Online Algorithm-Based Fault Tolerance for Cholesky Decomposition on Heterogeneous Systems with GPUs

Jieyang Chen, Xin Liang, and Zizhong Chen
Department of the Computer Science and Engineering
University of California, Riverside
{*jchen098, xlian007, chen*}*@cs.ucr.edu*

***Abstract*—Extensive researches have been done on developing and optimizing algorithm-based fault tolerance (ABFT) schemes for systolic arrays and general purpose microprocessors. However, little has been done on developing and optimizing ABFT schemes for heterogeneous systems with GPU accelerators. While existing ABFT schemes can correct computing errors like 1+1=3, we find that many memory storage errors can not be corrected by existing ABFT schemes. In this paper, we first develop a new ABFT scheme for Cholesky decomposition that can correct both computing errors and storage errors at the same time, and then develop several optimization techniques to reduce the fault tolerance overhead of ABFT for heterogeneous systems with GPU accelerators. Experimental results demonstrate that our fault tolerant Cholesky decomposition is able to correct both computing errors and storage errors in the middle of the computation and can achieve better performance than the state-of-the-art vendor provided version Cholesky decomposition library routine in CULA R18.**

I. Introduction

Today's computing systems are susceptible to errors. This can be evidenced from the fact that ECC protected DRAMs and RAID protected disks have been widely used to protect errors in today's computing systems. Heterogeneous systems with both CPUs and GPUs have been proven to be efficient to accelerate a variety of HPC applications. However, like the traditional computing systems with only CPUs, errors also occur frequently in heterogeneous systems. In [1], the authors presented the first large-scale study of the error rate for GPUs. Their test results show that there are two-third of the tested GPU hardware exhibit pattern-sensitive soft errors in GPU memory or logic part. Moreover, in [2], the authors proposed a GPGPU-SODA framework to analysis the soft error vulnerability of GPGPU micro-architecture. They observed that GPUs exhibit high soft error susceptibility, and the vulnerability is sensitive to workload characteristics.

When an error occur, if the computing system continues without interruption, it is often called a fail-continue error. Otherwise, it is often called a fail-stop error. While extensive researches have been done to tolerate fail-stop errors [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], this paper restricts its scope to tolerating fail-continue errors in Cholesky decomposition. Cholesky decomposition has been widely used to solve linear equations arising from linear least squares problems, non-linear optimization, Monte Carlo simulations, and Kalman filters.

Fail-continue errors are sometimes also called soft errors. A simple general approach to tolerate soft errors is the Triple Modular Redundancy (TMR) approach. In TMR, three identical computations are first performed on three different hardware platforms (or on the same hardware platform but replicated for 3 times for tolerating transient errors), then the three computational results are compared and voted. The correction computation result is the majority of the three results. For the purpose of detecting errors, Double Modular Redundancy (DMR) is a general approach that works by comparing the results of two identical computations on two different hardware platforms. While DMR and TMR are general approaches that can be applied to any application, they introduce very high overhead (i.e., 100% overhead to detect errors and 200% overhead to correct errors).

It is very well known that, for matrix operations, the algorithm-based fault tolerance (ABFT) technique developed by Huang and Abraham in [13] introduces much lower fault tolerance overhead than the general techniques DMR and TMR. In [13], Huang and Abraham proved that, for many matrix operations, the checksum relationship in the input checksum matrices is still held in the final computation results. Therefore, if the failed processor is able to continue their work and finish the computation, errors can be detected by verifying whether or not the checksum relationship is still held in the final computation results. Because the original computation complexity for the input matrix size n is in the order of $\mathcal{O}(n^3)$ but the error detection complexity is in the order of $\mathcal{O}(n^2)$, ABFT introduces much lower overhead than DMR. ABFT is originally proposed to detect miscalculations in matrix operations on systolic arrays offline after the computation is finished. Because of its low overhead, it is later extended to by many researchers detect and correct soft errors offline on general purpose microprocessors [14] and handle fail-stop errors on modern parallel computing clusters [15]. In [16], ABFT is also extended to correct fail-continue soft errors online in the middle of the computation so that corrupted computations can be corrected in a much more timely manner.

A. Limitations of the existing ABFT

Despite that tremendous progresses have been made in the field of algorithm-based fault tolerance for matrix operations, existing ABFT schemes for Cholesky decomposition have the following limitations:

1) **Capability to correct storage errors is limited:** Traditional ABFT schemes [13] correct errors offline at the end of the computation. They do not distinguish between computing errors (i.e., 1+1=3) and storage errors (i.e., 0 becomes 1). Because of the propagation of the error, one error in one element often causes numerous errors in the computation results. Furthermore multiple errors will accumulate. Therefore, it is either impossible or very expensive to correct storage errors at the end of the computation. While most recent online ABFT scheme [17], [18], [19] can correct computing errors online in the middle of the computation,

storage errors occurred between checksum verification and the next data access can not be corrected.

2) **Few schemes are developed and optimized for heterogeneous systems with GPU accelerators:** Classical ABFT schemes [13] are originally developed for systolic arrays. Because of their low overhead to detect errors, they have been extended by many researchers to detect and correct errors on modern general purpose microprocessors. However, little has been done on developing and optimizing ABFT schemes for heterogeneous systems with both CPUs and GPUs.

B. Our Contributions

This paper develops a new heterogeneous-system based ABFT scheme for Cholesky decomposition to correct both computing errors and storage errors at the same time. Several optimization techniques are developed to reduce the fault tolerance overhead for heterogeneous computing systems with both CPUs and GPUs. Experimental results demonstrate that our fault tolerant Cholesky decomposition is able to correct both computing errors and storage errors in the middle of the computation and achieve better performance than the state-of-the-art vendor provided version Cholesky decomposition library routine in CULA R18 [20]. Cholesky decomposition has been widely used to solve linear equations arising from linear least squares problems, non-linear optimization problems, Monte Carlo simulations, and Kalman filters. An efficient and fault tolerant implementation of Cholesky decomposition can therefore benefit a large number of users and a wide range of scientific fields. More specifically, the contributions of this paper include:

1) **First Online-ABFT scheme to correct both computing and storage errors:** Current state-of-the-art online-ABFT [18], [17], [21], [22], [23], [24], [25], [26] is based on post-updating-verification scheme, which verifies the data correctness immediately after updating the matrix. However, the correctness of the data between one verification and its immediate next reading is not protected. Therefore, errors occurred during this period will be propagated to pollute too many elements to correct. We designed a new online ABFT scheme that verifies the correctness of the matrix elements immediately before the data are accessed. Therefore, both computing errors and storage errors can all be detected and corrected before the using the matrix elements for the next stage of the computation. To the best of our knowledge, our ABFT scheme is the first online ABFT scheme that is able to correct both computing and storage errors at the same time.
2) **First Online-ABFT scheme for Cholesky decomposition on heterogeneous systems with GPU accelerators:** Existing ABFT schemes for Cholesky factorization are designed/optimized either for systolic arrays [13] or for general purpose microprocessors [18]. To the best of our knowledge, our ABFT scheme is the first ABFT scheme for Cholesky decomposition on heterogeneous systems with GPU accelerators.
3) **Innovative overhead reduction techniques for ABFT:** This paper developed three novel optimization techniques to optimize ABFT overhead on heterogeneous systems with GPU accelerators. Checksums recalculation is the key operation for data correctness verification in ABFT. It is an relatively expensive (i.e., $\mathcal{O}(n)$) operation on the critical path. Because it consists of several BLAS Level-2 operations, the efficiency of executing checksums recalculation on GPU is low. This paper designed an optimization approach that allows several BLAS Level-2 checksums recalculation operations being executed on the GPU concurrently using CUDA concurrent kernel execution feature. Checksum updating is another relatively expensive (i.e., $\mathcal{O}(n)$) operation in ABFT. It is non-trivial to decide whether the checksum updating operation should be executed on CPU or GPU. This paper designed a model to help to make this decision based on the relative speed of the involved CPU and GPU. Existing online ABFT schemes introduce considerable overhead because it verifies checksums at the end of every outer iteration. This paper significantly reduces the overhead for checksum verification by verifying checksums every k iterations, where k is a parameter related to the failure rate of the system.

II. Background

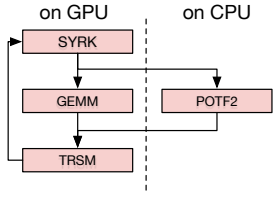
In the section, we provide several backgrounds that are necessary to understand the key ideas of this paper.

A. Cholesky Decomposition in MAGMA

Algorithm 1 MAGMA's Cholesky Decomposition

Require: Positive-definite $n \times n$ matrix A
1: **for** $j = 1$ to N **do**
2: [GPU] Do rank-k update: $A[j,j] -= A[j,0:j-1] \times A[j,0:j-1]^T$
3: Transfer $A[j,j]$ to CPU main memory $A_{CPU}[j,j]$
4: [GPU] Do matrix-matrix multiplication: $A[j+1:N,j] -= A[j+1:N,0:j-1] \times A[j,0:j-1]^T$
5: [CPU] Do single block Cholesky decomposition: $A_{CPU}[j,j] \rightarrow L_{CPU}[j,j]$
6: Transfer $L_{CPU}[j,j]$ to GPU main memory $L[j,j]$
7: [GPU] Solve linear systems: $A[j+1:N,j] = A[j+1:N,j] \times L[j,j]^T$
8: **end for**

Figure 1. MAGMA's Cholesky decomposition



MAGMA is a linear algebra library that utilizes the heterogeneous systems with GPU. CPU and GPU have different specialty in handling computation tasks: CPU is more efficient at doing less parallelized irregular patterned calculations; GPU, on the other hand, is better at handling highly parallelized calculations. So, to better utilize this characteristic of the heterogeneous systems, MAGMA assigns different operations to different computation units based on the degree of their parallelization. For Cholesky decomposition, MAGMA chose the inner product version

because it has more BLAS Level-3 operations, hence, can utilize the heterogeneous system more efficiently. As shown in Figure 1 and Algorithm 1, less parallelized POTF2 (line 5) is assigned to CPU and high-parallelized TRSM (line 7), SYRK (line 2) and GEMM (line 4) are assigned to GPU. Moreover, most data transfer (line 3 and 6) and POTF2 (line 5) on CPU are hidden by the most time-consuming GEMM (line 4) operation on GPU. So, the Cholesky decomposition in MAGMA is very efficient on heterogeneous systems.

B. Offline-ABFT and Online-ABFT

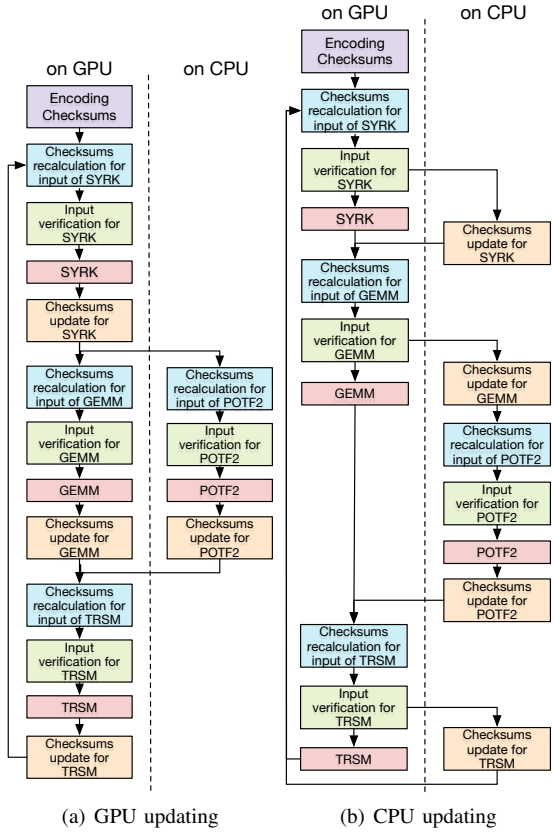
Offline-ABFT was first introduced by Abraham and Huang [13] to handle computing errors. The main idea is that, for a matrix operation $P(A_1, A_2, ..., A_n) = (X_1, X_2, ..., X_m)$, if we encode the input matrices into their checksum form, then apply the operation on the encoded matrices, the results are still encoded with checksums, which can be used to detect and correct error(s) in results.

$$P({A_1}^{enc}, {A_2}^{enc}, ..., {A_n}^{enc}) = ({X_1}^{enc}, {X_2}^{enc}, ..., {X_m}^{enc}).$$

The detection and correction is done after the whole computation is complete. It can handle non-propagating soft errors, but unable to handle errors that propagate.

Online-ABFT was first introduced by Davies and Chen [16] to correct errors before they propagate. The key idea is that checksums are not only ensured to be consistent by the end of computation, they are also maintained during computation. So they can be used to detect and correct errors in the middle of computation. Thus, any error could be corrected in a timely manner to avoid error propagation.

Figure 2. Enhanced Online-ABFT



(a) GPU updating (b) CPU updating

III. Design of Enhanced Online ABFT

In the current state-of-the-art Online-ABFT, checksums are maintained in the middle of computation. After each operation completes, those checksums are used to detect and correct any error in the result. Basically, for each operation, Online-ABFT consists 4 steps ordered as follow:

1) Original updating operation;
2) Corresponding checksums updating operation;
3) Checksums recalculation for result data;
4) Result error detection and correction.

However, the limitation of current Online-ABFT is that the data stored in memory is not protected from memory storage error, which could corrupt the result or even leads to fail-stop failure. To illustrate this problem, we show a general updating process, which is the kind of operation that takes up the majority computation of almost any matrix operation. For example, a data block A has just been updated and it will be used to update a data block B in a moment. The details in this process is shown as follow:

1) Data block A has just been completely updated;
2) As the result of updating, the errors in A are detected and corrected by Online-ABFT immediately;
3) Data block A has to wait in the memory for some other related tasks to complete before it can be referenced for updating B;
4) Updating data block B using data block A;
5) Again, as the result of the updating, the errors in B are detected and corrected by Online-ABFT immediately.

First, we can ensure the correctness of data block A after Step 2. However, it has been stored in the memory for a while in Step 3, so data block A could have memory storage errors in it. Also, data block B is stored in memory and may has not been verified recently. So, potentially both data block A and B could have memory storage errors before the updating in Step 4. Moreover, the updating in Step 4 potentially could generate computation errors in B. All these errors could persist and eventually affect the correctness of data block A and B in Step 4. Fortunately, Online-ABFT can handle the errors in B in Step 5. However, no one can guarantee the correctness of data block A now. Because A has already been updated, and it will not be updated or verified anymore in the future, so incorrect data will persist. Even worse, if data block A is going be used to update other part of the matrix, errors in A will propagate and in some cases it would cause unrecoverable or even fail-stop error. For example, a single memory error in a matrix block could break the positive-definite property of that block before the unblocked decomposition in Cholesky decomposition, which leads to termination of the whole process.

Even thought, the time interval between a data block is verified and referenced could be vary short and memory storage error doesn't occur often, however, as the problem size increases with the memory space growth in current HPC systems, this kind of time interval could be longer and the probability of memory storage error could be higher.

Although some memory storage errors can be fixed by the ECC feature in memory, ECC can only fix a single bit error in a byte. If there are more than one bit flipped, ECC cannot correct them, so the result is still incorrect.

To overcome this limitation in current Online-ABFT, we designed an innovative Enhanced Online-ABFT, in which both computation and memory storage error can be detected

and corrected. The main idea is that data blocks are no longer verified after they are updated, instead, data blocks are verified before each time they are referenced. So, any error including calculation error from last operation or memory bit-flips error occurred during storage can be corrected before use. For specific, each operation in our Enhanced Online-ABFT consists 4 steps and ordered as follow:

1) Checksums recalculation for input data;
2) Input error detection and correction;
3) Original updating operation;
4) Corresponding checksums updating.

IV. Implementation

Our implementation of Enhanced Online-ABFT Cholesky decomposition is based on the MAGMA's Cholesky decomposition routine. We choose the version, in which the initial matrix is stored on GPU memory. The overall design is shown in Figure 2. (We use slight different assignment strategies for different systems and we will explain it in Optimization 2 of next section) As we can see, each data input, including the data to be updated and the data to be referenced, is verified by checksums at first to ensure correctness. Then, the original updating operation is performed. Finally, corresponding checksums is updated to prepare for future correctness verification if it is referenced. The error correction before the updating operation ensures the correctness of the input for the immediate next upcoming updating operation, which not only ensures the correctness of the final result, but also prevents error propagation that may causes unrecoverable errors or fail-stop failure. As for the implementation details, we focus on three parts:

1) Encoding input matrix with checksums before Cholesky decomposition
2) Updating checksums during the decomposition
3) Detecting and correcting errors using checksums after each operation

A. Encoding Checksums

To encode a input matrix with checksum, the matrix is multiplied by a specially designed checksum vector to get the checksum. The resulted checksum can be row checksum, column checksum and full checksum. In MAGMA's blocked Cholesky decomposition, the input matrix is divided into blocks, which each one of them is treated as a updating unit. So, similarly we choose to encode the input matrix using the matrix block as a unit instead of the whole matrix. Although this strategy brings slightly more memory space overhead, it significantly strengthen the fault tolerance density.

Encoding one checksum is only good enough to verify the correctness of matrix blocks. To locate and correct errors, a second checksum with a different weight (calculated by a different weighted checksum vector) need to be added. Generally, $m+1$ column/row checksums could locate and correct up to m errors per column/row. As mentioned by [18], two row checksums or two column checksums works the best for Cholesky decomposition, so that they can locate and correct up to one error per column in a matrix block. We choose two column checksums in our Enhanced Online-ABFT Cholesky decomposition. The process of checksums encoding with two column checksums is illustrated as follow: (It is similar for two row checksums). First, we choose two weighted checksum vectors to be: $v_1 = [1, 1, 1...1]$ and $v_2 = [1, 2, 3...B]$, where B is the matrix block size. The matrix block to be encoded is $A = [a_1, a_2, a_3...a_B]$ with a_i represents the i^{th} column of A. So, the two column checksums can be calculated as:

$$chk_1 = {v_1}^T A = [r_{11}, r_{12}, r_{13}...r_{1B}]$$

$$chk_2 = {v_2}^T A = [r_{21}, r_{22}, r_{23}...r_{2B}]$$

For better efficiency, all checksums for an input matrix are stored together in a checksum matrix, so they can be updated together.

B. Updating Checksums

In this section, we describe the details in updating checksums for POTF2, TRSM, SYRK and GEMM. Inspired by the checksum updating algorithm in outer product Cholesky decomposition [18], we conduct the checksum updating algorithms for inner product Cholesky decomposition as follow. As an example, we show the process of the third iteration of decomposition a $5\ blocks \times 5\ blocks$ matrix in Figure 3. Upper left gray areas represent decomposed slate area, which will not be updated or referenced in the future. In this iteration, area A(in red) and B(in blue) will be updated to area LA(in red) and LB(in blue) using area LC(in green) and LD(in yellow).

Figure 3. Example Matrix

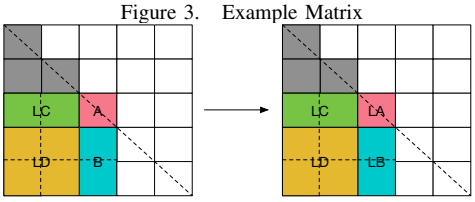


Figure 4. SYRK

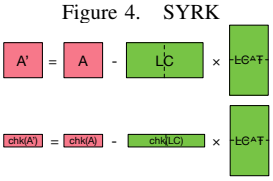


Figure 5. GEMM

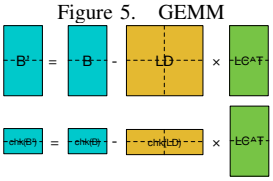
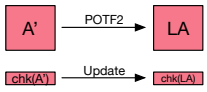


Figure 6. POTF2 and checksum updating



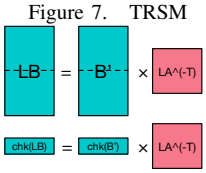
The first step, SYRK does rank-k update to the block on the diagonal (Area A), which will be decomposed in the current iteration. It can be described mathematically as:

$$A' = A - LC \times LC^T$$

The checksums of $C1$ can be updated as (Figure 4):

$$chk(A') = chk(A) - chk(LC) \times LC^T$$

Figure 7. TRSM



The next step is GEMM, which updates the panel (Area B) to be solved in the current iteration. It can be described mathematically as:

$$B' = B - LD \times LC^T$$

Obviously, the checksums update algorithm of GEMM should be (Figure 5):

$$chk(B') = chk(B) - chk(LD) \times LC^T$$

The third step, POTF2 operation is responsible for decompose a single block (Area A'). The checksum of the decomposed single block can be updated as:

Algorithm 2 checksums updating algorithm for POTF2

Require: factorized $n \times n$ lower triangular matrix LA with a column checksum chk

1: **for** $j = 1$ to N **do**
2: $chk[j] \leftarrow chk[j]/LA[j, j]$
3: $chk[j+1:n] \leftarrow chk[j+1:n] - chk[j] \cdot LA[j+1:n, j]$
4: **end for**

As shown in Figure 6, the checksum is updated to the checksums of LA after the execution of Algorithm 2. Then, the checksum is available to be used for detecting and correcting errors in LA.

The final step, TRSM solves linear triangular systems. It updates the panel sub-matrix B'using the result LA from POTF2, which can be described as:

$$LB = B' \times (LA^T)^{-1}$$

So, similarly, as shown in Figure 7 the checksums of B should be updated with:

$$chk(LB) = chk(B') \times (LA^T)^{-1}$$

C. Error Detection and Correction

Before each updating in Cholesky decomposition, verifying and correcting input data is necessary in our Enhanced Online-ABFT. We illustrate the process of error detecting, locating and correcting in a matrix block as follow. First, The matrix block to be detected is A and its corresponded two column checksums are:

$$chk_1 = [r_{11}, r_{12}, r_{13}...r_{1B}]$$

$$chk_2 = [r_{21}, r_{22}, r_{23}...r_{2B}]$$

Next, we recalculate the two column checksums for A:

$$chk'_1 = {v_1}^T A = [r'_{11}, r'_{12}, r'_{13}...r'_{1B}]$$

$$chk'_2 = {v_2}^T A = [r'_{21}, r'_{22}, r'_{23}...r'_{2B}]$$

Then, we compare chk_1 with chk'_1 to see whether they are close enough(within rounding error).

$$\delta_{1i} = r'_{1i} - r_{1i}$$

$$\delta_{2i} = r'_{2i} - r_{2i}$$

For instance, let's say we have found that $abs(\delta_{1i}) > e$, where e is the threshold of rounding error. So, an error is detected on the i^{th} column in the matrix block. By dividing $\delta_{2i}/\delta_{1i} = j$, we could get the row index j of the error and δ_{1i} give us the difference between the correct value and error value, so that we can correct it.

V. Optimizations

A. Optimization 1

We designed several innovative techniques aimed to minimize the overhead in our Enhanced Online-ABFT. Our optimization techniques mainly focused on three parts: checksums recalculation, checksums updating and the fault tolerance algorithm. In this section, we focus on minimizing the overhead brought by the checksums recalculation. In our Enhanced Online-ABFT, checksums recalculation is the key operation for data correctness verification. Optimizing its execution time is really important for several reasons: (1)It is on the critical path. For specific, it must be executed before each data correctness verification and the following original updating operation, so its execution cannot be overlapped with any one of them; (2)It is one of the few operations that bring majority overhead. However, the efficiency of executing checksums recalculation on GPU is low, since it consists several BLAS Level-2 vector-matrix multiplications. Moreover, due to the position of each block, these BLAS Level-2 operations cannot be merged into a more efficient BLAS Level-3 operation. To overcome this limitation, we designed an optimization technique that can significantly improve its efficiency. The key idea is that we allow several BLAS Level-2 checksums recalculation operations being executed on the GPU concurrently using CUDA concurrent kernel execution feature. CUDA allows each GPU executes multiple kernel functions concurrently [27] as long as two requirements are met:

1) Each kernel function must be assigned to a separated CUDA stream, which means each of them must not have any data dependency between each other;
2) There is enough GPU computation resources available for other kernel functions to execute.

The recalculation of each column checksums are independent form each other, so any number of them can be concurrently executed. For each GPU, there is a designed max number N of concurrent kernel execution, determined by its compute capability number. Moreover, depending on the resource usage of each kernel function and the total resources available on GPU, the max number of concurrent kernel execution in resources perspective could be different:

$$M = \frac{total\ resources\ on\ GPU}{Max\ resources\ usage\ of\ each\ kernel\ function}$$

So, the actual number of concurrent checksums recalculation is:

$$P = min(N, M)$$

In practice, the cuBLAS library used in MAGMA is not open sourced, so it is only possible to use profiling tools such

as nvprof to get the resources usage, however, it's still hard to accurately estimate the max number of concurrent execution given resources usage of each function. So, for simplicity, we just create N CUDA Streams to maximize the efficiency of checksums recalculation. Since all checksums recalculation operations are identical, we distribute them evenly among N CUDA streams.

B. Optimization 2

To further lower the overhead of our Enhanced Online-ABFT, we turn our focus on the checksum updating operations. Unlike checksums recalculation operations, checksum updating operations are not on the critical path. So, theoretically once the input data is verified, checksums updating can be executed concurrently with original updating operations and it can be assigned to both CPU or GPU as shown in Figure 2. If we put it on GPU as shown in Figure 2(a)(this concurrent relation is not shown), we can create a separate stream for it and use CUDA concurrent kernel execution feature. So it is possible that the execution can be partially/completely overlapped and the total overhead of it can be reduced. On the other hand, since CPU is idle most of the time in MAGMA's Cholesky decomposition, it is also possible to take the advantage of this and concurrently update checksums on CPU while GPU is performing other operations as shown in Figure 2(b), so the overhead can also be reduced. However, in this case, we need to ensure that CPU can complete its job close to the completion time of GPU. Otherwise, it may not be worth to do it on CPU.

To choose between CPU or GPU for checksums updating operation, we need to determine in which way we can gain minimum overhead. We designed an estimation model to help us make decision. First we define:

$$P_{GPU} = Peak\ performance\ of\ GPU(GFLOPS)$$

$$P_{CPU} = Peak\ performance\ of\ CPU(GFLOPS)$$

$$R = Data\ transfer\ rate\ between\ CPU\ and\ GPU$$

The number of FLOPS of the original MAGMA's Cholesky decomposition with $n \times n$ matrix input can be estimated as:

$$N_{Cho} = n^3/3$$

For checksums updating, the number of FLOPS is:

$$N_{Upd} = 2n^3/(3B)$$

In which, B is the block size. Moreover, the number of FLOPS for checksum recalculation is:

$$N_{Rec} = 2n^3/(3B)$$

Finally, if CPU is chosen for checksums updating, the extra data transfer overhead is:

$$D_{upd} = n^3/(3KB^2)$$

In which, K is number of iteration that we preform data correction verification once, will be explained in optimization 3. So, if we assign checksums updating on GPU, the estimated execution time is:

$$T_{Pick\ GPU} = \frac{N_{Cho} + N_{Upd} + N_{Rec}}{P_{GPU}}$$

If we let CPU concurrently do checksums updating, the estimated execution time is:

$$T_{Pick\ CPU} = max(\frac{N_{Cho} + N_{Rec}}{P_{GPU}}, \frac{N_{Upd}}{P_{CPU}} + \frac{D_{upd}}{R})$$

So, the decision depends on the peak performance of specific CPU and GPU and the data transfer rate between them.

C. Optimization 3

As our Enhanced Online-ABFT coveres more kinds of silent error than the Online-ABFT, it inevitablly brings more overhead. Let's look at the read/write pattern of MAGMA's Cholesky decomposition: Each block is updated $O(1)$ times and read $O(n)$ times on average. As a result, the Online-ABFT verifies each block $O(1)$ times on average and our Enhanced Online-ABFT verifies each block $O(n)$ times on average. Each block is verified more times, so it brings more overheads. The overhead of checksum encoding and checksums updating are still the same as in Online-ABFT. To lower the overhead in our Enhanced Online-ABFT, we focus on data verification process, since it is the only part that brings extra overhead. As shown in Table I, due to the extra number of verification in SYRK and GEMM, the overhead of our Enhanced Online-ABFT increases.

Table I
VERIFICATION COMPARISON

Operation	Online-ABFT		Enhanced Online-ABFT	
	verify	# of blocks	verify	# of blocks
POTF2	L	$O(1)$	A	$O(1)$
TRSM	B	$O(n)$	L, B	$O(n)$
SYRK	A	$O(1)$	A, C	$O(n)$
GEMM	B	$O(n)$	B, C, D	$O(n^2)$

We noticed that memory errors may not occur so frequently as our verification frequency. Verifying date correctness every iteration may over protect the data. So, inspired by the work [28], we designed an optimization, which allows Enhanced Online-ABFT to adjust the strength of protection. The basic idea is that instead of verifying input data every time in every iteration, now we only verify it once for every K iterations. Although both SYRK and GEMM bring more overhead, we can only apply this optimization only to GEMM and keep SYRK as same as before, since errors in the input of SYRK can propagate and cause unrecoverable situations or fail-stop failure if not corrected immediately. Moreover, it is also safe to apply this optimization for TRSM. There is a trade off between the overhead and the error correction capability. For systems with low error rate, we can increase K to lower the overhead. On the other hand, we need to keep K low for systems with high error rate. By properly adjusting the number K, we can achieve minimum overhead and still get enough error correction capability.

VI. OVERHEAD ANALYSIS

In this section, we analyze the overhead of our Enhanced Online-ABFT Cholesky decomposition and compare it with the overhead of Online-ABFT Cholesky decomposition. We show that our relative run-time and space overhead is similar to the overhead of Online-ABFT Cholesky decomposition. Table II defines the parameters we will use. The overhead of different steps of our fault tolerance algorithm are as follows:

Table II
DESCRIPTION OF EACH SYMBOL

Symbol	Description
n	input matrix size
B	matrix block size
K	Verify data every K iterations

1) Checksums Encoding overhead
This step is done before the Cholesky decomposition and it is the same for both Online-ABFT and Enhanced Online-ABFT. Each block in the input matrix is multiplied by the two checksum vectors to get the checksums. The number of floating operations can be calculated as:

$$O_{encode} = 1/2 \times 4 \times B^2 \times (n/B)^2 = 2n^2$$

The whole Cholesky decomposition takes $\frac{n^3}{3}$, so the relative overhead is $\frac{6}{n}$.
2) Checksums updating overhead
This step is done after each operation during the Cholesky decomposition, which is also same in both ABFTs. Checksum matrix is updated as same as the input matrix. The overhead of each operation in checksums updating is (Table III):

Table III
OVERHEAD OF CHECKSUM UPDATING

Operation	$O_{updating}$	Relative overhead
POTF2	$2Bn$	$\frac{6B}{n^2}$
TRSM	$2n^2$	$\frac{6}{n}$
SYRK	$2n^2$	$\frac{6}{n}$
GEMM	$\frac{2}{3B}n^3$	$\frac{2}{B}$

Since POTF2 bring little overhead, it can be ignored here. So the total updating relative overhead is: $\frac{12}{n}+\frac{2}{B}$.
3) Checksums recalculating overhead
   a) Online-ABFT
   This step is done after each operation in Cholesky decomposition. The checksums are recalculated after each block is updated. The overhead of each step can be calculated as in Table IV). Both the overhead of POTF2 and SYRK can be ignored here, so the total relative recalculation overhead is: $\frac{12}{n}$.

Table IV
OVERHEAD OF CHECKSUMS RECALCULATION OF ONLINE-ABFT

Operation	O_{recal_online}	Relative overhead
POTF2	$4Bn$	$\frac{12B}{n^2}$
TRSM	$2n^2$	$\frac{6}{n}$
SYRK	$4Bn$	$\frac{12B}{n^2}$
GEMM	$2n^2$	$\frac{6}{n}$

   b) Enhanced Online-ABFT
   This step is done before each operation in Cholesky decomposition. The checksums are recalculated for blocks that will be read or updated. The overhead of each step can be calculated in Table V. After ignoring the minor overhead brings by the POTF2, the total relative recalculation overhead is: $\frac{6K+6}{nK} + \frac{2}{BK}$

Table V
OVERHEAD OF CHECKSUMS RECALCULATION OF ENHANCED ONLINE-ABFT

Operation	$O_{recal\_enhanced}$	Relative overhead
POTF2	$4Bn$	$\frac{12B}{n^2}$
TRSM	$2n^2$	$\frac{6}{n}$
SYRK	$\frac{2n^2}{K}$	$\frac{6}{nK}$
GEMM	$\frac{2n^3}{3BK}$	$\frac{2}{BK}$

4) Checksums verification overhead
It step is used for verify the correctness of each operation in Cholesky decomposition. It only brings slight overhead, thus it can be ignored here.
5) Space overhead
For both ABFTs, checksums are stored in a checksum matrix, of which the size is: $\frac{2}{B}n^2$ and relative space overhead is: $\frac{2}{B}$.
6) Data Transfer overhead
If we choose to update checksums on GPU, it only brings slight data transfer overhead, which can be ignored here. If we choose to update checksums on CPU, there also involves some data transfer overhead:
   a) Initial checksums transfer: $\frac{2n^2}{B}$;
   b) Checksum updating related transfer: $\frac{n^2}{2}$;
   c) Verification related transfer:
      i) Online-ABFT: $\frac{n^2}{2B}$;
      ii) Enhanced Online-ABFT: $\frac{n^3}{3KB^2}$.
7) Summary
The overall relative overhead is shown in Table VI.

Table VI
OVERALL OVERHEAD

	Overall Relative overhead	$n \to \infty$
Online-ABFT	$\frac{30}{n} + \frac{2}{B}$	$\frac{2}{B}$
Enhanced Online-ABFT	$\frac{24K+6}{nK} + \frac{2K+2}{BK}$	$\frac{2K+2}{BK}$

B is determined by the performance of hardware and MAGMA's implementation. So, we can see with fixed B, if the matrix size is close to the block size, it will affect the relative overhead. In that case, the relative overhead will decrease with the increasing of the input matrix size. When the input matrix size is relatively large, the relative overheads of both ABFTs will continue decrease and converging to small constant. So, The Enhanced Online-ABFT Cholesky decomposition should behave similar to the original MAGMA Cholesky decomposition and current state-of-the-art Online-ABFT Cholesky decomposition with slightly lower efficiency.

VII. EXPERIMENTAL EVALUATIONS

A. Experiments Environments

Our Enhanced Online-ABFT Cholesky decomposition is built based the latest MAGMA version 1.6.2. It is linked with the cuBLAS 7.0 [29] on GPU and ACML 5.3.0 [30] on the CPU. We implemented the double precision version Cholesky decomposition. The interface of the routine is not changed. For best performance, all checksums-related operations are also implemented with ACML-equivalent and cuBLAS-equivalent subroutines in MAGMA.

Table VII
FAULT TOLERANCE CAPABILITY COMPARISON ON TARDIS WITH 20480×20480 CHOLESKY DECOMPOSITION

	No Error	Computation Error	Memory Error
Enhanced Online-ABFT	10.6572s	10.6614s	10.6678s
Online-ABFT	10.5067s	10.5244s	22.625s
Offline-ABFT	10.4489s	21.3942s	21.2631s

Table VIII
FAULT TOLERANCE CAPABILITY COMPARISON ON BULLDOZER64 WITH 30720×30720 CHOLESKY DECOMPOSITION

	No Error	Computation Error	Memory Error
Enhanced Online-ABFT	8.84598s	8.9253s	8.91492s
Online-ABFT	8.64649s	8.69622s	21.4162s
Offline-ABFT	8.64265s	21.4472s	21.3511s

We evaluated our implementation on two heterogeneous systems: TARDIS and BULLDOZER64. Tardis is a cluster system with 4 GPU nodes. The GPU node is equipped with two 16 cores 2.1GHz AMD 6272 Opteron Processors with 64GB DRAM and a NVIDIA Tesla M2075 GPU with 6GB memory. The micro-architecture of the GPU is Fermi. Bulldozer64 is a heterogeneous system equipped with four 16 cores 2.1GHz AMD 6272 Opteron Processors with 64GB DRAM and a NVIDIA Tesla K40c GPU with 12GB memory. The micro-architecture of the GPU is Kepler.

We tested our implementations with several different input matrix sizes from the largest our GPU memory allows to relatively small sizes. For Tardis system, the test is from 5120×5120 to 23040×23040. For Bulldozer64 system, the test is from 5120×5120 to 30720×30720. As for the block size, MAGMA chooses different block sizes for different GPUs. For Fermi GPU, the default block size is 256×256 and for Kepler GPU, it uses larger block size 512×512.

B. Fault tolerance capability comparison

This subsection compares the fault tolerance capability of our Enhanced Online-ABFT with Offline-ABFT and Online-ABFT by injecting different type of errors. As we can see in Table VII and VIII, all three ABFTs have similar execution time when there is no error. When a computating error is injected, it soon propagates to other areas and cause unrecoverable situation for Offline-ABFT. So it needs to repeat the whole decomposition again, which doubles its execution time. Since Online-ABFT can correct computating error in a time manner, its execution is not affected. When we injected a storage error between checksum verification and data access, both Offline-ABFT and Online-ABFT cannot correct it, so they need to re-do the decomposition again, which cost twice of the time. However, our Enhanced Online-ABFT can correct both types of errors without affecting execution time.

C. Optimization 1

We show the result of our first optimization technique. In this optimization, we use the CUDA concurrent kernel execution feature to let several checksums recalculations execute concurrently on GPU. We show the relative overhead of our Enhanced Online-ABFT before and after we apply this optimization.As we can see in Figure 8 and 9, blue line and red line represents the relative overhead before and after

Figure 8. Optimization 1 on Tardis

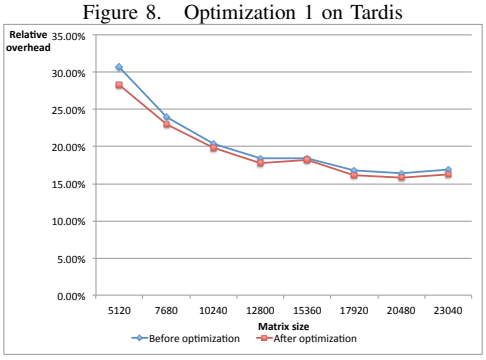


Figure 9. Optimization 1 on Bulldozer64

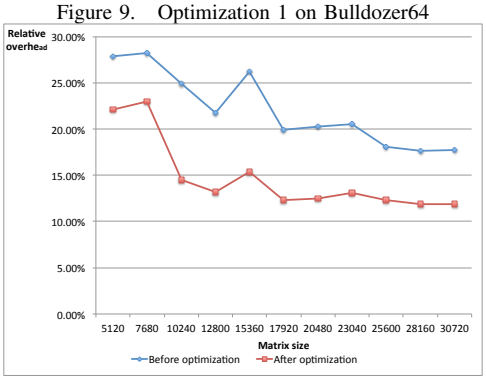


Figure 10. Optimization 2 on Tardis

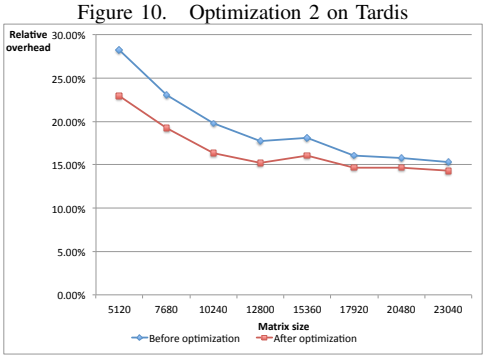


Figure 11. Optimization 2 on Bulldozer64

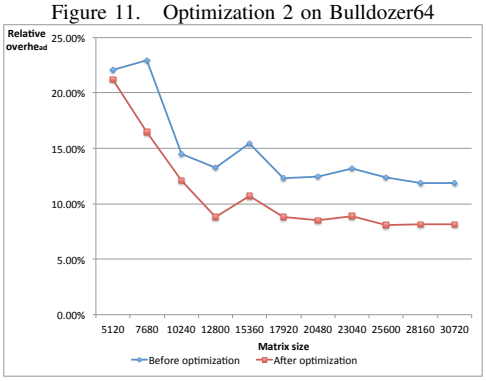


Figure 12. Optimization 3 on Tardis

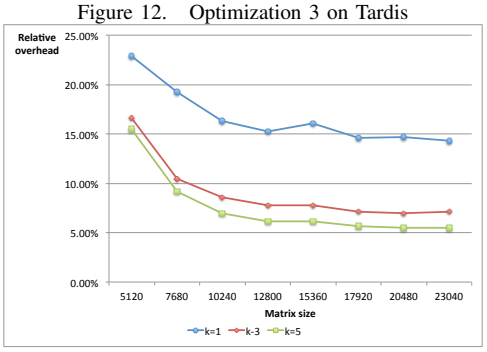


Figure 13. Optimization 3 on Bulldozer64

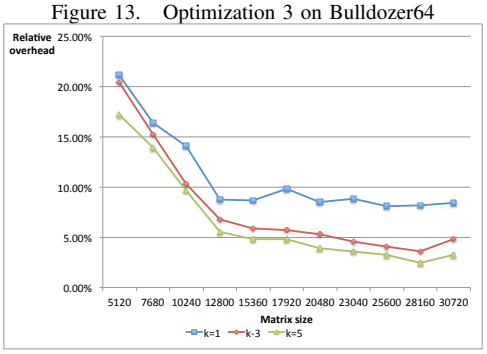


Figure 14. Relative Overhead on Tardis

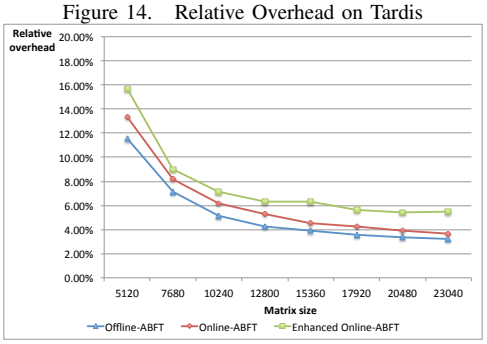


Figure 15. Relative Overhead on Bulldozer64

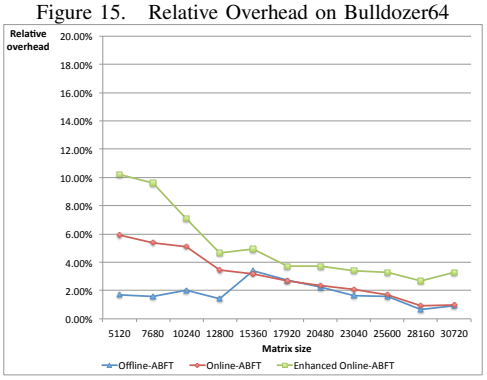


Figure 16. Performance on Tardis

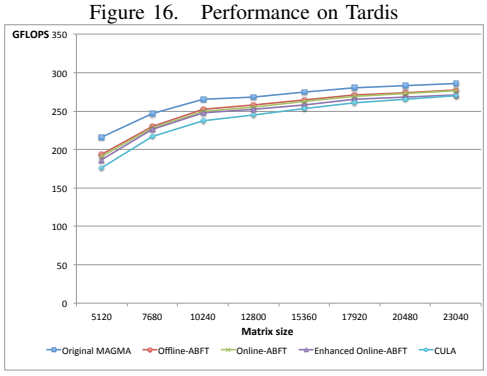
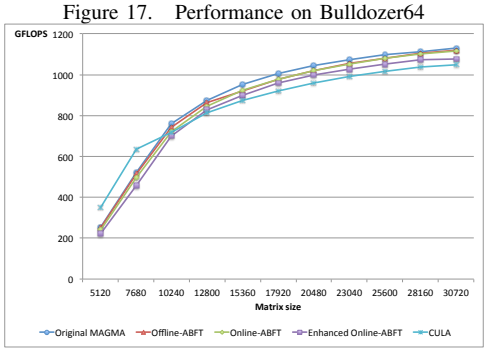


Figure 17. Performance on Bulldozer64



we apply this optimization. As we can see, optimization 1 reduces the relative overhead by about 2% on Tardis and about 10% on Bulldozer64. Note that the relative overhead is reduced a lot more on Bulldozer64. This is because Bulldozer64 is equipped with more powerful and advanced GPU, which could allow more checksum recalculations to be executed together, so it has much more efficiency.

D. Optimization 2

In this section, we show our test result on applying our second optimization technique, which aims to let CPU or a separate GPU CUDA stream concurrently do checksums updating with original updating and checksums recalculations. Determined by our testing system, we choose CPU to update checksums on Tardis system and choose GPU to update checksums on Bulldozer64 system. As shown in Figure 10 and 11, our optimization 2 reduces the relative overhead by about 5% on Tardis on average and about 8% on Bulldozer64 on average.

E. Optimization 3

In this section, we show the benefit brings by the our third optimization technique. This optimization aims to adjust the frequency of checksum verification in our Enhanced Online-ABFT. We only let our Enhanced Online-ABFT verify data correctness for every K iteration. We show the overhead change as we adjust K to be 1, 3, and 5. As we can see in Figure 12 and 13, the relative overhead of our Enhanced Online-ABFT has reduced significantly as we adjust K.

F. Overhead comparison

In this section, we compared the relative overhead between Offline-ABFT Cholesky decomposition, Online-

ABFT Cholesky decomposition, and our Enhanced Online-ABFT Cholesky decomposition. As shown in Figure 14 and 15, the overhead of our Enhanced Online-ABFT is close to constant when the matrix size is large. our Enhanced Online-ABFT only introduced less than 6% overhead on Tardis and less than 4% overhead on Bulldozer. It is only slightly higher than Offline-ABFT and Online-ABFT.

G. Performance comparison

Figure 16 and 17 compare the performance of the Original MAGMA's Cholesky decomposition, CULA's Cholesky decomposition, Offline-ABFT Cholesky decomposition, Online-ABFT Cholesky decomposition, and our Enhanced Online-ABFT Cholesky decomposition. Figure 16 and 17 indicate that the performance of Enhanced Online-ABFT is comparable to Offline-ABFT and Online-ABFT. Also, even with both computation error and memory error tolerance capability, our Enhanced Online-ABFT is still faster than CULA on both systems.

VIII. Conclusion

This paper presented a new ABFT scheme for Cholesky decomposition that can correct both computing and storage errors. Several optimization techniques were also developed to reduce the fault tolerance overhead. Experimental results demonstrate that our fault tolerant Cholesky decomposition can achieve better performance than the state-of-the-art Cholesky decomposition routine in CULA R18.

References

[1] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, 2010, pp. 691–696.

[2] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 226–235.

[3] S. Di, E. Berrocal, and F. Cappello, "An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 271–280.

[4] A. Bouteiller, "Fault-tolerant mpi," in *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015, pp. 145–228.

[5] J. Dongarra, T. Herault, and Y. Robert, "Fault tolerance techniques for high-performance computing," 2015.

[6] A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy," *ACM Transactions on Parallel Computing*, vol. 1, no. 2, p. 10, 2015.

[7] A. Katti and G. Di Fatta, "Epidemic fault tolerance for extreme-scale parallel computing," in *Internet and Distributed Computing Systems*. Springer, 2015, pp. 201–208.

[8] D. Hakkarinen, P. Wu, and Z. Chen, "Fail-stop failure algorithm-based fault tolerance for cholesky decomposition," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 5, pp. 1323–1335, 2015.

[9] H. Härtig, S. Matsuoka, F. Mueller, and A. Reinefeld, "Resilience in exascale computing (dagstuhl seminar 14402)," 2015.

[10] D. Hakkarinen and Z. Chen, "Algorithmic cholesky factorization fault recovery," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[11] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra, "Composing resilience techniques: Abft, periodic and incremental checkpointing," *International Journal of Networking and Computing*, vol. 5, no. 1, pp. 2–25, 2015.

[12] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 225–234, 2012.

[13] K.-H. Huang, J. Abraham *et al.*, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 1984.

[14] J. Gunnels, D. S. Katz, E. S. Quintana-Orti, R. Van de Gejin *et al.*, "Fault-tolerant high-performance matrix multiplication: Theory and practice," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 2001, pp. 47–56.

[15] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 19, no. 12, pp. 1628–1641, 2008.

[16] Z. Chen, "Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 167–176.

[17] P. Wu, C. Ding, L. Chen, T. Davies, C. Karlsson, and Z. Chen, "On-line soft error correction in matrix–matrix multiplication," *Journal of Computational Science*, vol. 4, no. 6, pp. 465–472, 2013.

[18] P. Wu and Z. Chen, "Ft-scalapack: Correcting soft errors online for scalapack cholesky, qr, and lu factorization routines," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 49–60.

[19] T. Davies and Z. Chen, "Correcting soft errors online in lu factorization," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 2013, pp. 167–178.

[20] "Cula." [Online]. Available: http://www.culatools.com/

[21] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen, "Matrix multiplication on gpus with on-line fault tolerance," in *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*. IEEE, 2011, pp. 311–317.

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

[23] ——, "High performance dense linear system solver with resilience to multiple soft errors," *Procedia Computer Science*, vol. 9, pp. 216–225, 2012.

[24] Y. Jia, P. Luszczek, G. Bosilca, and J. J. Dongarra, "Cpu-gpu hybrid bidiagonal reduction with soft error resilience," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2013, p. 2.

[25] A. Scholl, C. Braun, M. A. Kochte, and H.-J. Wunderlich, "Efficient on-line fault-tolerance for the preconditioned conjugate gradient method," in *On-Line Testing Symposium (IOLTS), 2015 IEEE 21st International*. IEEE, 2015, pp. 95–100.

[26] E. Yao, J. Zhang, M. Chen, G. Tan, and N. Sun, "Detection of soft errors in lu decomposition with partial pivoting using algorithm-based fault tolerance," *International Journal of High Performance Computing Applications*, p. 1094342015578487, 2015.

[27] "Streams and concurrency webinar." [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

[28] L. Chen, D. Tao, P. Wu, and Z. Chen, "Extending checksum-based abft to tolerate soft errors online in iterative methods."

[29] "Cublas." [Online]. Available: https://developer.nvidia.com/cuBLAS

[30] "Acml." [Online]. Available: developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/