Soft Error Resilient QR Factorization for Hybrid System

Peng Du
University of Tennessee, Knoxville
du@eecs.utk.edu

Piotr Luszczek
University of Tennessee, Knoxville
luszczek@eecs.utk.edu

Stanimire Tomov
University of Tennessee, Knoxville
tomov@eecs.utk.edu

Jack Dongarra
University of Tennessee, Knoxville
dongarra@eecs.utk.edu

ABSTRACT

As the general purpose graphics processing units (GPGPU) are increasingly deployed for scientific computing for its raw performance advantages compared to CPUs, the fault tolerance issue has started to become more of a concern than before when they were exclusively used for graphics applications. The pairing of GPUs with CPUs to form a hybrid computing systems for better flexibility and performance creates a massive amounts of computations that have a higher possibility to be affected by transient error – a soft error that silently modifies data causing errors to pass unnoticed. This is despite the fact that the newest Fermi generation of GPUs from NVIDIA are equipped with error correcting units to protect their memories. This problem is particularly serious for applications that employ numerical linear algebra since large sections of data are often modified between steps, and therefore even a single error could eventually propagate into a large area of result. In order to give protection to dense linear algebra computations on such hybrid systems, we developed an algorithm that is resilient to soft errors. We chose the right-looking Householder QR factorization as a demonstration of our algorithm for a hybrid system that features both GPUs and CPUs. Algorithm based fault tolerance (ABFT) is used to protect from errors in the trailing matrix and the right factor, while a checkpointing method is used to ensure the left factor is error-free. This work is based on a previous study of fault tolerance in matrix factorizations. Our contribution includes (1) a stable multiple-error checkpointing and recovery mechanism for the left-factor, which is also scalable in performance in the hybrid execution environment and does not cause severe performance degradation. (2) optimized Givens rotation utilities on the GPU to efficiently reduce an upper Hessenberg matrix to upper triangular form, and (3) a recovery algorithm based on QR update inside a hybrid system. Experimental results show that, our fault tolerant QR factorization can successfully detect and correct data altered by soft errors in both the left and right factors and we observe a decreasing percentage of overhead as the matrix size grows.

1. 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, when combined with a conventional multi-core CPU in a hybrid fashion, it has been shown to be capable of further boosting the performance of scientific applications [3] by executing tasks with less parallelism on CPUs, concurrently with tasks that have high parallelism on the GPUs.

As the deployment of the GPGPUs grows rapidly, the issue of fault tolerance that has been only affecting CPU-based computing systems [34, 15] starts to emerge on GPU-based platforms. Traditionally, fault tolerance had been ignored in systems utilizing the GPUs because they were originally developed mainly for graphics applications, such as 3D games which favor performance over reliability at bit-wise accuracy. Therefore, transient errors can be tolerated in a vast majority of rendering situations. As technology brings the GPUs into the scientific computing arena, transient errors during computing are no longer acceptable, and, to worsen the situation, in hybrid systems such errors could propagate between the CPUs and the GPUs making the hybrid systems even more fragile. Unlike fail-stop failure which brings down the whole system and halts the application execution, transient errors occur silently causing a "silent data corruption" due to various sources, mostly from cosmic radiation [19]. The errors leave no trace in system logs for system administrators to react at the time of failure. The consequences of the transient errors include incorrect application results, unpredictable code paths taken as a result of errors, and propagation of the initial failure which, all together, make the task of error detection and recovery so much more daunting.

Among the fault tolerance mechanisms, checkpointing and restart (C/R) is the most commonly used method. Straightforward in idea and implementation, C/R suffers large overhead from the frequent checkpointing which requires tedious and time consuming I/O operations. In the field of soft error, since no indication of error is made explicitly, there is a pressing need for more light-weight mechanism for fault tolerance.

In this work, we set out to provide fault tolerance and soft error resilience to the algorithms featured in our Matrix Algebra on the GPU and Multicore Architect (MAGMA) project [38]. As a demonstration, a single-GPU hybrid QR factorization is chosen. Future work will extend the algorithm in this work to multiple-GPU platforms. The rest of this paper is organized as the follows: Section 2 gives a lists the related work in the field of soft error protection on the GPGPU platforms. Section 3 introduces the target

QR algorithm and its implementation in MAGMA. Section 4 models soft error in the QR algorithm, and Section 5 details the recovery algorithm including the optimization of primitives for Givens rotations on the GPU. Section 6 proposes a multiple-error protection algorithm for the left factor Q through tracing the MAGMA QR. Section 7 shows experimental results that evaluate various aspects of our fault tolerant algorithm and, finally, Section 8 concludes the work and outlines possible future directions.

2. 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 a certain interval, either by the message passing middleware automatically or at the request of user application. C/R requires the least user intervention, but suffers from high checkpointing overhead when writing data to stable storage.

To reduce overhead, diskless checkpointing [32] is proposed to use system memory for checksum storage rather than disk storage. Even though diskless checkpointing has seen good applications such as FFT [13] and matrix factorizations [31], it is only suitable for applications that modify small amounts of memory between checkpoints.

Both C/R and diskless checkpointing need the error information for recovery, and unfortunately no such information is guaranteed with soft error. In order to detect error without frequently checking, algorithm based fault tolerance (ABFT) was proposed to remove periodical checkpointing and only perform error check when the execution being protected is finished [20, 2]. This eliminates checkpointing overhead, and the checksum 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 error in systolic arrays. Matrix data is encoded once before the computation begins. 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 back in the 1980s [22, 23] for a single soft error, which was later extended to multiple errors [30, 14, 5] by adopting methodology from error correcting code. These methods for systolic arrays offer promising direction, but requires modification in both algorithm and implementation, especially when dealing with hybrid systems and applications with GPGPU, where soft error could occur from either the host (CPU and the main memory) or the GPU. Soft error in the GPU has been exploited [18], and methods have been developed to detect [36, 40] and recover from error [35, 25, 24]. Recently, soft error in matrix multiplication on a GPU has also been studied [10].

Since the introduction of the 'Fermi' architecture [28], Error Correcting Code (ECC) has been integrated to protect from errors in the GPU global memory, however this adds overhead to communication and reduces overall computing performance.

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

3. HYBRID QR

In linear algebra, a 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 also in QR algorithm which is at the center of a special version of eigenvalue 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, a block version of the Householder transformations is adopted to achieve high performance with the memory hierarchy in modern systems. For example, given an input matrix A, a Householder matrix Q_1 is multiplied to A such that

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

This zeros out the elements under the diagonal in the first column. The next step is carried out on the trailing matrix A' with

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

In practice, MAGMA uses a block version of the QR factorization by accumulating a few steps of the Householder matrix. This version is rich in level 3 BLAS operations and therefore achieves high performance. The result of Q is stored under the lower diagonal of the input matrix in the form of WY representation of Householder transformation products[33, 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, in particular, consisting of a CPU and a GPU. The way to accomplish this is described as follows.

The hybrid QR that we consider has the input matrix and the result on the GPU memory. The computational pattern is similar to the LAPACK's QR – a sequence of 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 *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.

4. SOFT ERROR MODELING

MAGMA algorithms run with both the GPU and CPU, therefore soft errors on both platforms are considered a source of contamination. Also since the result of panel factorization and lookahead trailing panel commutes between the CPU and GPU frequently, soft error could propagate between the GPU and CPU as well, depending on when and where error occurs. To ease the error analysis and avoid dealing with the timing of errors, we adopt the error modeling technique proposed in [23].

4.1 Error Model

Luk et al. derived their model for both LU and QR using the "ZU" notation where Z represents the left factor and U represents the right factor that is upper triangular. We return to the "QR" notation for clarity, and have in mind the right-looking Householder QR algorithm as the implementation method.

Having the initial matrix,

$$A_0 = A,$$

Householder QR is carried out by introducing Householder transforms from the left to get the final triangular form. Let

$$A_t = Q_{t-1}A_{t-1}$$

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

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

e_i is a column vector with all 0 elements except 1 as the i^{th} element. Since no error warning is raised, the factorization continues from step t till the end. If the soft error at step t is viewed as the result of perturbation to an erroneous initial matrix

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

where $d = \lambda(Q_{t-1}\ldots Q_0)^{-1}e_i$, then the erroneous process of QR factorization equals to an error-free QR factorization from a erroneous initial matrix $\tilde{A}$.

In essence, this model treats soft error as a perturbation to the initial matrix similar to rounding errors so that backward error analysis [39] can be used for designing the recovery algorithm.

4.2 Checksum for R

In MAGMA, the right-looking Householder QR algorithm 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. Theorem 4.1 shows 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.

THEOREM 4.1. *Q in Householder QR factorization cannot be protected by performing factorization along with the vertical checksum.*

PROOF. Append a $m \times n$ nonsingular matrix A with checksum GA of size $c \times n$ along the column direction to get matrix $A_c = \begin{bmatrix} A \\ GA \end{bmatrix}$. G is $c \times m$ generator matrix. Suppose A has a QR factorization Q_0R_0.

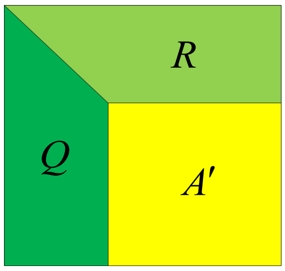

Figure 1: Different regions of A during factorization

Perform QR factorization to A_c:

$$\begin{bmatrix} A \\ GA \end{bmatrix} = Q_cR_c = \begin{bmatrix} Q_{c11} & Q_{c12} \\ Q_{c21} & Q_{c22} \end{bmatrix} \begin{bmatrix} R_{c11} \\ \varnothing \end{bmatrix}$$

Q_{c11} is $m \times m$ and Q_{c21} is $c \times m$. R_c is $m \times n$ and $\varnothing$ represents $c \times n$ zero matrix. $R_c \neq 0$ and is full rank. R_c is upper triangular with nonzero diagonal elements and therefore nonsingular.

$$Q_cQ_c^T = \begin{bmatrix} Q_{c11} & Q_{c12} \\ Q_{c21} & Q_{c22} \end{bmatrix} \begin{bmatrix} Q_{c11}^T & Q_{c21}^T \\ Q_{c12}^T & Q_{c22}^T \end{bmatrix} = I$$

Therefore,

$$Q_{c11}Q_{c11}^T + Q_{c12}Q_{c12}^T = I. \tag{3}$$

Since $A = Q_{c11}R_{c11}$ and R_{c11} is nonsingular, then $Q_{c11} \neq 0$ and is nonsingular.

Assume $Q_{c12} = 0$:

$Q_{c11}Q_{c21}^T + Q_{c12}Q_{c22}^T = 0$, therefore $Q_{c11}Q_{c21}^T = 0$. We have shown that Q_{c11} is nonsingular, so $Q_{c21}^T = 0$ and this conflicts with $GA = Q_{c21}R_{c11} \neq 0$, so the assumption $Q_{c12} = 0$ does not hold. From Equation 3, $Q_{c11}Q_{c11}^T \neq I$. This means even though $A = Q_{c11}R_{c11}$, $Q_{c11}R_{c11}$ is not a QR factorization of A. □

Therefore, Q is protected by static checkpointing in this work.

The part of the matrix other than Q is divided into two regions, the already formed R and the trailing matrix A', as shown in Fig. 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 static checkpointing as for Q. For R, we adopt the ABFT technique from [2, 21], which was also used in Luk's work [22, 23] for soft error in systolic arrays.

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

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

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

$c, v \in \mathbb{R}^{m\times 1}$ are checksum columns after factorization.

Due to soft error, A becomes the erroneous matrix $\tilde{A}$, and the checkpointed matrix becomes

$$(\tilde{A}\ Ae\ Aw)$$

And the QR factorization becomes:

$$(\tilde{A}\ Ae\ Aw) = \tilde{Q}(\tilde{R}\ \tilde{c}\ \tilde{v}) \tag{5}$$

From eq. 5

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

By the same token,

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

Assume residual vectors $r, s \in \mathbb{R}^{m\times 1}$

$$\tilde{r} = \tilde{c} - \tilde{R}e = \tilde{Q}^{-1}d \quad (6)$$

and

$$\tilde{s} = \tilde{v} - \tilde{R}w = w_j\tilde{Q}^{-1}d \quad (7)$$

Combining Equation 6 and 7,

$$\tilde{s} = w_j\tilde{r}. \quad (8)$$

$\tilde{r}$ can be used to check for error, and in case an error occurs, the column in which the error initially strikes can be determined by Equation 8.

5. RECOVERY ALGORITHM

With the knowledge of error column j, Luk et al. [23] recommended a spike-reducing technique to recover the left and right factors of *ZU* factorization without giving the actual algorithm. In this section we continue this work on a slightly different path due to the storage format of MAGMA QR.

5.1 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 Equation 2.

$$\begin{aligned}A-\tilde{A} &= (a_{\cdot j}-\tilde{Q}\tilde{R}_{\cdot j})e_j^T\\ A &= \tilde{Q}\tilde{R}+(a_{\cdot j}-\tilde{Q}\tilde{R}_{\cdot j})e_j^T\\ A &= \tilde{Q}\tilde{R}+\tilde{Q}(\tilde{Q}^Ta_{\cdot j}-\tilde{R}_{\cdot j})e_j^T\\ A &= \tilde{Q}(\tilde{R}+pe_j^T)\\ A &= \tilde{Q}\tilde{C},\ C=\tilde{R}+pe_j^T, p=\tilde{Q}^Ta_{\cdot j}-\tilde{R}_{\cdot j} \quad (9)\end{aligned}$$

C in Equation 9 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 related to 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 we choose Givens rotation as the non-zero elimination algorithm. In [23], Luk et al. also suggested a few methods including Givens rotation to eliminate this spike with matrix factorization modifying method [16] in $O(k^2)$ steps. Since Givens rotation is memory-bound, implementation on the GPU requires careful design for the best performance. This will be covered in section 5.3.

5.2 QR Update as the Recovery Algorithm

From Equation 2, it can be seen that the recovery algorithm is in essence a QR update problem. Since QR update is also widely used in applications where repeated updating is required [37], this work implements the QR update algorithm for the GPU and applies it 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 $v = e_j$,

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

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. It can be shown that $H = J^T \times R$ is an upper Hessenberg matrix, and therefore

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

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 place or the CPU 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.

5.3 Givens Rotation Utilities for the GPU

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

5.3.1 *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,

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1N} \\ x_{21} & \cdots & x_{2N} \end{bmatrix} = \begin{bmatrix} y_{11} & \cdots & y_{1N} \\ y_{21} & \cdots & y_{2N} \end{bmatrix} \tag{10}$$

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 make this situation worse, 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.

5.3.2 *Data Caching*

In Algorithm 1, DLARTG and DROTG are fused together to firstly create the upper Hessenberg matrix H, and then reduce it to upper triangular. This common operation has two 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 Equation 10 (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

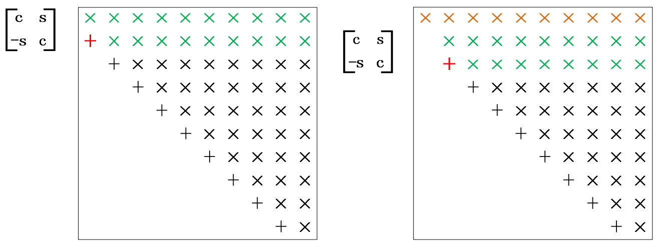

Figure 2: Reduction from upper Hessenberg to upper triangular

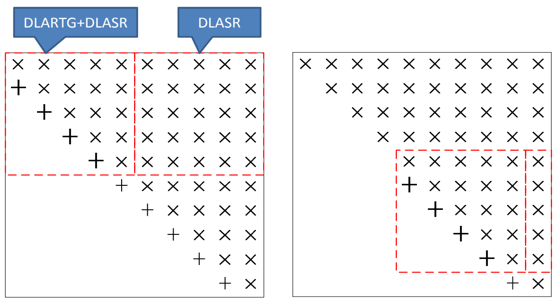

Figure 3: Reduction from upper Hessenberg to upper triangular (block algorithm)

the current step. This operation sweeps from top to bottm 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. Naturally 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 functionalies 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.

5.3.3 *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 [11]. 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 2 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 in use except for edge cases. In our experiment, with a Tesla T20, aka '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. 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 functionalies. 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.

5.3.4 *Efficient Memory Access Scheme*

Figure 4 is a modified memory access scheme to remedy the problem discussed in section 5.3.1 for the type II kernel in section 5.3.3.

In the original kernel, all threads are lined up in a row, and during each step each thread fetches two values in a column along with a Given rotation pair from global memory. For double precision (8-byte word) memory access within half warp to be coalesced, CUDA requires all 16 words to fall in the same 16-word segment [29] but since each element in consecutive columns of this row are separated by the leading dimension, the coalescing rule does not hold.

In order to benefit from the throughput advantage provided by coalescing, a level of inner blocking is added to the kernel. Take Tesla T20 for example where the maximum number of thread per thread block is 1024. Rather than striding one row down at each step, a 4×64 block of data (yellow) are fetched together from global memory to the corresponding 64×4 piece in a shared memory buffer of size 1024×4 using a 16×64 layout of the 1024 threads such that all 16 threads in each column of the grid have consecutive thread IDs. Therefore the $4 \times 64 = 256$ elements in the yellow zone are fetched by 64 coalesced accesses. These fetching loops continue from left to right until the four rows are completely loaded. After the loading, thread layout is re-arranged to 1024×1 in the inner blocking. Each thread loads two consecutive elements in a row from the shared memory. The layout of the shared memory buffer lowers the bank conflict to minimum. The inner blocking loop consumes the four rows of data (four columns in shared memory) to apply the corresponding Givens rotations, and once this four rows are finished, results are written back in the same coalesced manner as loading.

The scheme described in this section can also be used for other

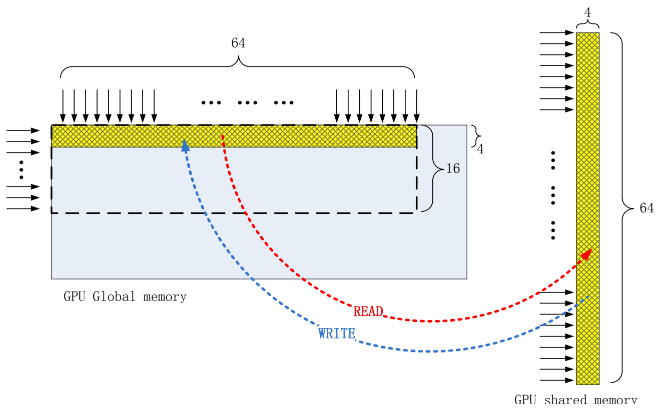

Figure 4: Global memory accesses in the blocked DLASR kernel

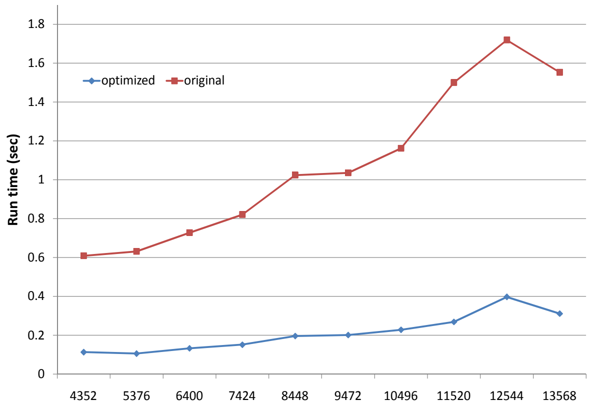

Figure 5: Run time comparison of the blocked DLASR (optimized) kernel and the original version

similar kernels in this work.

5.3.5 *Improvement Experiment*

Figure 5 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 7.3 where the impact of the new reduction algorithm on recovery performance is shown in Figure 9. By using a more efficient memory access pattern and the blocked algorithm for fused DLARTG and DLASR operation, 5x speedup is achieved.

6. PROTECTION FOR Q

Theorem 4.1 has shown that Q cannot be protected by ABFT as R, and the spike-eliminating algorithm 1 inherited from work by Luk et al. [23] function under the assumption that no soft error strikes $\tilde{Q}$, which is the erroneous Q caused by soft error in R or A'. In MAGMA QR, since Q occupies half of the matrix, it is as eligible to be soft error victim as other section of the matrix and therefore has to be protected.

6.1 Static Checkpointing for Q

In order to provide soft error resilience to Q, we propose to use a checkpointing algorithm because once a panel is factorized on the CPU, the result remains unchanged until the end of factorization.

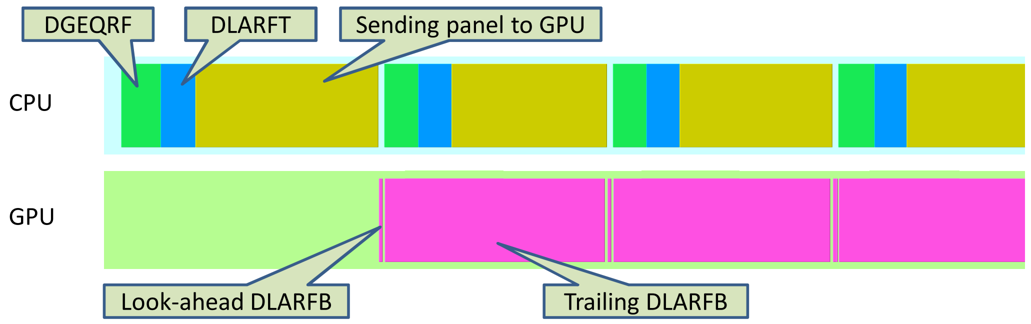

Figure 6: MAGMA QR tracing

For any column of the factorized panel $[v_1, v_2, \cdots, v_k]^T$, 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 [12] that one soft error in a column of L in LU can be protected with trivial overhead. In this work, we extend this to two errors per column in Q.

For any column of the factorized panel, $[v_1, v_2, \cdots, v_k]^T$, the vertical checkpointing produces the following two sets of checksums:

$$\begin{cases} v_1 + v_2 + \cdots + v_k = c_1 \\ w_1 v_1 + w_2 v_2 + \cdots + w_k v_k = c_2 \\ u_1 v_1 + u_2 v_2 + \cdots + u_k v_k = c_2 \end{cases} \tag{11}$$

Since all computation are carried out in floating point number with a fixed number of digits for exponent and fraction, the selection of w_i and u_i must fall in a certain range such that they do not cause large rounding errors. For example, in [14], the use of Vandermonde matrix where $w_i = j$ and $u_i = j^2$ make it not practical on modern computing systems because the fast increase of checkpointing weight causes notable precision loss by rounding errors. In this work we choose w_i and u_i as random numbers. Supposed errors cause v_i and v_j to be changed to $\tilde{v_i}$ and $\tilde{v_j}$, $i < j$, we have:

$$\begin{cases} v_1 + \cdots + \tilde{v_i} + \cdots + \tilde{v_j} + \cdots + v_k = \tilde{c_1} \\ w_1 v_1 + \cdots + w_i \tilde{v_i} + \cdots + w_j \tilde{v_j} + \cdots + w_k v_k = \tilde{c_2} \\ u_1 v_1 + \cdots + u_i \tilde{v_i} + \cdots + u_j \tilde{v_j} + \cdots + u_k v_k = \tilde{c_2} \end{cases} \tag{12}$$

Subtract Equations 12 from Equations 11, we have

$$\begin{cases} \tilde{c_1} - c_1 = \tilde{v_i} - v_i + \tilde{v_j} - v_j \\ \tilde{c_2} - c_2 = w_i(\tilde{v_i} - v_i) + w_j(\tilde{v_j} - v_j) \\ \tilde{c_3} - c_3 = u_i(\tilde{v_i} - v_i) + u_j(\tilde{v_j} - v_j) \end{cases} \tag{13}$$

In order to solve u_i, u_j, w_i and w_j, let $u_i = w_i^2$, $i = 1 \cdots k$. Equations 13 can be reduced to

$$(\tilde{c_3} - c_3) - (w_i + w_j)(\tilde{c_2} - c_2) + w_i w_j (\tilde{c_1} - c_1) = 0 \tag{14}$$

And w_i, w_j can be determined by iterating through all possibilities with $O(n^2)$ complexity because $i < j$, and for each i, $n - i$ pairs of w_i w_j are tested in Equation 14.

This checkpointing method also applies to one-error recovery. Supposed an error occurs to v_i only, and Equation 11 becomes

$$\begin{cases} \tilde{c_1} - c_1 = \tilde{v_i} - v_i \\ \tilde{c_2} - c_2 = w_i(\tilde{v_i} - v_i) \\ \tilde{c_3} - c_3 = u_i(\tilde{v_i} - v_i) \end{cases} \tag{15}$$

The same method in [12] can be used to determine w_i.

Using Equation 13, the error detection and recovery algorithm is in Algorithm 2. Note that this error protection for Q applies for each column of Q.

Algorithm 2 Error detection and recovery in Q

```
Require: $\tilde{A}$, error column $s$, $w_i \neq w_j$, $i, j \in \{1 \cdots k\}$
  Calculate $\hat{c_i} = \tilde{c_i} - c_i$, $i = 1, 2, 3$
  if $\hat{c_i} == 0$, $i = 1, 2, 3$ then
    No error
  else if $\hat{c_2}/\hat{c_1} == \hat{c_3}/\hat{c_2} == w_i$ then
    One error in row $i$, column $s$ of the output matrix
    Recover by solving $\tilde{c_1} - c_1 = \tilde{v_i} - v_i$
  else
    At least two errors in column $s$ of the output matrix
    Iterate all possible pairs $w_i$, $w_j \in w$
    if $(\tilde{c_3} - c_3) - (w_i + w_j)(\tilde{c_2} - c_2) + w_i w_j (\tilde{c_1} - c_1) = 0$ then
      Two errors are in rows $i$ and $j$, column $s$ of the output matrix
      Recover by solving the overdetermined least square equations in Equation13 with $w_i$ and $w_j$ as known constants and $x = \tilde{v_i} - v_i$ and $y = \tilde{v_j} - v_j$ as unknowns
    else
      More than two errors occurs
    end if
  end if
```

The error detection and recovery algorithm can be extended to t errors with complexity $O(n^t)$ to determine the locations of errors. Since the complexity of QR factorization is $O(n^3)$, when $t > 3$, it becomes more expensive by doing recovery than actually restarting the QR factorization or obtaining Q by solving $Q = AR^{-1}$. The overhead of $O(n^t)$ can be improved by implementing on the GPU with a parallelized search algorithm for error locations. This will be discussed in the future research.

6.2 Timing of Checkpointing

The checkponting for Q is carried out once per iteration. Therefore the placement of this procedure is critical to avoid large performance penalty, for example, by sitting on the critical path of execution.

As described in section 3, The GPU onsite version of MAGMA QR produces Q using the CPU implementation DGEQRF and during step i, an $M_i \times NB$ 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 6 generated by TAU (Tuning and Analysis Utilities) [6]. The size of this run is 17408×17408, and only the first few iterations are shown.

To minimize performance penalty, the checkponting is preferred to reside in a time slot of overlapping between the CPU and GPU. 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. By 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 started previously are finished. From Figure 6, 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 put 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 therefore 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. For such a situation, the checkpointing could be moved to run on the GPU between the time GPU DLARFB finishes and the initiation of cublasSetMatrix on the CPU.

7. PERFORMANCE EVALUATION

In this section we evaluate the performance of the fault tolerant QR algorithm on a hybrid system. The configuration of the experiment environment is in table 1:

MKL with 48 threads is used on the CPU and CUDA 4.0 is driving the GPU. All computing is in double precision and based on MAGMA version 1.0. The maximal matrix size is limited by the GPU global memory.

As discussed in section 5.2, the recovery algorithm requires a column of the original matrix. While this column may be re-generated cheaply, in our experiment we want to simulate the worst case where this convenience is not available, and therefore the original matrix is duplicated for the recovery process. Since the GPU memory is relatively small compared to that of the host, and is normally fully utilized for computing, the copy of the original matrix is put on the host memory. To avoid performance impact, the data transferring is performed asynchronously during the first panel factorization. The panel data is copied first so that DGEQRF on the CPU could start as soon as possible, and while the CPU is busy with the panel factorization, the rest of data is copied through DMA to the host memory. All the performance results shown in this section include this overhead.

	Brand	Frequency	# cores	Memory
CPU	AMD Opteron 6180 SE	2.5 GHz	48	256 Gb
GPU	NVIDIA C2050	1.1 GHz	14	2.7 Gb

Table 1: Experiment configuration

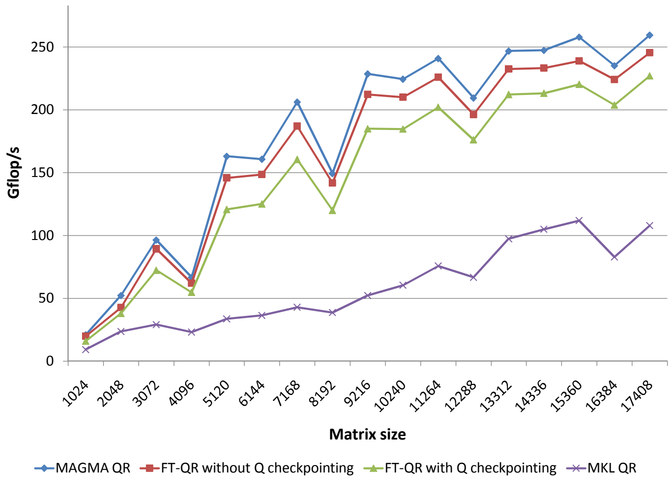

Figure 7: Performance of FT-QR with/without checkpointing for Q

7.1 Overhead Analysis

The overhead of fault tolerance comes from the following sources:

1. Duplicating the original matrix from the GPU to the CPU
2. Generating checksum on the GPU
3. Performing QR with two checksum columns on the GPU
4. checkpointing Q on the CPU
5. Check for error in R and A' on the GPU
6. Check for error in Q on the GPU
7. Recovery from error in Q on the CPU and GPU
8. Recovery from error in R and A' on the GPU

Each item of the overhead sources, except the memory copy, requires $O(n^2)$ extra FLOPS. And comparing to the $\frac{4}{3}n^3$ FLOPS of QR factorization, the overhead fades away when matrix size is large enough.

7.2 Checkpointing of Q

Figure 7 is an experiment to show the overhead caused by checkpointing Q. The red line shows the performance without checkpointing Q and the performance between the red line and blue line is the overhead caused by (1)-(3) and (5)-(6) in the overhead source list. With the checkpointing Q switched on, the green line performance dips by another 5% at large matrix sizes. The green line represents the case of our fault tolerant QR runs without any error. To compare the performance with the CPU implementation, the result of MKL QR running with 48 threads is also shown. It can be seen that even with the overhead of fault tolerance, our FT-QR is still showing 2-3x speedup over the CPU implementation.

7.3 Recovery

Since our algorithm can deal with errors in the full matrix, the recovery performance are divided into the left factor and the right factor.

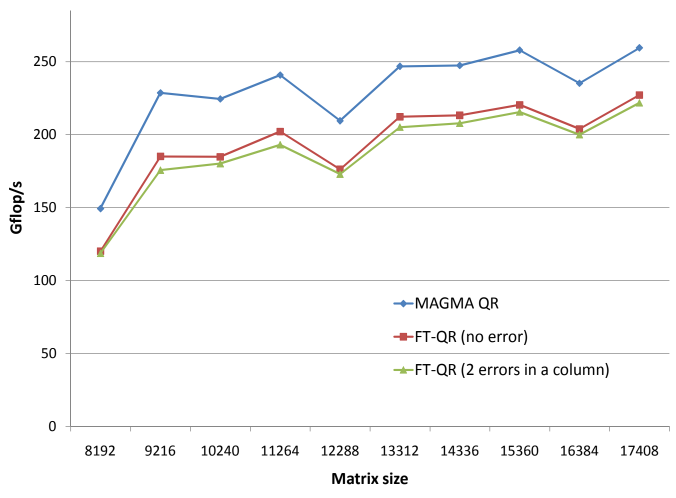

Figure 8: Performance of Recovery for Errors in Q

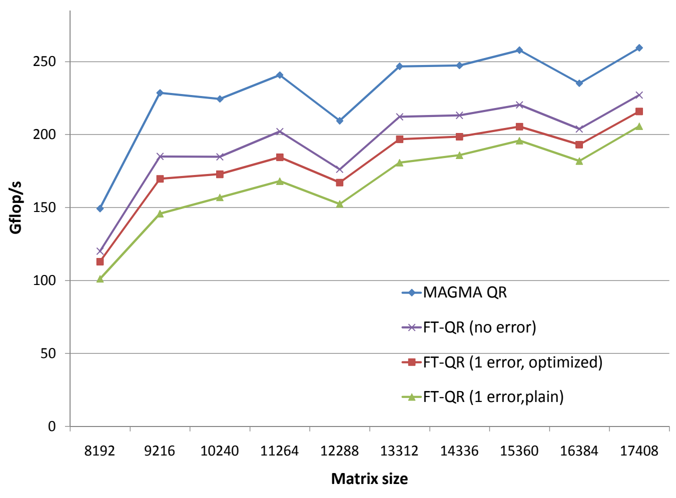

Figure 9: Performance of Recovery for Error in R and A'

7.3.1 *The Right Factor*

Figure 8 is the performance of recovery from errors in Q. Two errors are injected to column 312 in rows 612 and 729 respectively. This experiment is simulating random double errors in a column of Q and therefore the error locations are not informed to the recovery algorithm. Performance result shows a small overhead from the no-error case of the fault tolerant QR, and about 15% decrease from the original MAGMA QR. This percentage will continue to drop as matrix sizes grows larger permitted by GPU with larger global memory.

7.3.2 *The Left Factor*

Figure 9 is the performance of recovery from error initially in R or A'. For all matrix sizes, error is injected to a random location (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 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 QR. This percentage will also continue to drop with larger matrix sizes.

8. CONCLUSION

In this work we developed a soft error resilient QR algorithm for hybrid architecture where the CPU and GPU are utilized together. This work enables the high performance implementation of MAGMA QR to be tolerant to soft errors caused by radiation-based interference.

Based on the ABFT algorithm by Luk et al., the FT-QR algorithm can tolerate up to one soft error in data section R and A'. Since the recovery algorithm requires an error-free left factor Q, which is not guaranteed by Luk's algorithm, a stable and scalable multiple-error checkpointing/recovery mechanism is devised and placed in the computing environment based on the execution feature of MAGMA QR such that the checkpointing is hidden away from the critical path and therefore prevents severe performance impact. In addition, a more efficient recovery algorithm based on Givens rotation is designed. This fast Givens rotation utilities can also be used in other applications to reduce an upper Hessenberg matrix to upper triangular on the GPU.

To future work, we will extend this FT-QR to multi-GPU for larger problem size, and the multiple-error encoding method will be applied to the protection of the right factor.

Acknowledgment

The authors would like to thank Scott Biersdorff at the Performance Research Lab at the University of Oregon for his generous help on our experiment with TAU on the GPU, and Sam Crawford for the help on the text.

9. 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] 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.

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

[12] 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.

[13] E. Elnozahy, D. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*, pages 39–47. IEEE, 1991.

[14] 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.

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

[16] P. Gill, G. Golub, W. Murray, and M. Saunders. Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535, 1974.

[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*, 36th 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] C. Nvidia. Nvidia cuda c programming guide. *NVIDIA Corporation*, 2011.

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

[31] J. Plank, Y. Kim, and J. Dongarra. Algorithm-based diskless checkpointing for fault-tolerant matrix operations. In *ftcs*, page 0351. Published by the IEEE Computer Society, 1995.

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

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] J. Wilkinson. *The algebraic eigenvalue problem*. Oxford University Press, USA, 1965.

[40] 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.