Fault Tolerant One-sided Matrix Decompositions on Heterogeneous Systems with GPUs

Jieyang Chen*, Hongbo Li*, Sihuan Li*, Xin Liang*, Panruo Wu†,
Dingwen Tao◇, Kaiming Ouyang*, Yuanlai Liu*, Kai Zhao*, Qiang Guan‡, and Zizhong Chen*
*University of California, Riverside †University of Houston ◇University of Alabama ‡ Kent State University
{jchen098, hli035, sli049, xlian007, kouya001, yliu158, kzhao016, chen}@cs.ucr.edu,
tao@cs.ua.edu pwu7@uh.edu qguan@kent.edu

***Abstract*—Current algorithm-based fault tolerance (ABFT) approach for one-sided matrix decomposition on heterogeneous systems with GPUs have following limitations: (1) they do not provide sufficient protection as most of them only maintain checksum in one dimension; (2) their checking scheme is not efficient due to redundant checksum verifications; (3) they fail to protect PCIe communication; and (4) the checksum calculation based on a special type of matrix multiplication is far from efficient. By overcoming the above limitations, we design an efficient ABFT approach providing stronger protection for one-sided matrix decomposition methods on heterogeneous systems. First, we provide full matrix protection by using checksums in two dimensions. Second, our checking scheme is more efficient by prioritizing the checksum verification according to the sensitivity of matrix operations to soft errors. Third, we protect PCIe communication by reordering checksum verifications and decomposition steps. Fourth, we accelerate the checksum calculation by 1.7x via better utilizing GPUs.**

***Index Terms*—Algorithm-based fault tolerance, Linear algebra, Matrix decomposition, GPU, Heterogeneous system**

I. Introduction

A 100% reliable computing system is non-existent. Hardware faults can occur in almost every component of a computing system due to erroneous bit flips. When a fault strikes a critical hardware component at a critical timepoint, the fault occurs as an error such as CPU logic error, data corruption, disk crash, and network error [1]–[8]. These errors can either cause the abnormal termination of the computing process or lead to incorrect results — an error causing the former is known as fail-stop error and one causing the latter is known as *soft error* or fail-continue error. Soft errors are far more challenging to be dealt with than fail-stop errors [9] as soft errors' occurrences are hard to detect at runtime [10]–[14]. In this paper, we restrict our scope to handling faults that cause soft errors in computing systems.

Heterogeneous computing systems with GPUs are no exception — soft errors also occur in GPUs. Recent research has shown that GPUs are also very susceptible to soft errors [15]–[21] and soft error rate increases significantly as the GPU workload increases [22]. Also energy saving approaches for GPUs based on undervolting and overclocking greatly disrupt GPU's stability and thus incur frequent soft errors [23]–[25]. Though GPUs accelerate the scientific discoveries by delivering great performance, the existence of soft-errors significantly weakens the credibility of these discoveries.

One-sided matrix decomposition methods like LU, Cholesky, and QR play a pivotal role in many scientific applications. Much work has been done to develop highly optimized one-sided matrix decompositions on heterogeneous systems with GPUs [26]–[29]. However, a single error can invalidate a large portion of values in the resulting matrix due to the error propagation property in the process of matrix decomposition [13], [30]–[32]. Matrix decomposition's incorrect results resulting from soft errors greatly undermine the validity of applications depending on them.

Triple Modular Redundancy (TMR) [33] is a general approach to tolerate soft errors. TMR works in following way: it first either performs three identical computations with each on one hardware platform at the same time or performs the computation for three times on the same hardware, then compares the three results obtained, and finally reports the assumed correct result based on majority voting. Though it is a general approach that can be applied to any application, it introduces very high overhead (i.e., 200%).

To avoid such significant overhead, the algorithm-based fault tolerance (ABFT) technique was first proposed by Huang and Abraham [34], whichtolerate soft errors for matrix operations with far less overhead. Huang and Abraham proved that for many matrix operations the relationship between input matrix and its checksum holds in the final computation results, which can be used for error detection and correction in the end of computation. Suppose an n-by-n matrix is given. Its decomposition complexity is $\mathcal{O}(n^3)$, the error detection complexity is only $\mathcal{O}(n^2)$, and the overhead of error recovery is far less than that of TMR considering the recovery does not require re-execution. Due to this good property, much effort has been spent on using ABFT to protect one-sided matrix decompositions [11]–[13], [30]–[32].

A. Limitation of current works

Though tremendous progress has been made, existing ABFT approaches for one-sided matrix decomposition on heterogeneous systems with GPUs have following limitations:

1) **Insufficient protection.** Most current ABFT one-sided matrix decompositions [11], [12], [31], [32] only maintain *single-side* checksum protection, i.e., maintain checksum either by row or column, and thus they only protect a part of the matrix. Though *full* checksum protection based on both row and column checksums can provide better protection, it is only applied to LU decomposition and limited to CPU [13]. The study of full checksum protection for other matrix decomposition methods on GPU platform is non-existent. In addition, one single soft error in the GPU's memory system (DRAM and on-chip memory) caused by multiple bit-flips can propagate along a row (column) during major computations of matrix decompositions and thus corrupt the row (column), but it cannot be protected by either NVIDIA GPU's default Error-Correcting Code (ECC) [35]–[39] or current ABFT approaches.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

2) **Inefficient checking scheme.** ABFT checking scheme determines when to perform correctness check. It plays a pivotal role in determining the ABFT checking overhead. Using traditional ABFT checking scheme designed for single-side checksum [11], [12], [31], [32], full checksum based ABFT incurs unnecessary protection overhead due to redundant correctness check.
3) **Lack of PCIe communication protection.** PCIe is one the most important uncore component in heterogeneous systems with GPUs. Matrix decompositions heavily rely on it to transfer large-sized sub-matrices between CPU and GPU or inter-GPU. Soft errors can also affect PCIe and thus disrupt communication [1], [40], [41]. However, none of the previous ABFT approaches protect PCIe communication.
4) **Inefficient checksum calculation on GPU.** Checksum calculation [11], [12] requires the multiplication of a regular-sized matrix and a tall-and-skinny matrix based on an underlying linear algebra library [42]. However, the underlying library is very inefficient for the above type of matrix multiplication as GPU is significantly underutilized in this case.

B. Our contribution

By overcoming the above limitations, we design an efficient ABFT approach to provide stronger protection for three major one-sided matrix decomposition methods including Cholesky, LU, and QR on heterogeneous systems with GPUs.

1) **Full matrix protection.** We prove that full checksum protection is also applicable for Cholesky and QR decomposition. Based on full checksum protection, we are able to provide full matrix protection for all three core one-sided matrix decomposition methods except for a trivial step of QR that computes triangular factor. In addition, since the full checksum encodes the matrix in two dimensions, the protection comes along with the benefit of tolerating errors that accumulate along one row or column, which is usually caused by GPU memory error during matrix decompositions.
2) **Efficient checking scheme.** We study the error propagation pattern caused by computation, memory system, and communication error that occurs in all major operations of matrix decompositions. It helps us tell the sensitivity of a matrix operation to soft errors. We provide an efficient ABFT checking scheme by prioritizing the checksum verification according to the sensitivity of matrix operations, i.e., performing more verifications on more sensitive operations and less verifications on less sensitive ones.
3) **Protection for PCIe communication.** By carefully reordering checksum verification, communication, and computation, our new ABFT checking scheme can protect soft errors that occur in the communication over PCIe. It brings negligible overhead in error-free executions and less than 1% recovery overhead when error occurs.
4) **Optimized kernel for ABFT on GPU:** Based on the characteristics of its calculation and GPU architecture, we design an innovative highly optimized checksum encoding kernel on GPUs. Experiments show that our optimized kernel improves performance of checksum calculation by 1.7x on average and up to 1.9x compared with the existing best works [11], [12].

II. Related work

ABFT was first proposed by Abraham and Huang [34] for LU decomposition. This approach is very effective to detect any number of errors in the computation, but it is only able to correct up to one error in the decomposed matrices based on the checksum invariant in the end of computation. However, this recovery approach is not effective in practice as usually one error lead to amounts of errors due to the error propagation property of matrix operations [13], [30].

Later works proposed online ABFT that can detect errors during computation when they are not propagated far away and makes error correction much easier. In [43], Wu et al. proposed an online ABFT for matrix multiplication. In [32], Davies proposed an online ABFT for LU decomposition. In [31], Wu et al. extended online ABFT to three core one-sided matrix decompositions - Cholesky, LU, and QR on distributed memory computing systems. In [11], [12], Chen et al. proposed an online ABFT approach to protect memory error in additional to computation error in Cholesky decomposition on GPUs. Wu et al. [13] proposed a full checksum-based LU decomposition to handle memory error in CPU cache and registers.

TABLE I: Notation in Algorithms and Formulations.

$c(A)$	Column checksum(s) of matrix/matrix block A.
$r(A)$	Row checksum(s) of matrix/matrix block A.
$recal\_c(A)$	Recalculated column checksum(s) of matrix/matrix block A.
$recal\_r(A)$	Recalculated row checksum(s) of matrix/matrix block A.
A^c	Matrix/matrix block A *with* its column checksum(s).
A^r	Matrix/matrix block A *with* its row checksum(s).
A^f	Matrix/matrix block A *with* its full checksum(s).
n	Input matrix size ($n \times n$).
NB	Matrix blocks size ($NB \times NB$).
K	Number of DRAM/on-chip memory error(s) that cause 1D error propagation during TMU.

III. Backgrounds

A. Blocked matrix decomposition

Efficient one-sided matrix decomposition algorithms commonly follow blocked fashion as it delivers better performance. During the decomposition, the input matrix is divided logically into matrix blocks. Such matrix block is a basic unit in the decomposition process. one or multiple blocks can form a panel and a trailing matrix. The decomposition is an iterative process of *update operations*. In each update operation, sub-matrices composed of a part of the matrix blocks are used to update a sub-matrix composed of some blocks. *Update part* is a sub-matrix that gets updated during the update operation. *Reference part* is a sub-matrix that only gets referenced during the update operation.

Matrix decompositions share three similar major update operations in one iteration: (1) *Panel decomposition* (PD), (2) *Panel update* (PU), and (3) *Trailing matrix update* (TMU). The decomposition starts from the top left corner of the input matrix and iteratively works towards bottom right corner until done. **Fig. 1** shows one iteration of LU decomposition. In this iteration: first, column panel $A_{\cdot 1}$ is decomposed into $L_{\cdot 1}$ and U_{11}; then, row panel A_{12} is updated into U_{12}; finally, trailing matrix A_{22} is updated into A'_{22}. Due to data dependencies, these three steps have to be done in order. In implementations on modern heterogeneous systems with GPUs, e.g., the state-of-the-art MAGMA library [44]–[46]. these three steps are assigned to different computation units based on their specialties. PD follows irregular computation pattern, so it is assigned to CPUs. PU and TMU are highly parallelizable, so they are assigned to GPUs.

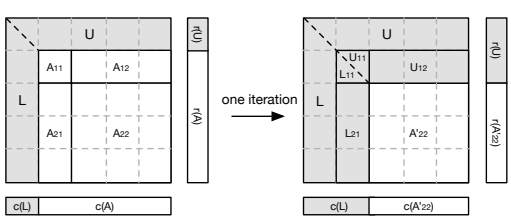


Fig. 1: Full checksum LU decomposition.

B. Checksum error detection and correction

ABFT is based on the idea that if we encode the input matrix with checksum, and perform a checksum maintaining algorithm along with the matrix operation, the relationship between checksum and the input matrix will still hold for resulting matrix, which can be used for error detection and correction. The key difference between online and offline ABFT is that online ABFT can maintain checksum relation during matrix decomposition (i.e., after each update operation). Offline ABFT, on the other hand, can only maintain checksum relation in the end of decomposition. In this subsection, we show how checksums are used in online ABFT [11], [32], [47]. We also adopt the similar general mechanism in this work.

Before the matrix decomposition, we first encode the input matrix with checksums. The checksum is the sum of matrix elements along either rows or columns. So the checksum can be used for error detection by verifying this relationship. To correct errors, the first step is to get error location and magnitude. The first step in turn requires two checksums encoded by two different checksum weights must be used. A usual choice of the two checksum weights are: $v_1 = [1, 1, 1...1]^T$ and $v_2 = [1, 2, 3...n]^T$. In practice [11], [12], each matrix block, not the whole input matrix, is usually used as a unit for checksum encoding, error detection and correction on heterogeneous systems with GPUs, since this fine-grained checksum encoding can be easily integrated with the original heterogeneous GPU version matrix decomposition implementations and it can significantly strengthen the fault tolerance protection density. For matrix block A, the column and row checksums are calculated as: $c(A) = \begin{bmatrix} {v_1}^T \\ {v_2}^T \end{bmatrix} \cdot A$ and $r(A) = A \cdot [v_1 v_2]$.

During matrix decompositions, using properly designed checksum maintaining algorithms, we can update checksums along with each update operation during the matrix decomposition, so that checksum relation is maintained after each update operation and we can use that to detect and correct errors on-line.

Upon error detection and correction, we check if the checksum relation still holds by calculating the checksums again on each relevant matrix block (i.e., $recal\_c(A)$ and $recal\_r(A)$) and comparing them with the checksums we maintained. We use the column checksum verification as an example here (row checksum verification is similar): we compare $recal\_c(A)$ with $c(A)$ to see whether they are close enough (within round-off error) by calculating: $\delta = c(A) - recal\_c(A)$. For instance, if we find that $|\delta_{1,i}| > e_c$, where e_c is the round-off error bound of column checksums, then an error is detected on the i^{th} column of the matrix block. By calculating $round(\delta_{2,i}/\delta_{1,i}) = j$ (round to the nearest integer), we get the row index j of the error and $\delta_{1,i}$ gives us the difference between the correct value and corrupted value. With both row and column index of the error and the magnitude of the error, we can correct the error.

Due to round-off error, the maintained checksums usually do not precisely match with corresponding matrix blocks even if no error occurs. To distinguish checksum mismatch caused by error or round-off error, we need to quantify to what degree a round-off error can develop (i.e., bound). For example, for full checksum protected TMU ($C^f \leftarrow C^f - A^c \times B^r$), based on [48], round-off error bound for column and row checksum can be derived from priori norm based error bound as follows:

$$e_c = |c(C) - recal\_c(C)| \leqslant \gamma_n \left\|A^c\right\|_1 \left\|B^r\right\|_1$$

$$e_r = |r(C) - recal\_r(C)| \leqslant \gamma_n \left\|A^c\right\|_\infty \left\|B^r\right\|_\infty$$

In the above equations, $\gamma_n = nu/(1 - nu)$, in which u is the unit round-off error (in IEEE 754 double bit floating point standard, $u \approx 10^{-16}$).

C. Full checksum LU decomposition

One of the most challenging part of designing online ABFT is maintaining checksums during matrix decompositions. Previous works were only able to maintain single-side checksums (i.e., either row or column checksum) during matrix decompositions [11], [12], [31], [32]. They usually only protect a part of the matrix and cannot tolerate errors that accumulate along one row or column, which is usually caused by memory error during matrix decompositions.

Recently, [13] made an improvement to online ABFT that can maintain full checksum for LU. It works as follows: (1) Before the decomposition, the input matrix is first encoded with full checksum; (2) During the decomposition, full checksum is maintained for trailing matrix, and single-side checksum is maintained for all panels; (3) In the end of decomposition, all decomposed matrices are protected by either column or row checksum. For example, **Fig. 1** shows one iteration of full checksum LU. Before this iteration, undecomposed part, trailing matrix $A_{..}$ (white part), has full checksum encoded. After the iteration, single-side checksum is maintained and extended for partially decomposed matrix (gray part). Full checksum is maintained for the new smaller trailing matrix, which will be used for the next iteration (submatrix A'_{22}).

1) $A^f_{:1} \rightarrow L^c_{:1} \times U^r_{11}$
2) $A^r_{12} \rightarrow L_{11} \times U^r_{12}$
3) $A'^f_{22} \leftarrow A^f_{22} - L^c_{21} \times U^r_{12}$

The full checksum brings two major benefits: **wider protection coverage** and **stronger protection**. Wider protection coverage means all parts of the matrix in LU are protected by checksums. Stronger protection means it can tolerate an erroneous row or column checksum in matrix [13], whereas single-side checksum can only tolerate one error at a time, since full checksum encode matrix on both matrix dimensions, which record more redundant information than single-side checksum [11], [31], [32]. In LU, full checksum is maintained for the trailing matrix, which is used in the majority computation (i.e., TMU) of one-sided matrix decompositions. So, it greatly strengthens the protection to LU. If we can maintain full checksum for TMU in other one-sided matrix decompositions, we can also provide wider and stronger protection for them. However, it is still unclear whether it can also be applied to other one-sided matrix decompositions, since maintain full checksum is non-trivial.

IV. Full checksum for Cholesky and QR

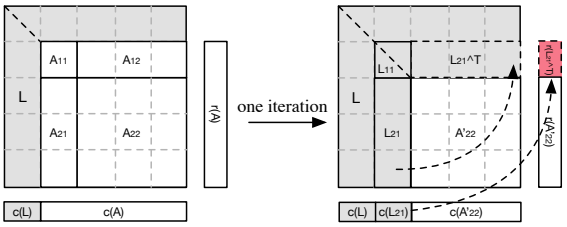
In previous works [11], [12], [31], [32], the common way to maintain checksums during matrix decompositions is updating the checksums during decompositions as if the checksums are an extended part of the original input matrix. That's to say, we update checksums by applying the same operation as the corresponding update operation. In this way, the checksum relation is naturally preserved during error-free executions. However, due to the characteristic of one-sided matrix decompositions, usually only single-side checksums were able to be maintained in this way. Maintaining full checksum, on the other hand, is challenging. In this work, we develop full checksum maintaining algorithm for Cholesky and QR decomposition by leveraging the algorithmic knowledge and developing deep-customized update operations for checksums that are not naturally preserved. Note that although in this paper we focus on implementations on heterogeneous systems

with GPUs, our full checksum for matrix decompositions can actually be applied to any computing systems. The design details are discussed as follows.

TABLE II: Single and full checksum Cholesky decomposition.

	Single-side Checksum	Full Checksum
PD	$L^c_{11} \leftarrow A^c_{11}$	$L^c_{11} \leftarrow A^c_{11}$
PU	$L^c_{21} \leftarrow A^c_{21} \times L^{-1}_{11}$	$L^c_{21} \leftarrow A^c_{21} \times L^{-1}_{11}$
TMU	$A'^c_{22} \leftarrow A^c_{22} - L^c_{21} \times (L_{21}{}^T)$	$A'^f_{22} \leftarrow A^f_{22} - L^c_{21} \times (L_{21}{}^T)^r$

*$A_{11/21}/L_{11/21}$ is panel before/after current iteration. A_{22}/A'_{22} is trailing matrix before/after current iteration.



Before current iteration, column checksum c(L) is maintained for partially decomposed part L and full checksum is maintained for trailing matrix A.

During TMU, column panel L21 is logically transposed into row panel together with it checksum c(L21), so that full checksum can be maintained for the new trailing matrix A'22.

Fig. 2: Full checksum Cholesky decomposition.

A. Full checksum for Cholesky decomposition

In Cholesky, similar to LU, there are three major steps in each iteration: PD, PU, and TMU. In existing ABFT approaches [11], [12], [31], only single-side checksum (column checksums for lower triangular Cholesky decomposition or row checksums for upper triangular Cholesky decomposition) is maintained for Cholesky as shown in the second column of **Table II**. Since Cholesky only decomposes half of the matrix (upper or lower triangular), it does not naturally preserve checksums for the other dimension.

To maintain full checksum for Cholesky, we need to modify the TMU. Using lower triangular Cholesky as an example, as shown in **Fig. 2**, since the input matrix is symmetrical, column panel L_{21} also serves (logically transposed) as row panel L^T_{21} during TMU. Also, column checksum of column panel $c(L_{21})$ is also the row checksum $r(L^T_{21})$ when it is logically transposed to row panel. As proved in [34], if we encode one matrix with column checksums and the other matrix with row checksums, the resulting multiplied matrix will have full checksum encoded and the basic computation of TMU is matrix-matrix multiplication. So, by transposing the column checksum of column panel to get row checksum, we can maintain full checksum for TMU as shown in right part of **Fig. 2**. We derive the equations for maintaining full checksum for Cholesky decomposition in the third column of **Table II** (red symbols show the modifications).

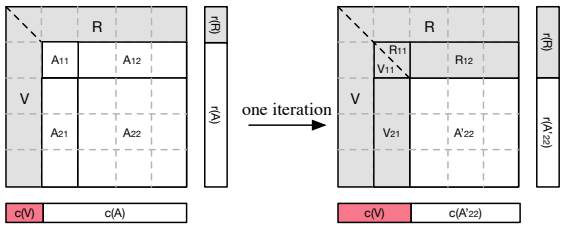
B. Full checksum for QR decomposition

TABLE III: Single and full checksum QR decomposition.

	Single-side Checksum	Full Checksum
PD	$V_{\cdot 1}$& $R^r_{11} \leftarrow A^r_{\cdot 1}$	$V^c_{\cdot 1}$ & $R^r_{11} \leftarrow A^f_{\cdot 1}$
CTF	$T \leftarrow V_{\cdot 1}$	$T \leftarrow V_{\cdot 1}$
TMU	$A'^r_{\cdot 2} \leftarrow A^r_{\cdot 2} - V_{\cdot 1} T^T V^T A^r_{\cdot 2}$	$A'^f_{\cdot 2} \leftarrow A^f_{\cdot 2} - V^c_{\cdot 1} T^T V^T A^r_{\cdot 2}$

*$A_{\cdot 1}$ is panel before current iteration, V and R are panels after current iteration. T is triangular factor matrix. $A_{\cdot 2}/A'_{\cdot 2}$ is trailing matrix before/after current iteration.

There are three major steps in each iteration of QR decomposition: PD, computing triangular factor (CTF), and TMU. In exiting QR decomposition with ABFT [31], only row checksum is used to protect row panel R as shown in the second column of **Table III**.



Before current iteration, benefiting from our new full checksum maintaining PD algorithm, both column checksum c(V) and row checksum r(R) are maintained for partially decomposed part V and R.

During TMU, with column checksum of column panel V21 and row checksum of trailing matrix A12 and A22, full checksum can be maintained for the new trailing matrix A'22.

Fig. 3: Full checksum QR decomposition.

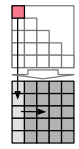
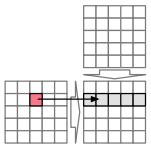
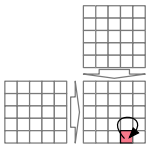
Algorithm 1 FT-xGEQRF2

1: **input:** panel P^f, size: $(m+1) \times (NB+1)$.
2: **output:** Householder vectors V^c.
3: **output:** Upper triangular matrix R^r.
4: **for** $j = 1 : NB$ **do**
5: $x = P_{j:m,j}$
6: $v = x + sign(x_1) \|x_{1:last-1}\|_2 e^c_1$
7: $v_{last} = v_{last} - P_{j-1,j}$
8: $P_{last,j:last} = P_{last,j:last} - P_{j-1,j:last}$
9: $v = v/\|v\|_2$
10: $P_{j:m+1,j:NB+1} = P_{j:m+1,j:NB+1} - 2vv^T_{1:last-1} P_{j:m,j:NB+1}$
11: $V^c \leftarrow v$
12: $R^r \leftarrow$ upper triangular part of P
13: **end for**

To maintain full checksum for QR, we need to maintain full checksum for TMU. So we need to be able to maintain column checksum for the first matrix operand and row checksum for the last matrix operand in the matrix-matrix multiplication used in TMU [34] (i.e., column checksum of Householder vectors $V_{\cdot 1}$ and row checksum for trailing matrix $A_{\cdot 2}$). The challenge lies in maintaining checksum for PD, since only row checksum can be maintained for decomposed upper triangular matrix R as shown in previous works [31]. However, column checksum cannot be naturally maintained for Householder vectors V in PD, due to its orthogonality [31].

In this work, we develop a new checksum maintaining algorithm for PD of QR that can maintain both column checksums for Householder vectors $V_{\cdot 1}$ and row checksums for the upper triangular part R_{11}. To maintain column checksum for Householder vectors, we need to capture information during PD, so the new checksum maintaining algorithm needs to be integrated with the computation of original PD. The pseudo code of PD integrated with our new checksum maintaining algorithm is shown in **Algorithm 1**. Before the PD, we first encode full checksum for panel. Then, we modify the Householder generating algorithm in lines 6∼8 in order to preserve its column checksums. This only brings $O(1)$ extra operation for each Householder vector generation. With checksum-ed Householder vector, we slightly modify line 10 to include column checksums. In the end we stored Householder vector together with its column checksums, so that resulting panel will have column checksums for Household vectors. With column checksums maintained, we can maintain full checksum for TMU with slight modification as shown in red symbols in the third column of **Table III** and red part in **Fig. 3**. Note that the computation of triangular factor is very irregular, which makes it hard to maintain checksums. To avoid catastrophic error propagation, we need to make sure T is correct before us-

TABLE IV: MUD of major update operations in one-sided matrix decompositions.

Operation	PD	PU		TMU	
Element Location	Update Part	Reference Part	Update Part	Reference Part	Update Part
Example element that brings maximum MUD					
MUD	2D	2D	1D	1D	0D

ing. Error in T can be detected by verifying the orthogonality of $(I - VT^TV^T)$. Since there is no checksum associated with T, we have to recover the corrupted T by re-computing it using V as shown in [31]. The runtime overhead for the verification and re-computation can be shown to be insignificant.

V. Fault model

In this work, we focus on tolerating three types of soft errors caused by faults in three important hardware components in heterogeneous systems with GPUs: CPU/GPU logic parts, CPU/GPU memory system, and PCIe.

1) **Computation error** occurs during update operations. It is caused by fault in the logic part of CPUs/GPUs, and results in calculation error (e.g., $1 + 1 = 3$). It can be observed as a standalone wrongly computed matrix element in the result. When a wrongly computed result is used to update other matrix elements, it can cause more errors.
2) **Memory system error** occurs anytime when the matrix is stored in the CPU/GPU memory system. It can occur in both off-chip memory (DRAM) or on-chip memory (cache, registers, or shared memory). It is caused by faults in memory system, and results in a error in the storage cell of memory. In this work, we only consider errors with multiple bit-flips in a word as many memory systems are equipped with ECC that cannot tolerate that kind of error. Error propagation can occur when a corrupted element is used to update other matrix elements. The difference between the off-chip memory error and on-chip memory error is that the initial corrupted matrix element is always observable for off-chip memory error. For on-chip memory error, on the other hand, the initial corrupted element is not always observable as some wrongly cached/loaded matrix elements may only get referenced, so there is no data write back.
3) **Communication error** occurs during data transfer between CPU and GPU or inter-GPU through PCIe. It is caused by faults in PCIe related hardware components, and results in a bit being wrongly transfered (e.g., bit 1 is sent, but bit 0 is received). Some PCIe Buses also have ECC that can protect single bit error in a word. So, in this work, we only consider multiple-bit error in a word. When communication error occurs, it can be observed as a standalone corrupted matrix element that appear in the receiver side after data transfer.

We assume that no more than one fault strikes the same matrix block between two neighbor checksum verifications. This is a relative rare case and can be hard to tolerate.

VI. Systematic error propagation study

Error propagation patterns in one-sided matrix decompositions was studied in [13], [32]. However, none of them was systematic enough. [32] focused on all three update operations in LU, but it failed to distinguish the errors in reference and update part. [13] did propagation study caused by error that occurs in both reference and update part, but they only carefully studied TMU and overlooked the details in other operations. In this work, we present a systematic error propagation study focusing on all major update operations and considering errors in both reference and update part.

TABLE V: Error propagation patterns of major update operations in matrix decompositions.

Operation	Computation error	Memory error		Communication error
		Reference part	Update part	
PD	2D	-	2D	-
PU	**1D**†	2D	**1D**†	-
TMU	0D*†	**1D**†	0D*†	-
Panel broadcast	-	-	-	0D*†

∗ tolerable by single-side checksum
† tolerable by full checksum
In non-tolerable cases, errors are detectable but need local in-memory recompute to recover.

A. Update patterns

We first analyze the computation patterns in each operation, which can help us characterize their error propagation patterns later. We define a term to quantify the complexness of the computation: *Maximum Update Dimensions* (MUD). MUD can be used to quantify elements or an update/reference part. MUD of an element x, denoted as $MUD(x)$, equals the maximum number of dimensions of any area where element x can directly or indirectly update in an update operation. Here the number of dimensions is defined as follows: $MUD(x)$ = 0D means x only updates itself; $MUD(x)$ = 1D means elements in whole or partial of one row or column get updated by x; $MUD(x)$ = 2D means elements beyond one row or column get updated by x; In addition, the MUD of an update/reference part A is defined as: $MUD(A) = \underset{x_{ij}\ in\ A}{max}(MUD(x_{ij}))$. This gives us a quantifiable term to measure the complexity of each operation in matrix decompositions. According to the algorithm of each operation, we summarize the MUD of each update/reference part of each update operation in **Table IV**. Each small box represents one element. Red boxes represent sample elements in corresponding update/reference part that bring the maximum MUD. Light gray/dark gray boxes represent elements that are directly/indirectly updated by red element.

B. Error propagation patterns

Error propagation happens when corrupted data is referenced for update operation, and then causes more data corruption. This is common in matrix decompositions, since elements in matrix are repeatedly referenced and updated. We define three levels of error propagation as follows:

- **0D**: a single standalone error with no error propagation;
- **1D**: an error propagates to entire/part of one row/column;
- **2D**: an error propagates beyond one row or column.

Higher degree of error propagation means the update operation is more sensitive to errors.

With update pattern analyzed in **Table IV**, we characterize the error propagation patterns. Note that we only consider the error propagation occurs within one operation. Error propagation across multiple operations can almost definitely cause 2D error propagation which is not tolerable. It is interesting to see that if an element is used to update certain other elements, the corruption to the element can also propagate in the same way as the update pattern. Depending on the type of soft error and when an error occurs, the exact error propagation pattern may be different, but we only consider the worst case where all related elements may be corrupted. So, $MUD(x)$ actually also indicates what level of error propagation would happen if x is corrupted. The corruption of x can be caused by all three kinds of soft error mentioned in our fault model. While $MUD(x)$ indicates the error propagation pattern caused by a

specific element, $MUD(A)$ indicates the worst case scenario considering all elements in it. It is the highest level of error propagation that can be caused by error in any element of A. According to our conclusion in **Table IV**, we summarize degree of error propagation as in **Table V**. Compared with previous works, [32] did not distinguish the errors in reference part and update part and they only consider the worst case, so it rated the error propagation level as follows: PD (2D), PU (2D), and TMU (1D). [13] successfully rated the different cases for TMU, but failed to look into PD and PU, in which they simply rated all of them as 2D. Our systematic study gives much more details that can help us design new ABFT checking scheme with more appropriate protection.

VII. New ABFT checking scheme

ABFT checking scheme determines when to perform checksum verification. It plays a pivotal role in determining the ABFT checking overhead. ABFT checking schemes can be classified into two categories: *prior-operation check ABFT* [11], [12] performs check on input data before each update operation; *post-operation check ABFT* [13], [31], [32] performs check on output data after each update operation. However, none of them are truly suitable for full checksum one-sided matrix decompositions, because they incur unnecessary ABFT verification overhead due to redundant verification when used with full checksum.

In this section, based on our previously designed full checksum one-sided matrix decomposition and systematic error propagation study, we design a new ABFT checking scheme that brings lower ABFT checking overhead.

A. New ABFT checking scheme design

Table V summarizes the protection capability of single-side and full checksum scheme. In addition to 0D error propagation, full checksum scheme can also tolerate 1D error propagation. When designing ABFT checking scheme, full checksum offers the following benefits:

1) It can avoid local re-computation for 1D error propagation cases, which significantly reduces recovery cost;
2) It also makes ABFT more tolerable to memory errors including DRAM and on-chip memory of CPU/GPU;
3) Since TMU can only have 0D/1D error propagation, we can partially eliminate or postpone its correctness check to reduce ABFT checking overhead without sacrificing protection strength. In addition, we can put more protection to operations that can lead to 2D propagations (e.g., PD and PU) to reduce the possibility of 2D propagations.

Algorithm 2 shows our new ABFT checking scheme. Since both PD and PU can have 2D propagation (i.e. high sensitive), we put correctness check both before and after their operations. This protection is stronger than previous works, in which they only put correctness check before [11], [12] or after [13], [31], [32] PD and PU. In addition, we postpone the post-operation correctness check of PD and PU to the time after their decomposed/update panel has been broadcasted. This further helps detect and correct communication error to avoid further propagation. Previous works [13], [31], [32] check the panel before panel broadcast. If an error occurs during communication, it may propagate to the next operation. Since we only postpone the correctness check, it does not bring extra ABFT checking overhead. Finally, we totally eliminate all correctness check of input before TMU. The reason is as follows. TMU relies on the decomposed panel and updated panel from previous PD and PU. Since we already put correctness check after those two operations, no 2D or 1D propagation can exist on those two panels before TMU. The only possible error propagation is 0D, which could be caused by memory errors while those two panels are stored in DRAM after PD/PU and before TMU. However, it can only leads to 1D propagation during TMU, so we discard all correctness check before TMU. After TMU, we propose a heuristic checking approach to protect TMU with low overhead, which will be discussed in the next subsection.

B. Heuristic checking for TMU

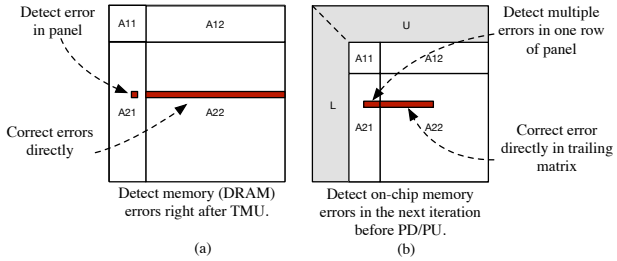


Fig. 4: Heuristic checking for TMU.

After TMU, there is no need to check the whole output. We don't need to worry about 0D propagation, since part of the result that needs attention will be verified before use in the next iteration, so that any 0D propagation will be fixed before use. For 1D propagation in TMU, we propose several heuristic checking rules for efficient handling. (1) for 1D memory (DRAM) error propagation, it must come from the memory errors in row/column panel, so we can detect them by checking row and column panel instead of the expensive correctness check to trailing matrix as shown in **Fig. 4**a. In case of error, we can just fix the corresponding rows/columns in trailing matrix. (2) for 1D on-chip memory propagation, it also comes from row/column panel, but it cannot be observed in them since memory is not corrupted (only the cached data in on-chip memory is corrupted). The only observable fact is part of one row/column of trailing matrix is incorrect. So, we leave it to the column/row panel check of the next iteration. Once we detect multiple errors occur to the same row/column of a matrix block before PD or PU, its likely that's caused by the on-chip memory error occurred in previous TMU, then we check the whole row/column to fix it as shown in **Fig. 4**b. Note that one rare case is ignored in this heuristic checking where multiple not-yet-detected on-chip 1D error propagations accumulate within the same matrix blocks that becomes 2D propagation, which needs re-compute of multiple iterations to fix. However, we can easily overcome this problem by periodically check the correctness of trailing matrix based on the on-chip memory error rate to avoid 2D propagation. For simplicity, we omit the overhead for this correctness check in our analysis. It can be shown that even if we add this correctness check, the overall ABFT checking overhead is still lower than previous works, in which they need to check the trailing matrix in every iteration.

C. Distinguish communication error with other kinds of error

Once each GPU received decomposed/update panel, it checks the correctness of the panel. If error is detected, the error could cause by computation/memory error during the last PD/PU or communication error during the panel broadcast. Distinguish communication error with other kinds of error, we count the number of GPUs that received panel with corrupted elements (i.e., corrupted panel). If all GPUs received corrupted panel, its very likely that the corruption is caused by error during last PD/PU, and the worse case is 2D error propagation, which is not correctable by ABFT. So, to be safe, we initiate local in-memory restart of the last PD/PU, and then broadcast again. We only need to make a copy of the panel before PD, which only brings slight overhead. Otherwise, if only some

Algorithm 2 New ABFT checking scheme

ngpu ← total number of GPUs
for j = 1 : N/NB **do**
  [GPU$_{j\%ngpu}$ → CPU] Transfer panel
  [CPU] ***Check the panel to be decomposed with heuristic checking for TMU***
  [CPU] Panel Decomposition
  [CPU → GPU$_{1...ngpu}$] Panel Broadcast
  [GPU$_{1...ngpu}$] ***Check decomposed panel***
  [GPU$_{1...ngpu}$] ***Check the panel to be updated with heuristic checking for TMU***
  [GPU$_{1...ngpu}$] Panel Update
  [GPU$_{1...ngpu}$] ***Check updated panel***
  [GPU$_{j\%ngpu}$ → GPU$_{1...ngpu}$] Panel Broadcast
  [GPU$_{1...ngpu}$] Trailing Matrix Update
  [GPU$_{1...ngpu}$] ***Heuristic panel checking for TMU***
end for

TABLE VI: ABFT verification comparison (one iteration).

Checking scheme	PD		PU		TMU		Total
	before	after	before	after	before	after	
prior	b		$2b$		b^2+2b		b^2+5b
post		b		b		b^2	b^2+2b
Ours	**b**		**2b**		**b**	**(K+2)b**	**(K+6)b**

of the GPU received corrupted panel, they must be caused by communication error, so we let each GPU correct those errors using checksum.

D. Distinguish 1D and 2D error propagation in PU

Computation/memory error in PU can cause either 1D error or 2D error propagation. 1D error propagation is actually correctable, although we only have single side checksum for the panel. According to **Table IV**, error is always propagated to one row/column for column/row panel and we always maintain column/row checksum for column/row panel as shown in section IV. 2D error propagation in updated panel, on the other hand, needs local in-memory restart of the last PU. The possibility of causing two different error propagation patterns are overlooked in previous works [31], [32] where they treat all cases as 2D error propagation. To distinguish the two cases, we calculate the error row and column index in a matrix block. If the calculated error locations do not reside in the same row (for column panel) or column (for row panel), it must be caused by 2D error propagation. Otherwise, we treat it as 1D error propagation.

E. Fault tolerance overhead analysis

To compare the ABFT checking overhead, **Table VI** compared the number of matrix blocks needed to be checked for correctness in one iteration. We assume the size of current undecomposed sub-matrix is $j \times j$ for simplicity. Given matrix block size NB, and we define $b = j/NB$. K is the number of memory/on-chip memory error that causes 1D propagation. As we can see, when K is small, the new ABFT checking scheme has much lower checking overhead than both existing checking schemes.

VIII. Checksum Encoding Optimization

Checksum encoding procedure is one of the key operations in our fault tolerant matrix decompositions. It is used for initializing checksums before decomposition and recalculate checksum for each ABFT check. The most common choice of implementation[11] is to use general matrix-matrix multiplication (GEMM) in highly optimized linear algebra libraries [42], [49]. However, input size of matrix of the checksum encoding makes the computation to be memory intensive rather than compute intensive. Implementations of GEMM are usually optimized for computing intensive workloads, so it causes GPU being inefficiently utilized during checksum encoding, which brings considerable high overhead for ABFT on GPUs. So, instead of using GEMM, we design a new computing kernel on modern GPUs dedicated for checksum encoding.

A. Algorithm-level optimization

In our ABFT scheme, in order to both detect and recover errors, we encode matrices with two checksums each with different weights: $v_1 = (1, 1, 1, ..., 1)^T$ and $v_2 = (1, 2, 3, ..., n)^T$. So, the column checksums of $NB \times NB$ matrix block A can be calculated as: $c(A) = [v_1 v_2]^T A$ and row checksum can be calculated similarly. To optimize, since weights in v_1 are all 1s, so we reduce the first checksum encoding into simple summation. For v_2, we hard-code its weights into the kernel to avoid unnecessary memory accesses. This allows us to reduce 25% of the $flops$ and $O(2NB^2)$ global memory accesses.

B. Memory access optimization

Optimization for memory access has been one of the most important aspects for GPU computing [50]–[52]. Checksum encoding is a memory-bound computation. So, improving its memory access efficiency is even more critical for high performance. The first challenge is ensuring full coalescing memory access given different matrix storing types or checksum types in ABFT. To optimize, we divide input matrix into smaller tiles and use threads to load tiles of data to shared memory and registers in a coalesced way. The tile loading style ensures efficient coalesced memory access regardless of the input matrix storage type or checksum encoding type. The choice of tile size can affect concurrency on GPU. We pick its size using off-line profile. The details is omitted here.

Even coalesced, long global memory access latency [53]–[55] can become another factor limiting the performance of memory-bound computations on GPU. Due to the high shared memory and register usage, the number concurrent active threads is low [56], which limits their abilities to hide memory access latency. To overcome this limitation, we use data prefetching to efficiently hide this latency. To optimize, instead of loading current tile and consuming it in current iteration, we now load the current tile in previous iteration and process it in current iteration. While we are processing current tile, we load the next tile, so that we can hide next tile loading time using current tile's processing time. As an example, **Fig. 5** shows the checksum encoding on the first two tiles. By adjusting the tile size, we can achieve good latency hiding effect and overall performance.

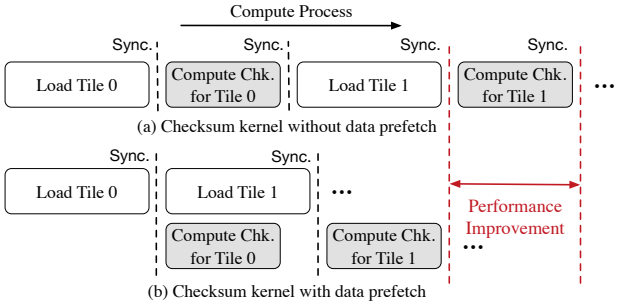


Fig. 5: Checksum encoding w/ and w/o data prefetch.

IX. Overhead Analysis

In this section, we analyze the overhead of our new ABFT scheme applied to Cholesky, LU and QR decomposition on heterogeneous system with GPUs. We show that relative performance and space overhead of all three decompositions are only small constants.

A. Performance overhead

1) Checksum encoding: Input matrix is first encoded before decomposition. Checksums are computed using our optimized checksum encoding algorithm. Compute full checksum for one block takes: $8NB^2 flops$. Cholesky decomposition only references half of the matrix (upper or lower triangular), so we only encode half of the matrix. LU and QR decomposition use the whole matrix, so the entire matrix is encoded. The relative checksum encoding overhead is:

$$O_{Cho\_enc} = \frac{T_{Cho\_enc}}{T_{Cho}} = \frac{(1/2)\times(n/NB)^2\times 6NB^2}{(1/3)n^3} = \frac{9}{n}$$

$$O_{LU\_enc} = \frac{T_{LU\_enc}}{T_{LU}} = \frac{(n/NB)^2\times 6NB^2}{(2/3)n^3} = \frac{9}{n}$$

$$O_{QR\_enc} = \frac{T_{QR\_enc}}{T_{QR}} = \frac{(n/NB)^2\times 6NB^2}{(4/3)n^3} = \frac{9}{2n}$$

2) Checksum updating: Checksum updating operations simply follows each original operation but with smaller input size. We maintain full checksums for those operations, so the relative overhead is:

$$O_{Cho\_upd/LU\_upd/QR\_upd} = \frac{T_{Cho\_upd/LU\_upd/QR\_upd}}{T_{Cho/LU/QR}} \approx \frac{6}{NB}$$

3) Checksum verification: Derived from **Table VI**, we compute the relative verification overhead of our new ABFT scheme:

$$O_{Cho\_ver} = \frac{T_{Cho\_ver}}{T_{Cho}} = \frac{24(K+4)n^2}{(1/3)n^3} = \frac{72K+288}{n}$$

$$O_{LU\_ver} = \frac{T_{LU\_ver}}{T_{LU}} = \frac{24(K+4)n^2}{(2/3)n^3} = \frac{36K+144}{n}$$

$$O_{QR\_ver} = \frac{T_{QR\_ver}}{T_{QR}} = \frac{24(K+6)n^2}{(4/3)n^3} = \frac{18K+108}{n}$$

4) Overall performance overhead: By summarize the above calculation, we derive the overall relative overhead of each decomposition as shown in **Table VII**. When matrix size is large, the relative overhead is close to a small constant.

TABLE VII: Overall Overhead.

Matrix Decomposition	Overall Relative Overhead
Cholesky	$\frac{72K+297}{n} + \frac{6}{NB}$
LU	$\frac{36K+153}{n} + \frac{6}{NB}$
QR	$\frac{36K+225}{2n} + \frac{6}{NB}$

B. Memory space overhead

Memory space overhead mainly comes from encoding checksums. We maintain full checksums for input matrix, so the relative overhead brought by the checksum encoding is:

$$O_{chk\_space} = \frac{2\times 2\times n\times n/NB}{n^2} = \frac{4}{NB}$$

X. Experimental evaluation

We evaluate our implementation on HPCC, a heterogeneous system. It is equipped with one 32-core Intel Haswell CPUs, 128 GB DRAM, and eight NVIDIA Tesla K80 GPUs with each having 12 GB memory, where GPUs are connected through PCIe. Our fault tolerant matrix decomposition is built based on MAGMA [27] 2.3.0 that is linked with cuBLAS 9.0 [42] and Intel MKL 2018.1.163 [49], where cuBLAS and Intel MKL are respectively basic linear algebra libraries on GPU and CPU. We implemented the double precision Cholesky, LU, and QR decompositions for multi-GPUs.

A. Fault tolerance protection strength test

This evaluation aims to compare two checksum encoding techniques combined with different ABFT checking schemes based on erroneous executions caused by soft errors in terms of two aspects: *fault tolerance capability* and *recovery overhead*. Evaluating based on real-world soft errors is not possible because we cannot know when errors occur, let alone if the protection approach is triggered. Thus we simulate erroneous executions caused by soft errors via fault injection in source code level. We simulate four kinds of faults in total: (1) computation error, (2) off-chip memory error, (3) on-chip memory error, and (4) communication error. Computation errors are simulated by flipping one bit in an element of the output matrix block via XOR operation. The other three kinds of errors are simulated by flipping two or more bits in the same way considering ECC can correct the error resulting from one bit flip. It should be noted that we always choose significant enough bits to be flipped such that it will make value alteration distinguishable from round-off errors based on a known round-off error bound [48]. It should be noted that the simulation of each kind of error also depends on good timing: (1) for computation error, we inject a fault to an element in the matrix immediately after a target operation; (2) for off-chip-memory error, we inject a fault to an element before an operation; (3) for on-chip-memory error, we inject a fault to an element before an operation and then change it back after the operation but before ABFT correctness check; and (4) for communication error, we inject a fault to an element immediately after it is received. Note we only inject one fault that causes one kind of error in one execution and thus can observe if the ABFT protection is effective.

We compare single-side checksum prior-operation check ABFT [11], single-side checksum post-operation check ABFT [31], [32], full checksum post-operation check ABFT [13], and full checksum ABFT with our new ABFT checking scheme (note full checksum prior-operation check ABFT is non-existent). The evaluation is based on Cholesky, LU, and QR decomposition, but only the result of LU is shown in **Table VIII** due to space limit considering the evaluation with each shows very similar result (omitted results can be provided upon request).

Table VIII shows full checksum provides more comprehensive protection against the above considered errors than single-side checksum. We observe that single-side checksum fails to tolerate errors occurring in PU as it is lack of checksum protection on updated panel. Also single-side checksum provides very limited protection against memory errors in TMU since it cannot tolerate errors causing 1D error propagation. Instead, full checksum tolerates all kinds of listed errors.

Table VIII also shows ABFT checking scheme incurs up to 7% less overhead to recover from a soft error than post-operation checking scheme, which is shown by comparing our ABFT approach with the ABFT approach based on full checksum and post-operation check. The efficiency of our ABFT checking scheme results from following techniques: (1) it detects and corrects errors more timely as it prioritizes checksum verifications on sensitive operations like PD and PU, which doesn't require local restart in many cases; (2) it tolerates errors in PCIe communication with much less overhead than existing work via postponing checksum verification after panel broadcast; and (3) based on our error propagation study, we can recover from 1D error propagation with far less overhead, which was previously recovered with a more expensive method used to correct 2D error propagation, i.e., local restart.

B. Fault tolerance coverage analysis

To compare the protection coverage of our new full checksum ABFT with existing works in a statistical view, we use a probability model to estimate the expected fault recovery overhead needed for each approach given hardware error rates. We define the following cases that can occur during each operation with calculations of probability of each case. In the equations, OP represents operation, which can be replace

TABLE VIII: ABFT protection strength and overhead comparison based on LU decomposition.

Fault→	Mem.△		PD (CPU)			Panel broadcast⊗	Mem.△		PU (GPU)			Panel broadcast⊗	Mem.△		TMU (GPU)		
			Comp.∩	Mem.†					Comp.∩	Mem.†					Comp.∩	Mem.†	
	Ref.	Upd.		Ref.	Upd.		Ref.	Upd.		Ref.	Upd.		Ref.	Upd.		Ref.	Upd.
Prior (single)	-	Y◇	R, 3%	-	R, 3%	R, 5%	Y◇	Y◇	N	N	N	N	N	Y◇	Y◇	N	Y◇
Post (single)	-	R, 3%	R, 3%	-	R, 3%	R, 8%	R, 8%	N	N	N	N	N	N	Y◇	Y◇	N	Y◇
Post (full)	-	R, 3%	R, 3%	-	R, 3%	R, 8%	R, 8%	R, 8%	R, 8%	R, 8%	R, 8%	R, 8%	R, 8%	Y◇	Y◇	Y, 3%	Y◇
Ours (full)	**-**	**Y◇**	**R, 3%**	**-**	**R, 3%**	**Y ◇**	**Y◇**	**Y ◇**	**Y◇**	**R, 8%**	**Y◇**	**Y◇**	**Y◇**	**Y◇**	**Y◇**	**Y, 3%**	**Y◇**

Notations: (1) △, DRAM memory fault between two operations; †, DRAM and on-chip memory fault during update operations; Also, we distinguish memory faults that occurs to the reference part and update part of an update operation; ⊗, PCIe fault during panel broadcast; ∩, computation fault in CPU/GPU during update operations. (2) Y◇, errors are fixed by ABFT with $< 1\%$ overhead in addition to fault free execution; Y, errors are fixed by ABFT with certain overhead in addition to fault free execution; R, errors are detected but need local restarting to fix with certain overhead in addition to fault free execution; N, errors are not detected and causes incorrect final results and need a complete restart.

with PD, PU, or TMU. OP' represent the operation before current the operation.

TABLE IX: Notation in Probability Model.

R_1	Floating point calculation error rate.
$R_2(T)$	Off-chip memory error (per matrix element) in a given time period of T.
$R_3(T)$	On-chip memory error (per matrix element) in a given time period of T.
R_4	PCIe data transfer error (per matrix element) between CPU-GPU and GPUs.
n	the size of current trailing matrix.
nb	block size.
$T_{OP}(n, nb)$	Time complexity of OP.
$A_{OP}(n, nb)$	Actual time cost of OP on a given platform.
$M_{OP\_U\ or\ R}(n, nb)$	Memory footprint of update part/reference part of OP in terms of number of matrix elements.
$M_{OP\_BC}(n, nb)$	The amount of data transfered after OP in terms of number of matrix elements.

A: No calculation error occurred during an update operation $P(A) = (1 - R_1)^{T_{OP}(n,nb)}$;

B: A calculation error occurred during an update operation. $P(B) = T_{OP}(n, nb) \times (1 - R_1)^{T_{OP}(n,nb)} \times R_1$;

C: No off-chip memory error occurred among matrix elements in the update/reference part of an operation (in between update operations) $P(C) = (1 - R_2(A_{OP}(n, nb)))^{M_{OP\_U\ or\ R}(n,nb)}$;

D: An off-chip memory error causes one matrix element in the update/reference part of an operation being wrongly stored (in between update operations) $P(D) = M_{OP\_UorR}(n, nb) \times (1 - R_2(A_{OP}(n, nb)))^{M_{OP\_U\ or\ R}(n,nb)-1} \times R_2$;

E: No off-chip/on-chip memory error occurred among matrix elements in the update/reference part of an operation (during an update operation)$(P(E) = 1 - R_{2\ or\ 3}(A_{OP}(n, nb)))^{M_{OP\_U\ or\ R}(n,nb)}$;

F: An off-chip/on-chip memory error causes one matrix element in the update/reference part of an operation being wrongly stored (during an update operation) $P(F) = M_{OP\_U}(n, nb) \times (1 - R_{2\ or\ 3}(A_{OP}(n, nb)))^{M_{OP\_U\ or\ R}(n,nb)-1} \times R_{2\ or\ 3}$;

G: No error during broadcasting $P(G) = (1 - R_4)^{M_{OP\_BC}(n,nb)}$;

H: PCIe error causes one matrix element being wrongly transfered during broadcasting $P(H) = M_{OP\_BC}(n, nb) \times (1 - R_4)^{M_{OP\_BC}(n,nb)-1} \times R_4$;

In our analysis, we assume at most one faulty case can occur to one operation at the same time. We calculate four possible outcome that each operation can have:

- **Fault Free**: None of the error we consider in the work occurred during the operation;
- **ABFT Fixable**: An error is detected and can be recovered by ABFT;
- **Local Restart**: An error is detected but cannot be recovered by ABFT. Local restart (only restart the faulty operation) is needed to recover;
- **Complete Restart**: An error has occurred, but it is not detectable until the very end of computation. The whole computation needs to restart to recover;

We use one iteration of LU decomposition as an example here. We set $T_1 = 1e-13$, $T_2 = 1e-9$, $T_3 = 1e-9$, $T_4 = 1e-11$, $n = 10240$, and $nb = 256$. The values chosen here are only for illustration propose. Actual error rate highly depends on multiple factors of hardware platform. The off-chip memory error is set to be linearly proportional to storage time. The on-chip memory error is set to be linearly proportional to operation's execution time. The recovery overhead for each case is based our experiment in the previous subsection. **Fig. 6, 7, 8** shows the probability of four outcomes of the three operations. We truncated the probability of fault free execution to better zoom in to the part with faults. **Fig. 9, 10, 11** shows expected time cost for fault recovery given the probability of four outcomes of the three operations. We can see that by combining the full checksum and our new checking scheme, the new ABFT brings wider coverage and lower or similar fault recovery overhead compared with previous works.

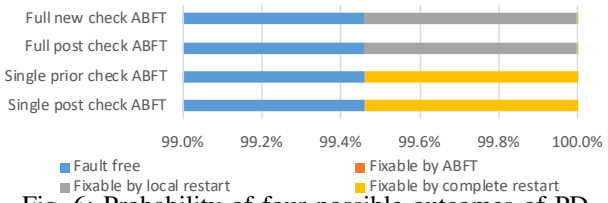


Fig. 6: Probability of four possible outcomes of PD.

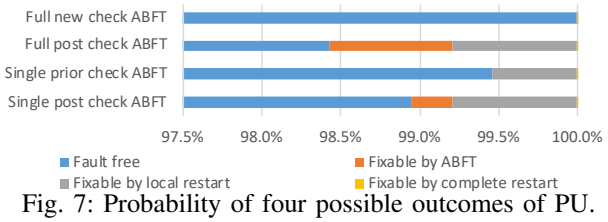


Fig. 7: Probability of four possible outcomes of PU.

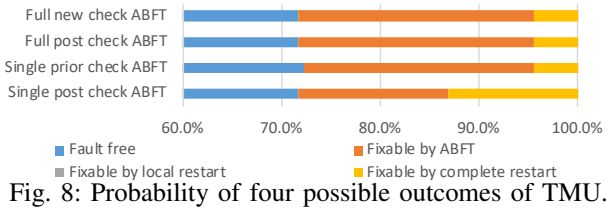


Fig. 8: Probability of four possible outcomes of TMU.

C. Performance boost of checksum encoding

Our specialized designed kernel boosts the performance of checksum encoding significantly and thus also reduces the

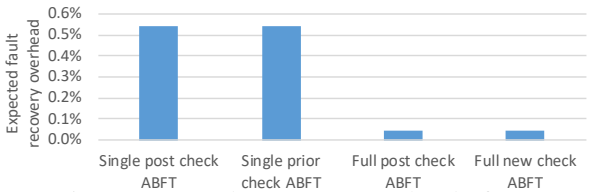


Fig. 9: Expected Recovery Overhead of PD.

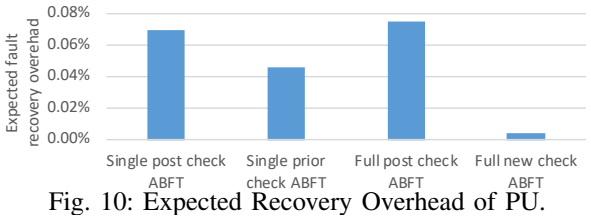


Fig. 10: Expected Recovery Overhead of PU.

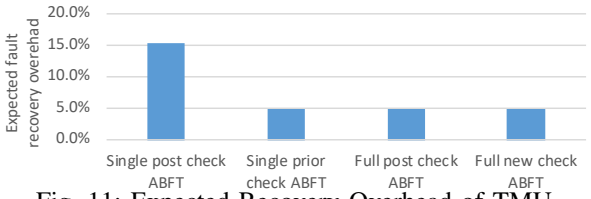


Fig. 11: Expected Recovery Overhead of TMU.

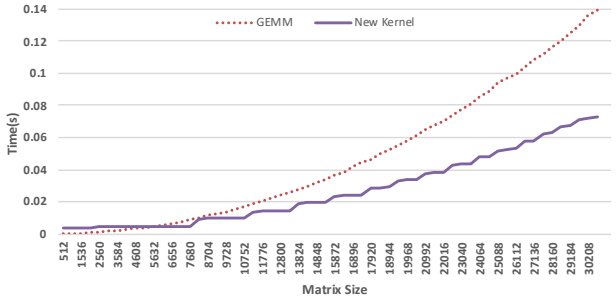


Fig. 12: Performance of new checksum encoding kernel vs. default (GEMM).

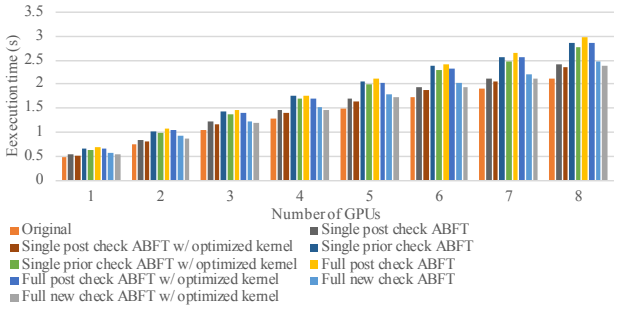


Fig. 13: Overhead comparison of Cholesky decomposition.

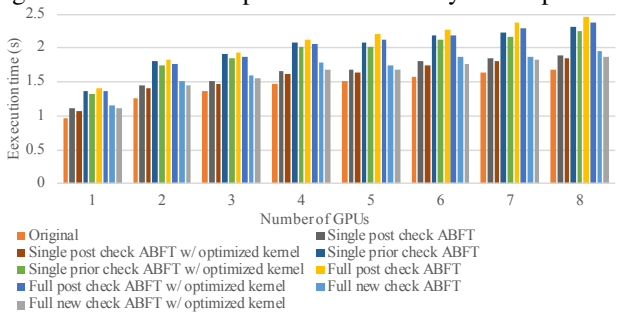


Fig. 14: Overhead comparison of LU decomposition.

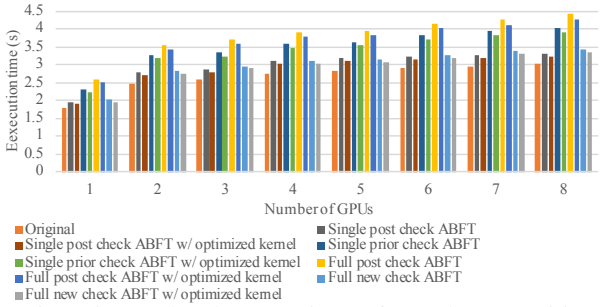


Fig. 15: Overhead comparison of QR decomposition.

overall fault tolerance overhead. We evaluate the performance boost of checksum encoding by comparing the checksum encoding performance using our kernel and that of the default in the same ABFT framework on different matrix sizes. **Fig. 12** shows our kernel achieves 1.7x speedup on average and up to 1.9x speedup. This positive result demonstrates our kernel makes far more efficient use of GPU to encode checksums.

D. Scalability and overhead comparison

A practical ABFT approach should incur *low overhead* and demonstrate *good scalability*. To evaluate the effectiveness of our new ABFT approach, we compare following four methods: (1) single-side checksum prior-operation check ABFT, (2) single-side checksum post-operation check ABFT, (3) our new ABFT approach without the optimized checksum-encoding kernel, and (4) our new ABFT approach with the kernel. The ABFT approach used in [13] shows similar overhead to single-side checksum prior-operation check ABFT, so it is omitted here. The comparison is based on weak scaling of the three decomposition methods. For LU and QR, we fix matrix size per GPU as 10240x10240, so the whole matrix size is $(num.\ of\ gpus \times 10240) \times 10240$ For Cholesky, since the input matrix needs to be symmetric, we adjust the matrix size so that the workload on each GPU is close to 10240x10240. For simplicity, we also round the matrix size to be multiplies of block size set by MAGMA, which only brings less than 2% negligible workload change. The whole input matrix size is $round(\sqrt{num.\ of\ gpus \times 10240 \times 10240})$.

Fig. 13, 14, and **15** respectively shows the comparison result for Cholesky, LU, and QR decomposition. Note this evaluation is based on error-free execution and thus the measured overhead only comes from error detection, i.e., no overhead is spent on error recovery.

Regarding overhead, we observe following phenomenon: (1) prior-operation check incurs 20% more overhead than post-operation check, which is because the amount of input data of an operation verified by prior-operation check is usually more than that of output data verified by post-operation check; (2) our optimized checksum encoding kernel reduces the overall fault tolerance overhead by 3-5%; and (3) the overhead our new ABFT approach based on the kernel is around 10% for QR and 15% for Cholesky and LU, which is comparable to the overhead of post-operation check ABFT.

Regarding scalability, we find that the overhead of our new ABFT based on the optimized kernel remains constant in the weak scaling for each decomposition method.

XI. Conclusion

We provide an efficient ABFT approach that provides stronger protection for three major one-sided matrix decomposition methods including Cholesky, LU and QR on heterogeneous systems. First, we provide full matrix protection by using checksums in two dimensions. Second, our checking scheme is more efficient by prioritizing the checksum verification according to the sensitivity of matrix operations to soft errors. Third, we protect PCIe communication by reordering checksum verifications and decomposition steps. Fourth, we accelerate the checksum calculation by 1.7x on average via optimizing the multiplication of a regular-sized matrix and a tall-and-skinny matrix on GPU architecture. Evaluation results demonstrate that our ABFT approach provides stronger protection yet with no more overhead.

References

[1] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 243–247.

[2] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, 1979.

[3] J. F. Ziegler and H. Puchner, *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress, 2004.

[4] E. Normand, "Single event upset at ground level," *IEEE transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[5] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, 2005.

[6] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[7] T. Tsai, N. Theera-Ampornpunt, and S. Bagchi, "A study of soft error consequences in hard disk drives," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–8.

[8] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu, "LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 117–130.

[9] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Improving performance of iterative methods by lossy checkponting," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 52–65.

[10] X. Liang, J. Chen, D. Tao, S. Li, P. Wu, H. Li, K. Ouyang, Y. Liu, F. Song, and Z. Chen, "Correcting Soft Errors Online in Fast Fourier Transform," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 30.

[11] J. Chen, X. Liang, and Z. Chen, "Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus," in *2016 International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[12] J. Chen, S. Li, and Z. Chen, "GPU-ABFT: Optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs," in *Networking, Architecture and Storage (NAS), 2016 International Conference on*, 2016.

[13] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, J. Chen, and Z. Chen, "Towards practical algorithm based fault tolerance in dense linear algebra," in *Proceedings of the 25th International Symposium on High-Performance Parallel and Distributed Computing*, 2016.

[14] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen, "New-sum: A novel online abft scheme for general iterative methods," in *Proceedings of the 25th International Symposium on High-Performance Parallel and Distributed Computing*, 2016.

[15] 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*, 2010.

[16] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux *et al.*, "Understanding GPU errors on large-scale hpc systems and the implications for system design and operation," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 331–342.

[17] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 221–230.

[18] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda, "GPGPUs: how to combine high computational power with high reliability," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 341.

[19] B. Fang, J. Wei, K. Pattabiraman, and M. Ripeanu, "Evaluating error resiliency of GPGPU applications," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 1502–1503.

[20] B. Fang, J. Wei, K. Pattabiraman, and M. Ripeanu, "Towards building error resilient GPGPU applications," *SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.

[21] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "A systematic methodology for evaluating the error resilience of GPGPU applications." *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3397–3411, 2016.

[22] 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*, 2011.

[23] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in GPUs: a direct measurement approach," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 294–307.

[24] J. Tan, S. L. Song, K. Yan, X. Fu, A. Marquez, and D. Kerbyson, "Combating the reliability challenge of GPU register file at low supply voltage," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 2016, pp. 3–15.

[25] S. Catalán Pallarés, J. R. Herrero Zaragoza, E. S. Quintana Ortí, and R. Rodríguez Sánchez, "Energy balance between voltage-frequency scaling and resilience for linear algebra routines on low-power multicore architectures," 2017.

[26] "CULA:." [Online]. Available: www.culatools.com

[27] "MAGMA:." [Online]. Available: icl.cs.utk.edu/magma

[28] J. Chen, L. Tan, P. Wu, D. Tao, H. Li, X. Liang, S. Li, R. Ge, L. Bhuyan, and Z. Chen, "GreenLA: green linear algebra software for gpu-accelerated heterogeneous computing," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 667–677.

[29] J. Chen and Z. Chen, "Cholesky Factorization on Heterogeneous CPU and GPU Systems," in *Frontier of Computer Science and Technology (FCST), 2015 Ninth International Conference on*. IEEE, 2015, pp. 19–26.

[30] 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," *The International Journal of High Performance Computing Applications*, vol. 29, no. 4, pp. 422–436, 2015.

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

[32] 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*, 2013.

[33] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[34] K.-H. Huang, J. Abraham *et al.*, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, 1984.

[35] "NVIDIA Fermi Compute Architecture Whitepaper:." [Online]. Available: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[36] "NVIDIA Kepler Compute Architecture Whitepaper:." [Online]. Available: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[37] "NVIDIA Maxwell Compute Architecture Whitepaper:." [Online]. Available: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

[38] "NVIDIA Pascal Compute Architecture Whitepaper:." [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[39] "NVIDIA Volta Compute Architecture Whitepaper:." [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[40] H. Cho, C.-Y. Cher, T. Shepherd, and S. Mitra, "Understanding soft errors in uncore components," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[41] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, "Fault simulation and emulation tools to augment radiation-hardness assurance testing," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 2119–2142, 2013.

[42] "CUBLAS:." [Online]. Available: developer.nvidia.com/cuBLAS

[43] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen, "Fault tolerant matrix-matrix multiplication: correcting soft errors on-line," in *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, 2011.

[44] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.

[45] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*. Atlanta, GA: IEEE Computer Society, April 19-23 2010, pp. 1–8, DOI: 10.1109/IPDPSW.2010.5470941.

[46] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with GPUs," *Numerical Computations with GPUs*, pp. 1–26, 2014.

[47] P. Wu, N. DeBardeleben, Q. Guan, S. Blanchard, J. Chen, D. Tao, X. Liang, K. Ouyang, and Z. Chen, "Silent data corruption resilient two-sided matrix factorizations," in *Proceedings of the 22nd Principles and Practice of Parallel Programming*, 2017.

[48] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[49] "MKL:." [Online]. Available: software.intel.com/en-us/mkl

[50] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal, "Adaptive and transparent cache bypassing for gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 17.

[51] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, pp. 297–311.

[52] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained synchronizations and dataflow programming on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 109–118.

[53] D. Shen, S. L. Song, A. Li, and X. Liu, "Cudaadvisor: Llvm-based runtime profiling for modern gpus," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 214–227.

[54] A. Li, Y. Tay, A. Kumar, and H. Corporaal, "Transit: A visual analytical model for multithreaded machines," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 101–106.

[55] A. Li, S. L. Song, E. Brugel, A. Kumar, D. Chavarria-Miranda, and H. Corporaal, "X: A comprehensive analytic model for parallel machines," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 242–252.

[56] A. Li, S. L. Song, A. Kumar, E. Z. Zhang, D. Chavarría-Miranda, and H. Corporaal, "Critical points based register-concurrency autotuning for gpus," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1273–1278.