v_1 + w_2 v_2 + \cdots + w_k v_k = c_2 \\ u_1 v_1 + u_2 v_2 + \cdots + u_k v_k = c_3 \end{cases} \tag{9}$$

Let $u_i = w_i^2,\ i = 1 \cdots k$, (9) yields the check equation:

$$(\tilde{c_3} - c_3) - (w_i + w_j)(\tilde{c_2} - c_2) + w_i w_j (\tilde{c_1} - c_1) = 0 \tag{10}$$

w_i, w_j can be determined by iterating through all possible combinations. The complexity is $O(n^2)$ because, assuming $i < j$, for each i, up to $n-i$ pairs of $w_i\ w_j$ are tested in (10) until a pair is found that fits the check equation. With w_i and w_j, the errors at row i and j of v can be corrected.

B. Timing of Checkpointing

In order to protect Q at the earliest time, checkpointing for Q is performed right after every panel factorization. Since panel factorization is on the critical path of execution, it is important to place the checkpointing at a time which does not cause serious performance degradation.

As described in section III, MAGMA QR produces Q using the CPU implementation DGEQRF and during step i, an $N_i \times NB$ block of the trailing matrix is sent from the GPU to the CPU memory to be factorized by DGEQRF. Then the triangular factor T of a real block reflector H is constructed by DLARFT on the CPU and both the panel factorization and T are sent to the GPU to update the trailing matrix using a GPU version DLARFB. This process is illustrated by the trace of an actual MAGMA QR run on a 48-core CPU + NVIDIA T20 GPU machine shown in Figure 5 generated by TAU (Tuning and Analysis Utilities) [6]. The size of this run is 17408×17408, and only the first few iterations are shown.

The best way to avoid excessive performance penalty is to hide the checkpointing into a time slot when one of CPU and GPU waits for another to finish and therefore exposes a block of idle time. Even though the DLARFB on the GPU takes a long time to finish, by using lookahead it keeps the CPU busy most of the time, leaving very little room for extra operation. Closely examining the tracing, we notice that the yellow section that represents cublasSetMatrix(), which sends panel factorization result from the CPU to the GPU, actually takes longer than the actual communication, and the reason is that cublasSetMatrix() is a blocking call on the GPU and it does not start the data transfer until all activities on the GPU

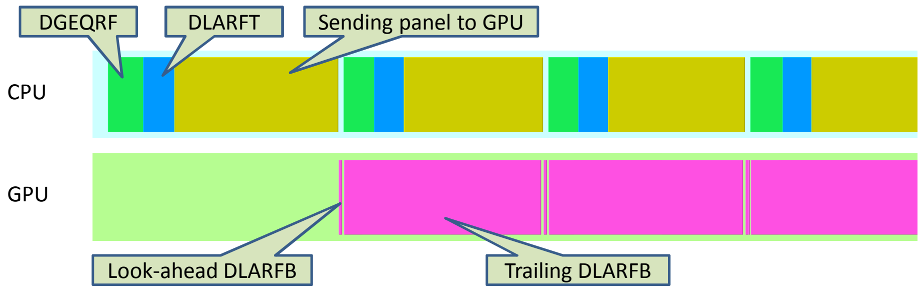

Fig. 5. MAGMA QR tracing

started previously are finished. From Figure 5, clearly cublasSetMatrix() is always called on the CPU during the trailing matrix update (DLARFB) on the GPU and this accordingly not only blocks both the data transferring to the GPU, but also puts the CPU in a busy wait and therefore cannot perform other tasks. This does not affect the performance of MAGMA QR since MAGMA QR uses 1-depth lookahead and the next trailing matrix update cannot start anyway without the previous one finished.

To release the CPU from the busy wait, cublasSetMatrix() is replaced with an asynchronous data transferring function cudaMemcpy2DAsync(). This function initiates the data transferring and returns control immediately to the CPU. The time gap between this initiation time and when the GPU DLARFB is finished is large enough to hide the checkpointing Q from the critical path. As the trailing matrix becomes smaller, there is a certain threshold of time when the GPU DLARFB finishes before the initiation of cublasSetMatrix(), and this could expose the checkpointing and cause performance impact, but this only accounts for a small portion of the execution. Such a situation can be further improved by moving the checkpointing to the GPU between the time GPU DLARFB finishes and the initiation of cublasSetMatrix on the CPU.

VII. Performance Evaluation

In this section we evaluate the performance of the fault tolerant QR algorithm on two hybrid systems. CUDA 4.0 is used in both experiments with MAGMA 1.0.

The first system is equipped with NVIDIA C2050 with a 48-core AMD Opteron CPU (48 threads for MKL). Figure 6 is the performance of recovery from error initially in R or A'. For all matrix sizes, error is injected to a randomly picked data at (7681,7682) in A' on the GPU right before the 31st step of panel factorization. The purple line is the performance of FT-QR with checkpointing Q and no error. Two recovery performances are shown. The green line is the plain (unoptimized) implementation of Givens rotation utilities on the GPU. This implementation is limited by the GPU global memory access speed without the help of coalescing and shared memory. The red line is the optimized recovery performance where a blocked and fused DLARTG and DLASR with better memory access mechanism is in place. At the largest problem size available to this GPU, the optimization improves 5% of the recovery performance. The recovery from one soft error in A', using the optimized algorithm, reduces 15% of the overall performance of the MAGMA QR.

The NVIDIA C2050 has relative small on-chip global memory which limits the size of matrix in the first experiment. The second testing platform is the Keeneland Initial Delivery system which features a cluster of NVIDIA M2070 with 6GB memory, and each host runs two Intel Westmere hex-core CPUs. Figure 7 is the performance of both the original and soft error resilient MAGMA QR on a single node of Keeneland. Error recovery experiments use the same setup as in the test on C2050, and as a comparison, MKL QR performance is also shown running with 12 threads on a single node. The extended matrix size range shows consistent overhead to the result on C2050, verifying that with the small overhead of fault resilience functionalities, the hybrid QR still outperforms multi-threaded QR on the multicore CPUs by almost 100%, and errors can be recovered with little performance impact.

VIII. Conclusion

In this work we developed a soft error resilient QR algorithm for hybrid architecture equipped with CPU and GPU. Based on the MAGMA's QR implementation, our FT-QR algorithm can recover Q and R from soft errors that occur

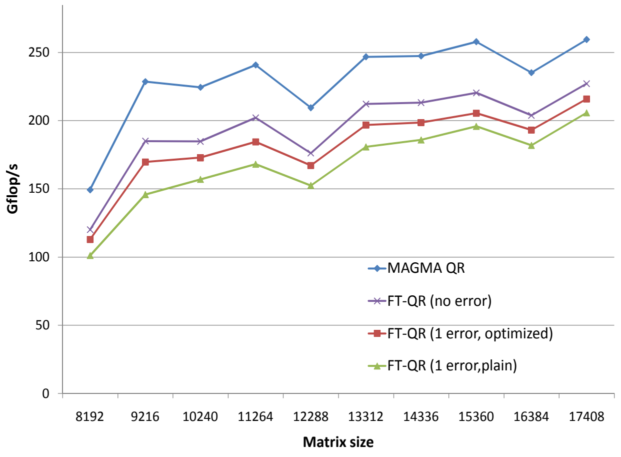

Fig. 6. Performance of recovery for error in R on NVIDIA C2050

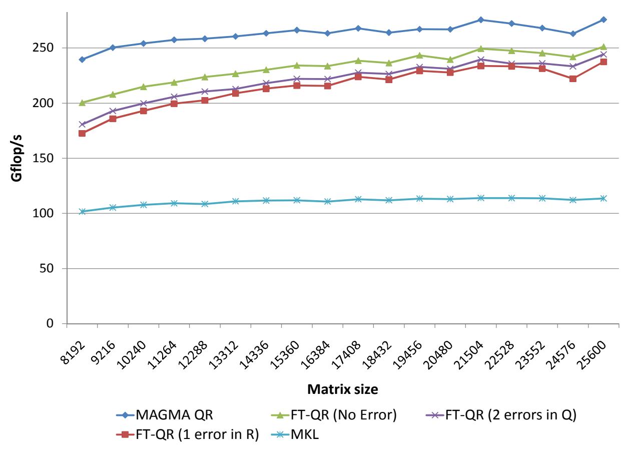

Fig. 7. Performance on NVIDIA C2070

during any step of the QR factorization in the whole matrix. In order to increase the performance of recovery, an optimized Givens rotation for GPU is designed. Experimental result shows that our fault tolerance functionalities impose small performance impact to the MAGMA QR. As future work, the proposed algorithm will be extended in general to dense linear factorizations like Cholesky, LU and QR on more complicated heterogeneous architectures such as multiple-GPU cluster.

References

[1] Fault tolerance for extreme-scale computing workshop report, 2009.

[2] J. Abraham. Fault tolerance techniques for highly parallel signal processing architectures. *Highly parallel signal processing architectures*, pages 49–65, 1986.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[4] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[5] C. Anfinson and F. Luk. A linear algebraic model of algorithm-based fault tolerance. *Computers, IEEE Transactions on*, 37(12):1599–1604, 1988.

[6] A. Biersdorff, W. Spear, and S. Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. 2010.

[7] D. Bindel, J. Demmel, W. Kahan, and O. Marques. On computing givens rotations reliably and efficiently. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):206–238, 2002.

[8] C. Bischof and C. Van Loan. The wy representation for products of householder matrices. In *Selected Papers from the Second Conference on Parallel Processing for Scientific Computing*, pages 2–13. Society for Industrial and Applied Mathematics, 1985.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.

[10] J. Demmel et al. *Applied numerical linear algebra*, volume 150. Society for Industrial and Applied Mathematics Philadelphia, PA,, USA, 1997.

[11] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen. Matrix multiplication on gpus with on-line fault tolerance. In *Proceedings of the 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2011)i*. IEEE Computer Society Press, 2011.

[12] J. Dongarra and D. Walker. The design of linear algebra libraries for high performance computers. 1993.

[13] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. Technical Report 253, LAPACK Working Note, Aug. 2011.

[14] P. Du, P. Luszczek, and J. Dongarra. High performance dense linear system solver with soft error resilience. In *Proceedings of the IEEE Cluster 2011*. IEEE Computer Society Press, 2011.

[15] P. Fitzpatrick and C. Murphy. Fault tolerant matrix triangularization and solution of linear systems of equations. In *Application Specific Array Processors, 1992. Proceedings of the International Conference on*, pages 469–480. IEEE, 1992.

[16] G. Gibson. Failure tolerance in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022, 2007.

[17] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[18] I. Haque and V. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 691–696. IEEE Computer Society, 2010.

[19] T. Heijmen. Radiation-induced soft errors in digital circuits-a literature survey. 2002.

[20] K. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.

[21] K. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 2006.

[22] F. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques* 1. *Journal of Parallel and Distributed Computing*, 5(2):172–184, 1988.

[23] F. Luk and H. Park. Fault-tolerant matrix triangularizations on systolic arrays. *Computers, IEEE Transactions on*, 37(11):1434–1438, 1988.

[24] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity gpus. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE.

[25] N. Maruyama, A. Nukada, and S. Matsuoka. Software-based ecc for gpus. In *2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09)*, 2009.

[26] O. Maslennikow, J. Kaniewski, and R. Wyrzykowski. Fault tolerant qr-decomposition algorithm and its parallel implementation. In *Euro-Par'98 Parallel Processing*, pages 1–1. Springer, 1998.

[27] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon. *TOP500 Supercomputer Sites*, 36^{th} edition, November 2010. (The report can be downloaded from http://www.netlib.org/benchmark/top500.html).

[28] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi v1.1. Technical report, NVIDIA Corporation, 2009.

[29] H. Park. On multiple error detection in matrix triangularizations using checksum methods. *Journal of Parallel and Distributed Computing*, 14(1):90–97, 1992.

[30] J. Plank, K. Li, and M. Puening. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):972–986, 1998.

[31] R. Schreiber and C. Van Loan. A storage-efficient wy representation for products of householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[32] B. Schroeder and G. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[33] J. Sheaffer, D. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 55–64. Eurographics Association, 2007.

[34] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. On testing gpu memory for hard and soft errors. In *Proc. Symposium on Application Accelerators in High-Performance Computing*, 2009.

[35] G. Shroff and C. Bischof. Adaptive condition estimation for rank-one updates of qr factorizations. *SIAM journal on matrix analysis and applications*, 13:1264, 1992.

[36] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[37] K. Yim and R. Iyer. Hauberk: Lightweight silent data corruption error detectors for gpgpu. *In Proceedings of the 17th Humantech Thesis Prize (Also in IPDPS 2011)*, 2011.