UNIVERSITY OF CALIFORNIA
RIVERSIDE

# Fault Tolerant and Energy Efficient One-Sided Matrix Decompositions on Heterogeneous Systems with GPUs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jieyang Chen

March 2019

Dissertation Committee:

Dr. Zizhong Chen, Chairperson
Dr. Laxmi N. Bhuyan
Dr. Rajiv Gupta
Dr. Daniel Wong
Dr. Zhijia Zhao

The Dissertation of Jieyang Chen is approved:

Committee Chairperson

University of California, Riverside

# Acknowledgments

First and foremost, I am extremely grateful to my PhD advisor, Dr. Zizhong Chen, for his guidance, patience, and support. Without his help, I would not have been here. I would like to thank Dr. Zizhong Chen for leading me into the research field of High Performance Computing. Dr. Zizhong Chen gave me the freedom to choose research topics that interest me, in the meantime, teaching me how to do research and make high quality research contributions. Dr. Zizhong Chen taught me to seriously treat every detail discovered during research. His enthusiasm in research motivates me to keep working on research on my future career path.

I would like to thank Dr. Bhuyan N. Laxmi, Dr. Rajiv Gupta, Dr. Daniel Wong, and Dr. Zhijia Zhao for serving on my PhD dissertation committee. They provided me insightful comments and valuable suggestions to help me finish a well-structured and high-quality thesis.

I am very lucky to have collaborated with many excellent researchers and scientists. Particularly, I am especially grateful to Dr. Qiang Guan for his support during my long-term internship at Los Alamos National Laboratory. I would like to thank Dr. Shuaiwen Leon Song for providing me the internship at Pacific Northwest National Laboratory.

I am indebted to all members of the Supercomputing Laboratory at UCR, Dr. Li Tan, Dr. Panruo Wu, Dr. Dingwen Tap, Dr. Hongbo Li, Xin Liang, Sihuan Li, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, and Elisabeth Giem. I would like to thank everyone for their good advice, collaboration, and assistance.

Last but not least, I would like to thank my family for all their love and encouragement. For my parents who raised me with a love of science and supported me in all my pursuits. For my loving, supportive, encouraging, and patient wife Nan Xiong, who has been with me all these years and has made them the best years of my life. Thank you!

**Funding Acknowledgement** I am grateful to the funding supports from the National Science Foundation under Grant No. 1305624, No. 1305622, No. 1513201, and No. 1619253, the US Government contract DE-AC52-06NA25396 for Los Alamos National Laboratory (operated by Los Alamos National Security, LLC, for the US Department of Energy), the SZSTI basic research program JCYJ20150630114942313, and the Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase). Results presented in this work were obtained using the Chameleon Cloud sponsored by the National Science Foundation.

**Publication Acknowledgement**

I acknowledge that part of this thesis was published previously in the following conferences.

- Chapter 2 was previously published [50] in the *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2016)*, Chicago, Illinois, USA, May 23 - 27, 2016.

- Chapter 3 was previously published [52] in the *Proceedings of the 30th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2018)*, Dallas, Texas, USA, Nov 11 - 16, 2018.

- Chapter 4 was previously published [54] in the *Proceedings of the 28th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016)*, Salt Lake City, Utah, USA, Nov 13 - 18, 2016.

To my parents for all the support.

# ABSTRACT OF THE DISSERTATION

Fault Tolerant and Energy Efficient One-Sided Matrix Decompositions on
Heterogeneous Systems with GPUs

by

Jieyang Chen

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2019
Dr. Zizhong Chen, Chairperson

Heterogeneous computing system with both CPUs and GPUs has become a class of widely used hardware architecture in supercomputers. As heterogeneous systems delivering higher computational performance, they are being built with an increasing number of complex components. This is anticipated that these systems will be more susceptible to hardware faults with higher power consumption. Numerical linear algebra libraries are used in a wide spectrum of high-performance scientific applications. Among numerical linear algebra operations, one-sided matrix decompositions can sometimes take a large portion of execution time or even dominate the whole scientific application execution. Due to the computational characteristic of one-sided matrix decompositions, they are very suitable for computation platforms such as heterogeneous systems with CPUs and GPUs. Many works have been done to implement and optimize one-sided matrix decompositions on heterogeneous systems with CPUs and GPUs. However, it is challenging to enable stable and high performance one-sided matrix decompositions running on computing platforms that are unreliable and high energy consumption. So, in this thesis, we aim to develop novel fault tolerance and energy

efficiency optimizations for one-sided matrix decompositions on heterogeneous systems with CPUs and GPUs.

To improve reliability and energy efficiency, extensive researches have been done on developing and optimizing fault tolerance methods and energy-saving strategies for one-sided matrix decompositions. However, current designs still have several limitations: (1) Little has been done on developing and optimizing fault tolerance method for one-sided matrix decompositions on heterogeneous systems with GPUs; (2) Limited by the protection coverage and strength, existing fault tolerance works provide insufficient protection when applied to one-sided matrix decompositions on heterogeneous systems with GPUs; (3) Lack the knowledge of algorithms, existing system level energy saving solutions cannot achieve the optimal energy savings due to potentially inaccurate and high-cost workload prediction they rely on when they are used in one-sided matrix decompositions; (4) It is challenging to apply both fault tolerance techniques and energy saving strategies to one-side matrix decompositions at the same time given that their current designs are not naturally compatible with each other.

To address the first problem, based on the original (Algorithm Based Fault Tolerance) ABFT, we develop the first ABFT for matrix decomposition on heterogeneous systems with GPUs together with the novel storage errors protection and several optimization techniques specifically for GPUs. As for the second problem, we design a novel checksum scheme for ABFT that allows data stored in matrices to be encoded in two dimensions. This stronger checksum encoding mechanism enables much stronger protection including enhanced error propagation protection. In addition, we introduce a more efficient checking

scheme. By prioritizing the checksum verification according to the sensitivity of matrix operations to soft errors with optimized checksum verification kernel for GPUs, we can achieve strong protect to matrix decompositions with comparable overhead. For the third problem, to improve energy efficiency for one-sided matrix decompositions, we introduce an algorithm-based energy-saving approach designed to maximize energy savings by utilizing algorithmic characteristics. Our approach can predict program execution behavior much more accurately, which is difficult for system level solutions for applications with variable execution characteristics. Experiments show that our approach can lead to much higher energy saving than existing works. Finally, for the fourth problem, we propose a novel energy saving approach for one-sided matrix decompositions on heterogeneous systems with GPUs. It allows energy saving strategies and fault tolerance techniques to be enabled at the same time without brings performance impact or extra energy cost.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The heterogeneous system with CPUs and GPUs has become one of the most important system architectures in modern High Performance Computing (HPC) systems. Today, 5 of the top 10 supercomputers [30] are using heterogeneous system architecture. Since the heterogeneity enables it to efficiently handle a wide spectrum of workload types, many scientific applications and libraries [17, 9] have their release versions that are implemented and optimized for this architecture. Numerical linear algebra libraries are commonly used in many high performance scientific applications. These libraries solve systems of linear equations, linear least square problems, and eigenvalue/eigenvector problems. Among numerical linear algebra operations, one-sided matrix decomposition methods like LU, Cholesky, and QR play a pivotal role in many scientific applications. Due to their computational characteristic, their workloads are very suitable for computing systems such as heterogeneous systems with GPUs. So, much work has been done to develop highly optimized one-sided matrix decompositions on heterogeneous systems with GPUs [9, 17] and these libraries have been widely used by many scientific applications.

Driven by the ever-increasing demand of computation power, more and more powerful supercomputers have been built, which comprise an increasing number of complex components with shrinking feature size [152]. This is anticipated to result in these systems being increasingly susceptible to hardware faults in many components in HPC systems. Among those, soft errors, also known as silent data corruption (SDC), caused by hardware transient faults are especially hard to tolerate given that they usually only cause incorrect computation results without interrupting computation processes and they also have a much higher error rate compared with other types of error [150] given the same hardware condition. Heterogeneous systems with both CPUs and GPUs are no exception. Recent research has shown that GPUs are also very susceptible to soft errors [94, 170, 77, 90, 79, 80, 78] and soft error rate increases significantly as the GPU workload increases [155]. Also, energy saving approaches for GPUs based on undervolting and overclocking greatly disrupt GPU's stability and thus incur frequent soft errors [113, 156, 47]. Due to the computation pattern in one-sided matrix decompositions, the majority of matrix elements are repeatedly referenced during the decomposition process. So, soft errors can not only impact individual matrix elements but also lead to serious error propagations, which can invalidate the whole decompositions results [185, 184, 180, 65] and significantly weakens the reliability of scientific applications that heavily depend on matrix decompositions. It is very well known that, for matrix operations, the algorithm-based fault tolerance (ABFT) technique developed by Huang and Abraham in [102] introduces much lower fault tolerance overhead than more general fault tolerance approaches such as (Triple Modular Redundancy) TMR. While many works have been proposed to apply ABFT on CPUs [39, 58, 148, 151, 146, 57, 75, 123, 180, 183, 65], not much work has been proposed to optimize ABFT on one-sided matrix decompositions on heterogeneous systems with GPUs. To this end, in this thesis, we design the first ABFT for one-sided matrix decompositions on heterogeneous systems with GPUs – Enhanced On-line ABFT. By designing a novel ABFT checksum maintaining algorithm for GPUs, our Enhanced On-line ABFT effectively tolerates errors during matrix decompositions. Also, in additional to tolerating to soft errors that occur in logic circuits (e.g., computation error) as found in many previous ABFT works, our Enhanced Online ABFT can also tolerate a considerable amount of soft errors in GPU off-chip memory. Finally, the optimizations are carefully designed to reduce fault tolerance overhead on multiple generations of modern GPUs. To further improve the fault tolerance capability of handling error propagations, we present Full checksum ABFT. Full checksum ABFT is designed based on Enhanced On-line ABFT. Besides all the fault tolerance capabilities provided in Enhanced On-line ABFT, Full checksum ABFT enables also much wider and stronger error protection for matrix decompositions benefited from our novel full checksum scheme specially designed for one-side matrix decompositions. This stronger error protection can effectively reduce and tolerate error propagations during matrix decompositions.

Another thing we notice is that demanding requirements of energy efficiency in HPC nowadays are becoming prevalent and challenging due to growing power costs. Although GPUs are proven to be more energy efficient compared with CPUs, they still consume a considerable amount of power in modern computing systems. For example, typical

GPUs have peak power consumption ranges from 100 to 300 W. Modern HPC systems usually have 4 to 8 GPUs per computing node, which makes the total power consumption of heterogeneous computing nodes far higher than computing nodes with only CPUs. However, high system power consumption can usually lead to high heat dissipation, which imposes high pressure on the cooling systems. Without proper cooling at all times, systems may suffer performance degradation or hardware instability, which may impact the usability of scientific applications running on them. So, it is essential that we design systems or applications in an energy efficient way. Since many scientific applications heavily rely on numerical linear algebra libraries, their energy efficiency can greatly impact or even dominate the whole applications. Improving the energy efficiency of commonly used libraries is an effective approach to energy efficient scientific computing. Unfortunately, existing libraries are focused on performance, inconsiderate of energy savings opportunities that do not adversely impact performance. For example, MAGMA decomposes a program to tasks and schedules sequential and less parallelizable tasks on CPU and larger more parallelizable ones on GPU. Consequently, MAGMA achieves better performance than its counterpart libraries for homogeneous CPU computing. Yet, inherent in the DAG-based task scheduling in MAGMA, processing units scheduled with tasks on non-critical paths unavoidably experience idle time, i.e., slack. The slack can be further exploited for energy savings by leveraging hardware power-aware techniques including Dynamic Voltage and Frequency Scaling (DVFS). DVFS has been used to save energy on CPU by scaling down CPU speed during underused execution phases [84] [145] [144] [158], and now is also available on memory [63] [66] and GPU cards [129] [24]. Such advanced energy-saving software-based hardware techniques can further significantly boost the energy efficiency of scientific applications on heterogeneous systems. Maximizing energy saving requires accurate slack prediction, which is challenging for exiting system-level approaches. To this end, we propose a novel algorithmic slack reclamation energy-saving approach for one-sided matrix decompositions on GPUs. Our work exploits algorithmic knowledge of linear algebra operations to predict slack on CPU and GPU and use application-level DVFS strategies to reclaim the slack for energy savings. Compared to system-level solutions that rely on online learning and prediction for DVFS scheduling decisions, our work accurately pinpoints and fully reclaims the slack, achieving more energy savings with less overhead.

Finally, we notice that despite optimizations have been proposed for one-sided matrix decomposition in terms of energy efficiency and fault tolerance, none of the previous work is able to optimize both at the same time. One major challenge is that many energy efficiencies and fault tolerance approaches are not compatible with each other. For example, fault tolerance approach such as ABFT brings performance overhead, which in turns decrease energy efficiency. Energy saving approach as such overclocking or undervolting can decrease system reliability and can cause serious error propagation in matrix decompositions, which is beyond tolerable for ABFT. To this end, we propose bi-directional slack reclamation, an enhanced energy saving approach that can be applied with fault tolerance at the same time.

## 1.1 Problem Statement

The main problem in this thesis is to design fault tolerant and energy efficient techniques for one-sided matrix decompositions on heterogeneous systems with GPUs. The algorithms and implementations are for large-scale HPC applications on high-performance computing facilities. One-side matrix decompositions including Cholesky, LU, and QR decompositions are used as core computation components for many high performance scientific applications. They are implemented in many state-of-the-art numerical libraries, such as MAGMA [17] and CULA [9]. Soft errors include single and multiple bit-flips in on-chip and off-chip memory systems, computation error in logic circuits, and communication error in PCIe.

## 1.2 Thesis Statement

Specially designed ABFT schemes can enable efficient fault tolerance in one-sided matrix decompositions on heterogeneous systems with GPU. Algorithmic slack reclamation energy-saving approach for one-sided matrix decompositions on GPUs can achieve more energy savings with less overhead. Bi-directional slack reclamation enables fault tolerance and energy saving optimizations to be applied to one-sided matrix decompositions at the same time.

## 1.3 Contributions

### 1.3.1 Fault Tolerant One-sided Matrix Decompositions with Enhanced Online ABFT

To enabled fault tolerance for one-sides matrix decomposition on heterogeneous systems with GPUs, we design the first ABFT for one-sided matrix decompositions on heterogeneous systems with GPUs – Enhanced Online ABFT. By designing a novel ABFT checksum maintaining algorithm for GPUs, our Enhanced On-line ABFT effectively tolerates errors during matrix decompositions. More specifically, the contributions of this work include:

- The first ABFT for one-sided matrix decompositions on heterogeneous systems with GPUs.
- First Online-ABFT scheme to correct both computing and storage errors. Our 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.
- We develop three novel optimization techniques to optimize ABFT overhead on heterogeneous systems with GPU accelerators. Experimental results show that our ABFT only brings less than 10% performance overhead.

### 1.3.2 Fault Tolerant One-sided Matrix Decompositions with Full Checksum ABFT

To further enable stronger error propagation protection, we propose Full checksum ABFT. Full checksum ABFT enables also much wider and stronger error protection for matrix decompositions benefited from our novel full checksum scheme specially designed for one-side matrix decompositions. This stronger error protection can efficiently reduce and tolerate error propagations during matrix decomposistions. More specifically, the contributions of this work include:

- We prove that full checksum protection is applicable for all three one-side matrix decompositions. 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. Since the full checksum encodes the matrix in two dimensions, the protection comes along with the benefit strong error propagation protection.

- We give the first systematic study of error propagation pattern caused by computation, memory system, and communication error that occurs in all major operations of matrix decompositions. Based on the study results, we provide an efficient ABFT checking scheme by prioritizing the checksum verification according to the sensitivity of matrix operations, which leads to strong error protection with low fault tolerance overhead.

- By carefully reordering checksum verification, communication, and computation, our new ABFT checking scheme can also protect soft errors that occur in communications over PCIe with negligible overhead.

- 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 work.

### 1.3.3 Energy Efficient One-sided Matrix Decompositions with Algorithmic Slack Reclamation

To accurately predict the slacks in one-sided matrix decompositions and maximize energy saving, we propose a novel algorithmic slack reclamation energy saving approach for one-sided matrix decompositions on GPUs. Our work exploit algorithmic knowledge of linear algebra operations to predict slack on CPU and GPU, and use application-level DVFS strategies to reclaim the slack for energy savings. More specifically, the contributions of this work include:

- Novel algorithmic slack prediction model for one-sided matrix decompositions on heterogeneous systems with GPUs.

- An energy efficient one-sided matrix decompositions on heterogeneous systems with GPUs that effectively leverages the algorithmic characteristics of the linear algebra operations to maximize energy savings.

- Detailed theoretical and empirical comparison between the proposed designs and state-of-the-art work show that our algorithmic energy saving approaches can save upto 3x more energy than existing works.

- Our design is transparent to applications. With the same programming interface as the existing library MAGMA, existing MAGMA users do not need to modify their source codes to energy saving benefits.

### 1.3.4 Energy Efficient and Fault Tolerant One-sided Matrix Decompositions with Bi-directional Algorithmic Slack Reclamation and ABFT

We propose `PowerLA`, enhanced one-sided matrix decompositions that are both energy efficient and fault tolerance. Specifically, our contributions are listed as follows:

- We extend clock frequency range used in DVFS based slack reclamation energy saving approach to further include overclocking frequencies. This enables us to obtain more energy saving.

- We carefully design optimizations to incorporate overclocking and ABFT to ensure execution correctness.

- We propose a novel bi-directional slack reclamation, which allows slacks to be reclaimed by both processors at the same time. This greatly improves flexibility when reclaiming slacks.

- We implement our optimization on three core one-sided matrix decompositions and evaluate our implementation on our energy aware heterogeneous computing systems with GPUs. Results show that our proposed work can obtain more energy saving with fault tolerance capability at the same time.

# Chapter 2

# Fault Tolerant One-sided Matrix Decompositions with Enhanced Online ABFT

## 2.1 Introduction

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, soft errors also occur frequently in heterogeneous systems [94, 155]. A widely used efficient fault tolerance approach for matrix operations is algorithm-based fault tolerance (ABFT) technique, which was developed by Huang and Abraham in [102] since it introduces much lower fault tolerance overhead than the general techniques DMR and TMR.

Following their work, many researchers have extended ABFT to support more algorithms on different hardware platforms [65, 180, 123, 183, 57, 146, 151].

### 2.1.1 Limitations of Existing ABFTs

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

1. **Few schemes are developed and optimized for heterogeneous systems with GPU accelerators:** Classical ABFT schemes [102] 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.

2. **Capability to correct storage errors is limited:** Traditional ABFT schemes [102] 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 [182, 180, 65] can correct computing errors online in the middle of the computation, storage errors occurred between checksums verification and the next data access can not be corrected.

### 2.1.2 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 [9]. 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 [65, 180, 182, 67, 71, 106, 147, 185] 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 [102] or for general purpose microprocessors [180]. 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. Checksums updating is another relatively expensive (i.e., $\mathcal{O}(n)$) operation in ABFT. It is non-trivial to decide whether the checksums 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 checksums verification by verifying checksums every $K$ iterations, where $K$ is a parameter related to the failure rate of the system.

## 2.2 Background

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

### 2.2.1 Cholesky Decomposition in MAGMA

**Algorithm 1** MAGMA's Cholesky Decomposition

```
Require: Positive-definite $n \times n$ matrix A
for $j = 1$ to $N$ do
    [GPU] Symmetric rank-k updating (SYRK):
    $A[j,j] -= A[j,0:j-1] \times A[j,0:j-1]^T$
    Transfer $A[j,j]$ to CPU main memory $A_{CPU}[j,j]$
    [GPU] Matrix-matrix multiplication (GEMM): $A[j+1:N,j] -= A[j+1:N,0:j-1] \times A[j,0:j-1]^T$
    [CPU] Single block Cholesky decomposition (POTF2): $A_{CPU}[j,j] \rightarrow L_{CPU}[j,j]$
    Transfer $L_{CPU}[j,j]$ to GPU main memory $L[j,j]$
    [GPU] Linear systems solving (TRSM):
    $A[j+1:N,j] = A[j+1:N,j] \times L[j,j]^T$
end for
```

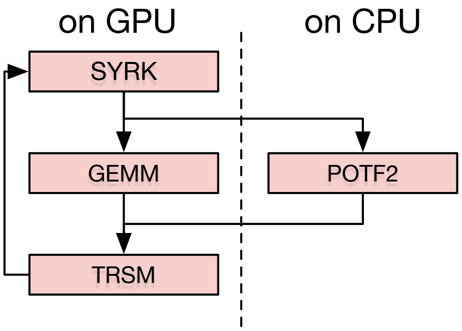


Figure 2.1: MAGMA's Cholesky decomposition

MAGMA [17] is a linear algebra library that utilizes the heterogeneous systems with GPUs. 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 2.1 and Algorithm 1, less parallelized single block Cholesky decomposition (POTF2) in line 5 is assigned to CPU and highly-parallelized symmetric rank-k updating (SYRK) in line 2, matrix-matrix multiplication (GEMM) in line 4 and linear systems solving (TRSM) in line 7 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.

### 2.2.2 Offline-ABFT and Online-ABFT

Offline-ABFT was first introduced by Abraham and Huang [102] 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 errors in results.

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

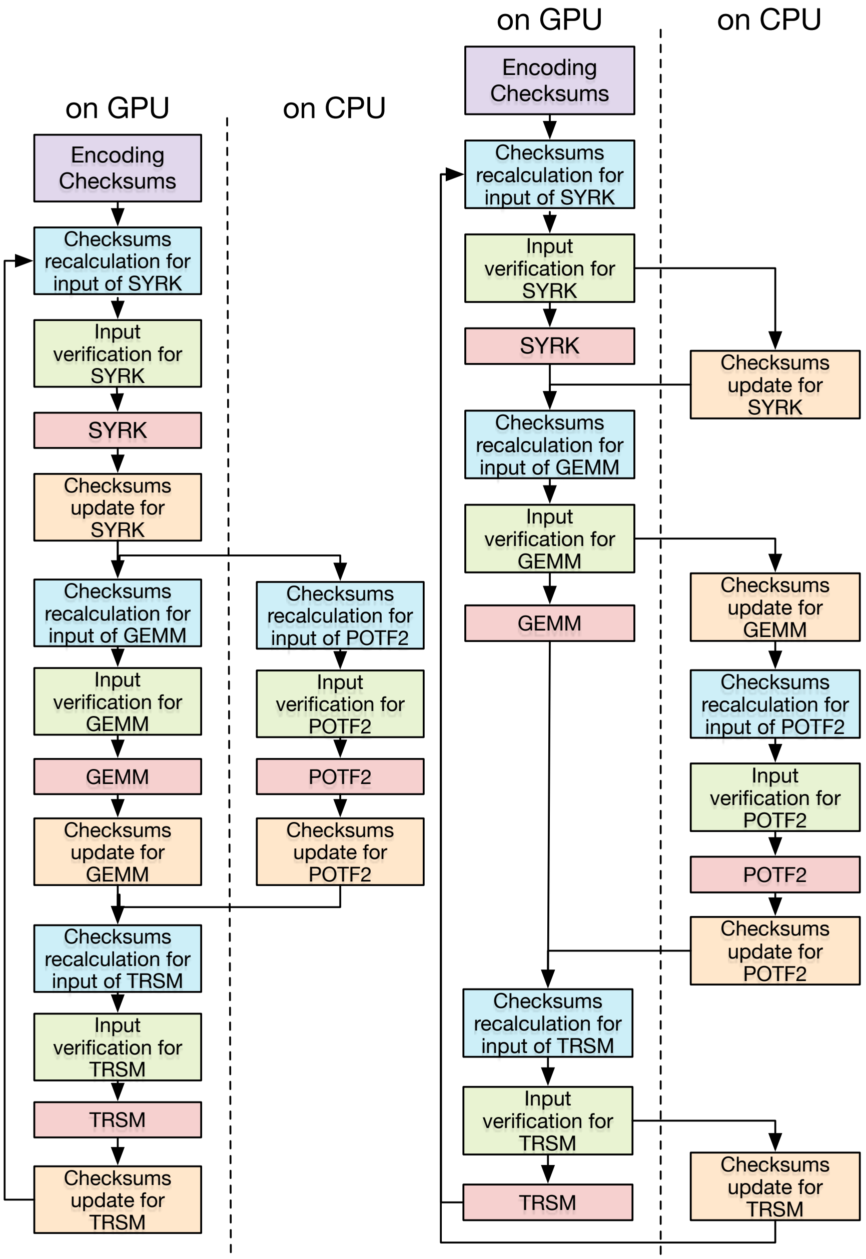
The detection and correction is done after the whole computation is complete. It can handle non-propagating soft errors, but unable to handle errors that propagate.

Online-ABFT was first introduced by Davies and Chen [57] 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.

## 2.3 Fault Model

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

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


on GPU
on CPU
Encoding Checksums
Checksums recalculation for input of SYRK
Input verification for SYRK
SYRK
Checksums update for SYRK
Checksums recalculation for input of GEMM
Input verification for GEMM
GEMM
Checksums update for GEMM
Checksums recalculation for input of POTF2
Input verification for POTF2
POTF2
Checksums update for POTF2
Checksums recalculation for input of TRSM
Input verification for TRSM
TRSM
Checksums update for TRSM
on GPU
on CPU
Encoding Checksums
Checksums recalculation for input of SYRK
Input verification for SYRK
SYRK
Checksums update for SYRK
Checksums recalculation for input of GEMM
Input verification for GEMM
GEMM
Checksums update for GEMM
Checksums recalculation for input of POTF2
Input verification for POTF2
POTF2
Checksums update for POTF2
Checksums recalculation for input of TRSM
Input verification for TRSM
TRSM
Checksums update for TRSM

(a) GPU updating (b) CPU updating

Figure 2.2: Enhanced Online-ABFT

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. In this part of the work, we only aim to tolerate memory error that occurs to off-chip memory and leave on-chip memory error to the next part of this thesis.

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.

## 2.4 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 follows:

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 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, any error in $A$ is 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 data block $B$;

4. Updating data block $B$ using data block $A$;

5. Again, as the result of the updating, any error in $B$ is detected and corrected by Online-ABFT immediately.

First, we can ensure the correctness of data block $A$ after Step 2. However, since it has been stored in the memory for a while in Step 3, some memory storage errors could appear 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 $A$ will be used to update other parts of the matrix, errors in $A$ will propagate and in some cases they 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 single block decomposition (POTF2) 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 will be longer and the probability of memory storage error will be higher. Although some memory storage errors can be fixed by the ECC feature in memory, the most commonly used ECC scheme can only fix a single bit error per word. 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. Moreover, since we use checksums to correct errors, multiple consecutive bits errors can also be corrected as long as they only corrupted one element, which is stronger than ECC. 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.**

## 2.5 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 2.2. (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 its correctness. Then, the original updating operation is performed. Finally, corresponding checksums are 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

### 2.5.1 Encoding Checksums

To encode a input matrix with checksums, 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 an 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. As mentioned by [180], 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 or row 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.

### 2.5.2 Updating Checksums

In this section, we describe the details in updating checksums for single block Cholesky decomposition (POTF2), linear systems solving (TRSM), symmetric rank-k updatring (SYRK) and matrix-matrix multiplication (GEMM). Inspired by the checksum updating algorithm in outer product Cholesky decomposition [180], 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* × 5 *blocks* matrix in Figure 2.3. Upper left gray areas represent decomposed slate area, which will not be

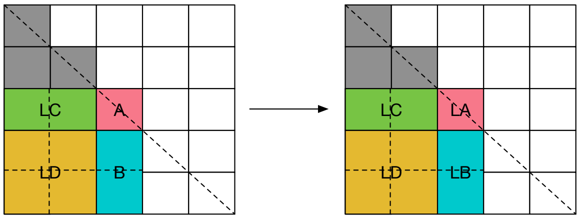
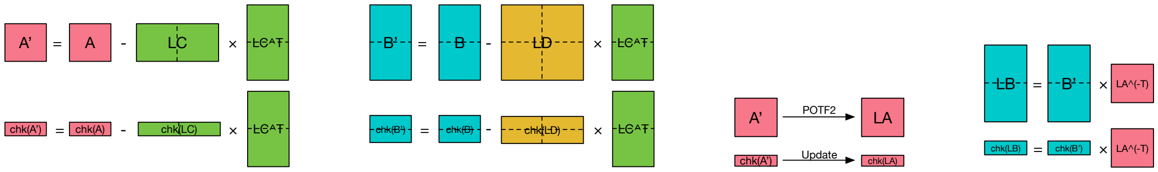


Figure 2.3: Example 5 $Blocks \times 5$ $Blocks$ Matrix



(a) SYRK (b) GEMM (c) POTF2 (d) TRSM

Figure 2.4: Checksum Update in Enhanced Online-ABFT

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).

The first step is SYRK (symmetric rank-k update) which updates the the block on the diagonal (Area A). It can be described mathematically as:

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

The checksums of $A$ can be updated as (Figure 2.4(a)):

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

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

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

The checksums update algorithm of GEMM is (Figure 2.4(b)):

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

The third step, POTF2 operation is responsible for decomposing a single block (Area A').

The checksums 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 $chk$
for $j = 1$ to $N$ do
    $chk[j] \leftarrow chk[j]/LA[j,j]$
    $chk[j+1:n] \leftarrow chk[j+1:n] - chk[j] \cdot LA[j+1:n,j]$
end for
```

As shown in Figure 2.4(c), 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 submatrix $B'$ using the result $LA$ from POTF2, which can be described as (Figure 2.4(d)):

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

So, similarly, the checksums of $B'$ are updated as (Figure 2.4(d)):

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

### 2.5.3 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) by calculate:

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

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

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 $\delta_{1i}$ give us the difference between the correct value and error value, so that we can correct it.

## 2.6 Optimizations

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 ABFT scheme.

### 2.6.1 Optimization 1

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 let 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 [29] 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.

### 2.6.2 Optimization 2

To further lower the overhead of our Enhanced Online-ABFT, we turn our focus on the checksums updating operations. Unlike checksums recalculation operations, checksums updating operations are not on the critical path. So, once the input data are verified, checksums updating can be executed concurrently with original updating operations and it can be assigned to either CPU or GPU as shown in Figure 2.2. If we put it on GPU as shown in Figure 2.2(a)(this concurrent relation is not shown), we can create a separate stream for it and utilize 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 2.2(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 assign it to CPU.

To choose between CPU and GPU for checksums updating operation, we need to determine in which way we can achieve lower overhead. We designed an estimation model to help us make this 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 checksums 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 iterations that we preform data correction verification once. It will be discussed in optimization 3. So, if we assign checksums updating to 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.

### 2.6.3 Optimization 3

As our Enhanced Online-ABFT can handle more kinds of silent errors than the Online-ABFT, it inevitably 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 overhead. The overhead of checksums 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 2.1, due to the extra number of verification in SYRK and GEMM, the overhead of our Enhanced Online-ABFT increases.

Table 2.1: Verification Comparison

| Operation | Online-ABFT | | Enhanced Online-ABFT | |
|---|---|---|---|---|
| | verify | # of blocks | verify | # of blocks |
| POTF2 | $LA$ | $O(1)$ | $A'$ | $O(1)$ |
| TRSM | $LB$ | $O(n)$ | $LA, B'$ | $O(n)$ |
| SYRK | $A'$ | $O(1)$ | $A, LC$ | $O(n)$ |
| GEMM | $B'$ | $O(n)$ | $B, LC, LD$ | $O(n^2)$ |

We noticed that memory errors may not occur so frequently as our verification frequency. Verifying data correctness in every iteration may over protect the data. So, inspired by the work [56], we designed an optimization, which allows Enhanced Online-ABFT to adjust its protection strength. The basic idea is that instead of verifying input data 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 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 to 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 $K$, we can achieve minimum overhead and still get enough fault tolerance capability.

## 2.7 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 5.1 defines the parameters we use. The overhead of different steps of our fault tolerance algorithm are shown as follow:

Table 2.2: Description of each symbol

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

1. **Overhead of checksums encoding**

   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 point operations is: $O_{encode} = 1/2 \times 4 \times B^2 \times (n/B)^2 = 2n^2$. The whole Cholesky decomposition takes $\frac{n^3}{3}$, so the relative overhead is $\frac{6}{n}$.

2. **Overhead of checksums updating**

   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 2.3):

   Table 2.3: Overhead of Checksums 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 brings little overhead, it is ignored here. So the total relative overhead of checksums updating is: $\frac{12}{n} + \frac{2}{B}$.

3. **Overhead of checksums recalculation**

   (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 is shown in Table 2.4). Both the overhead of POTF2 and SYRK is ignored here, so the total relative recalculation overhead is: $\frac{12}{n}$.

   (b) **Enhanced Online-ABFT**

   This step is done before each updating operation in Cholesky decomposition.

Table 2.4: 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 | $2n^2$ | $\frac{6}{n}$ |

The checksums are recalculated for each block that will be read or updated. The overhead of each step is shown in Table 2.5. After ignoring the minor overhead brings by the POTF2, the total relative recalculation overhead is: $\frac{6K+6}{nK} + \frac{2}{BK}$

Table 2.5: 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. **Overhead of checksums verification**

   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 checksums matrix, of which the size is: $\frac{2}{B}n^2$ and relative space overhead is: $\frac{2}{B}$.

6. **Overhead of data transfer**

   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, it involves some data transfer overhead:

(a) **Initial checksums transfer:** $\frac{2n^2}{B}$;

(b) **Checksums updating related transfer:** $\frac{n^2}{2}$;

(c) **Verification related transfer:** $\frac{n^2}{2B}$ (Online-ABFT) or $\frac{n^3}{3KB^2}$ (Enhanced Online-ABFT)

Table 2.6: Overall Overhead

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

7. **Summary**

The overall relative overhead is shown in Table 2.6. Block size B is determined by MAGMA's implementation and it is usually fixed. 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 overhead of both ABFTs will continue decrease and converging to a 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 slight lower efficiency.

## 2.8 Experimental Evaluations

### 2.8.1 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 [7] on GPU and ACML 5.3.0 [1] on the CPU. We implemented the double precision version Cholesky decomposition. The routine interface is not changed. For best performance, all checksums-related operations are also implemented with ACML-equivalent and cuBLAS-equivalent subroutines in MAGMA.

Table 2.7: 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 2.8: 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 |

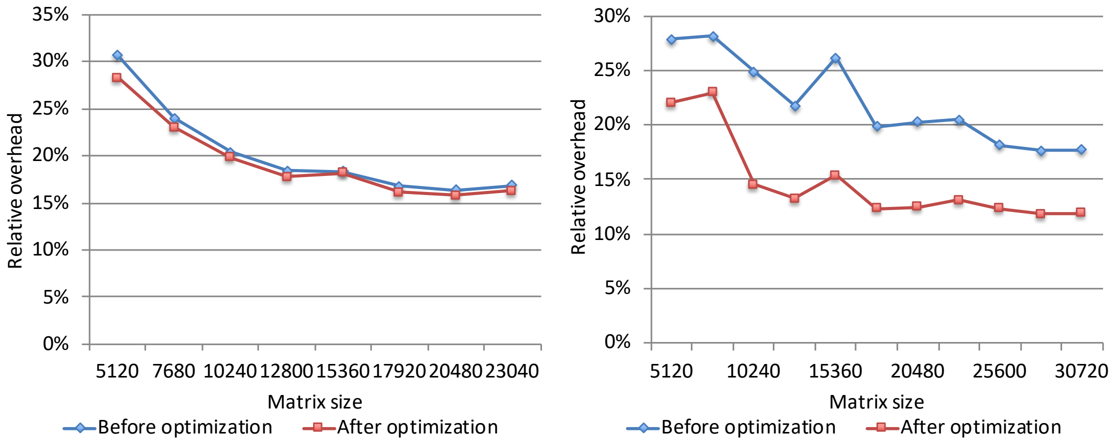
We evaluated our implementation on two heterogeneous systems: Tardis and Bulldozer64. Tardis is a cluster system with 4 GPU nodes. The GPU node is equipped with two 16 cores 2.1GHz AMD 6272 Opteron Processors with 64GB DRAM and a NVIDIA Tesla M2075 GPU with 6GB memory. The micro-architecture of the GPU 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 $5120 \times 5120$ to $23040 \times 23040$. For Bulldozer64 system, the test is from $5120 \times 5120$ to $30720 \times 30720$. As for the block size, MAGMA chooses different block sizes for different GPUs. For Fermi GPU, the default block size is $256 \times 256$ and for Kepler GPU, it uses larger block size $512 \times 512$.

### 2.8.2 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 2.7 and 2.8, 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 costs twice of the time. However, our Enhanced Online-ABFT can correct both types of errors without affecting execution time.



(a) On Fermi GPU

(b) On Kepler GPU

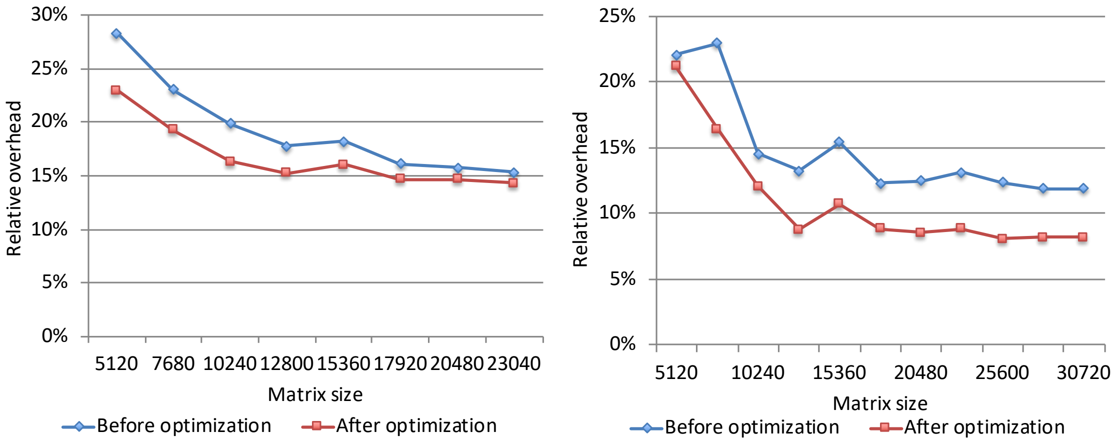
Figure 2.5: Optimization 1

### 2.8.3 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 2.5, blue line and red line represents the relative overhead before and after 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 a more powerful and advanced GPU, which could allow more checksums recalculations to be executed together, so it has much more efficiency.

### 2.8.4 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

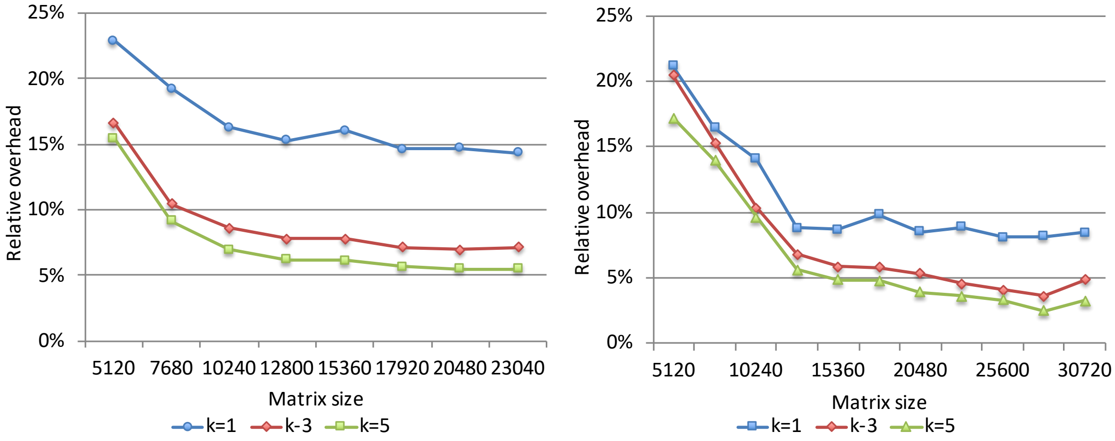


(a) On Fermi GPU

(b) On Kepler GPU

Figure 2.6: Optimization 2

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 2.6, our optimization 2 reduces the relative overhead by about 5% on Tardis on average and about 8% on Bulldozer64 on average.



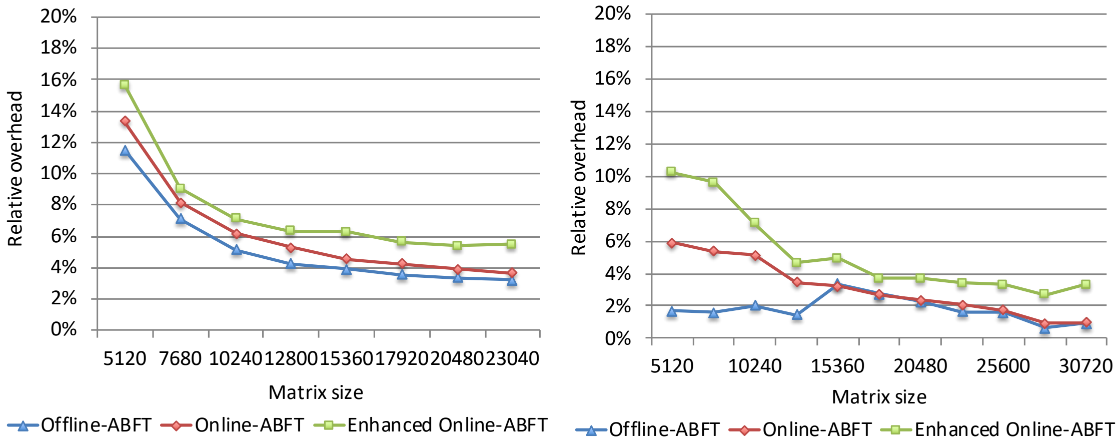
(a) On Fermi GPU

(b) On Kepler GPU

Figure 2.7: Optimization 3

### 2.8.5 Optimization 3

In this section, we show the benefit brings by the our third optimization technique. This optimization aims to adjust the frequency of checksums 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 2.7, the relative overhead of our Enhanced Online-ABFT has reduced significantly as we adjusting $K$.



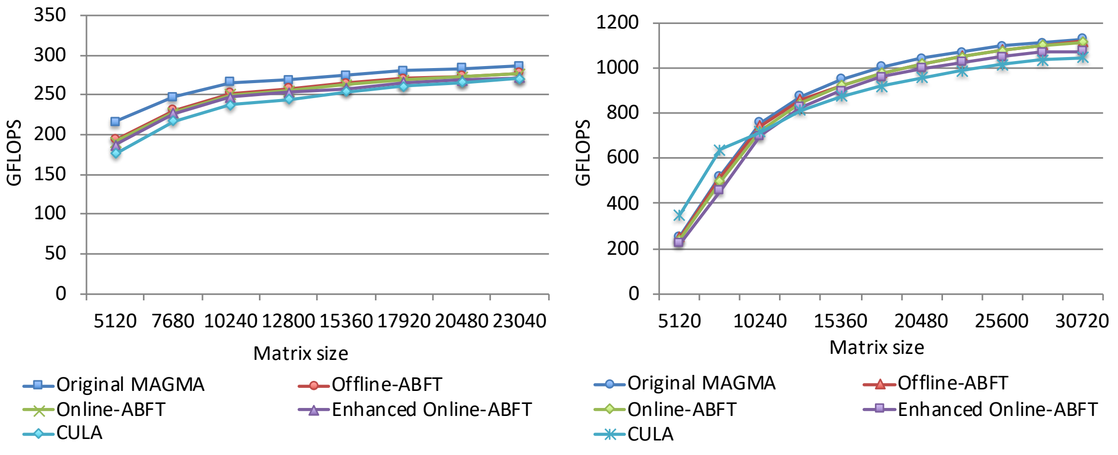
(a) On Fermi GPU

(b) On Kepler GPU

Figure 2.8: Relative Overhead

### 2.8.6 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 2.8, the overhead of our Enhanced Online-ABFT is close to constant when the matrix size is large. our Enhanced Online-ABFT only



(a) On Fermi GPU (b) On Kepler GPU

Figure 2.9: Performance on Tardis

introduced less than 6% overhead on Tardis and less than 4% overhead on Bulldozer64. It is only slightly higher than Offline-ABFT and Online-ABFT.

### 2.8.7 Performance Comparison

Figure 2.9 compares 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 2.9 indicates 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.

## 2.9 Summary

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.

# Chapter 3

# Fault Tolerant One-sided Matrix Decompositions with Full checksum ABFT

## 3.1 Introduction

In the last chapter, we introduced Enhanced On-line ABFT, which is the first ABFT designed for one-sided matrix decompositions on heterogeneous systems with GPUs. It can efficiently tolerate errors propagations in between matrix operations with computational and memory error protection, but it still has several other major limitations e.g., error propagations can occur during a single operation. Specifically, the current state-of-the-art ABFT designs have the following limitations:

1. **Insufficient protection.** Most current ABFT one-sided matrix decompositions [180, 50, 53, 65] 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 [184]. 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) [20, 21, 22, 23, 26] or current ABFT approaches.

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 [65, 180, 50, 53], 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 [59, 138, 141]. However, none of the previous ABFT approaches protect PCIe communication.

4. **Inefficient checksum calculation on GPU.** Checksum calculation [50, 53] requires the multiplication of a regular-sized matrix and a tall-and-skinny matrix based on an underlying linear algebra library [7]. However, the underlying library is very inefficient for the above type of matrix multiplication as GPU is significantly underutilized in this case.

### 3.1.1 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

Table 3.1: Notation in algorithms and formulations in this chapter.

| | |
|---|---|
| $c(A)$ | 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 in TMU. |

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 [50, 53].

## 3.2 Backgrounds

### 3.2.1 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. 3.1** 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 [171, 172, 69]. 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.

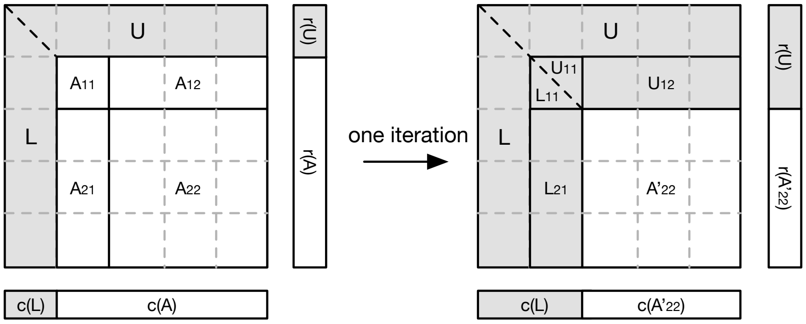


Figure 3.1: Full checksum LU decomposition.

### 3.2.2 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 [65, 50, 181]. 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 [50, 53], 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 $r(A) = A \cdot \begin{bmatrix} v_1 v_2 \end{bmatrix}$.

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 [89], round-off error bound for column and row checksum can be derived from priori norm based error bound as follows:

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

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

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, $u \approx 10^{-16}$).

### 3.2.3 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 [180, 50, 53, 65]. 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, [184] 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. 3.1** 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 (sub-matrix $A'_{22}$).

1. $A^f_{\cdot 1} \rightarrow L^c_{\cdot 1} \times U^r_{11}$
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 [184], 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 [50, 180, 65]. 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.

## 3.3 Full Checksum for Cholesky and QR

In previous works [50, 53, 180, 65], 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.

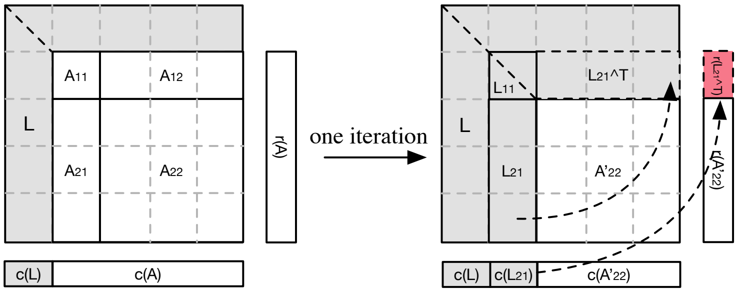
Table 3.2: Single and full checksum Cholesky decomposition.

| | Single-side Checksum | Full Checksum |
|---|---|---|
| PD | $L_{11}^{c} \leftarrow A_{11}^{c}$ | $L_{11}^{c} \leftarrow A_{11}^{c}$ |
| PU | $L_{21}^{c} \leftarrow A_{21}^{c} \times L_{11}^{-1}$ | $L_{21}^{c} \leftarrow A_{21}^{c} \times L_{11}^{-1}$ |
| TMU | $A'^{c}_{22} \leftarrow A_{22}^{c} - L_{21}^{c} \times (L_{21}{}^{T})$ | $A'^{f}_{22} \leftarrow A_{22}^{f} - L_{21}^{c} \times (L_{21}{}^{T})^{r}$ |

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

### 3.3.1 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 [180, 50, 53], 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 3.2**. Since Cholesky only decomposes half of the matrix (upper or lower triangular), it does not naturally preserve checksums for the other dimension.

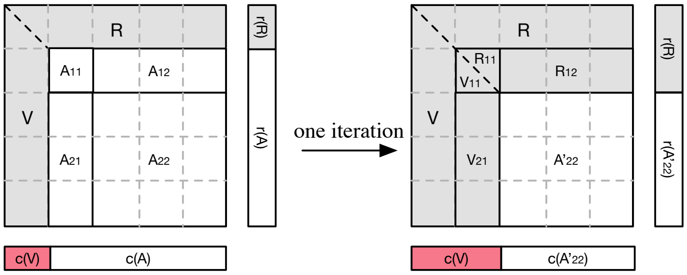


Before current iteration, column checksum c(L) is maintained for partially decomposed part L and full checksum is maintained for trailing matrix A.

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.

Figure 3.2: Full checksum Cholesky decomposition.

To maintain full checksum for Cholesky, we need to modify the TMU. Using lower triangular Cholesky as an example, as shown in **Fig. 3.2**, since the input matrix is symmetrical, column panel $L_{21}$ also serves (logically transposed) as row panel $L_{21}^T$ during TMU. Also, column checksum of column panel $c(L_{21})$ is also the row checksum $r(L_{21}^T)$ when it is logically transposed to row panel. As proved in [102], 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. 3.2**. We derive the equations for maintaining full checksum for Cholesky decomposition in the third column of **Table 3.2** (red symbols show the modifications).



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.

Figure 3.3: Full checksum QR decomposition.

Table 3.3: 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 | $A'^r_{\cdot 2} \leftarrow A^r_{\cdot 2} - V_{\cdot 1}T^TV^TA^r_{\cdot 2}$ | $A'^f_{\cdot 2} \leftarrow A^f_{\cdot 2} - V^c_{\cdot 1}T^TV^TA^r_{\cdot 2}$ |

*$A_{\cdot 1}$ 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.

### 3.3.2 Full Checksum for QR Decomposition

There are three major steps in each iteration of QR decomposition: PD, computing triangular factor (CTF), and TMU. In exiting QR decomposition with ABFT [180], only row checksum is used to protect row panel $R$ as shown in the second column of **Table 3.3**.

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 [102] (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 [180].

**Algorithm 3** 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:     $v = x + sign(x_1) \left\| x_{1:last-1} \right\|_2 e_1^c$
7:     $v_{last} = v_{last} - P_{j-1,j}$
8:     $P_{last,j:last} = P_{last,j:last} - P_{j-1,j:last}$
9:     $v = v / \left\| v \right\|_2$
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
```

However, column checksum cannot be naturally maintained for Householder vectors $V$ in PD, due to its orthogonality [180].

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 3**. 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 3.3** and red part in **Fig. 3.3**. 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 using. 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 [180]. The runtime overhead for the verification and re-computation can be shown to be insignificant.

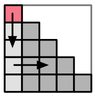
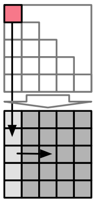
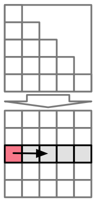
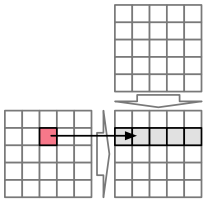
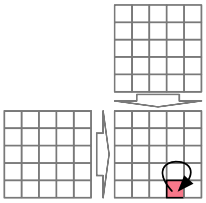
## 3.4 Fault Model

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

Table 3.4: *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 |

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.

## 3.5 Systematic Error Propagation Study

Error propagation patterns in one-sided matrix decompositions was studied in [184, 65]. However, none of them was systematic enough. [65] focused on all three update operations in LU, but it failed to distinguish the errors in reference and update part. [184] 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.

### 3.5.1 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: $MUD(x) = 0$D means $x$ only updates itself; $MUD(x) = 1$D means elements in whole or partial of one row or column get updated by $x$; $MUD(x) = 2$D means elements beyond one row or column get updated by $x$; In addition, the $MUD$ of an update/reference part $A$ is defined as: $MUD(A) = \max\limits_{x_{ij} \ in \ A}(MUD(x_{ij}))$. 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 3.4**. Each small box represents one element. Red boxes represent

Table 3.5: 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.

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.

### 3.5.2 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 3.4**, 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 3.4**, we summarize degree of error propagation as in **Table 3.5**. Compared with previous works, [65] 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). [184] 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.

## 3.6 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* [50, 53] performs check on input data before each update operation; *post-operation check ABFT* [184, 180, 65] 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.

### 3.6.1 New ABFT Checking Scheme Design

**Table 3.5** 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;

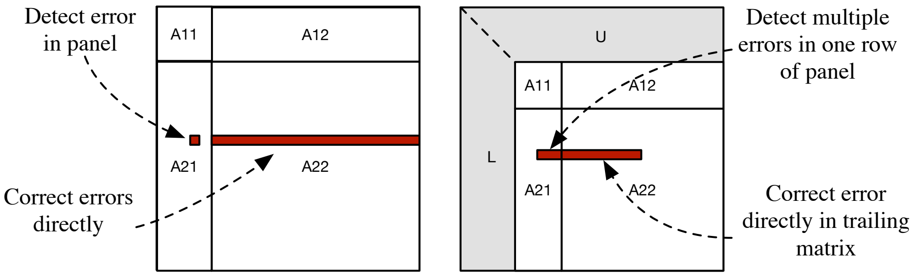
2. It also makes ABFT more tolerable to memory errors including DRAM and on-chip memory of CPU/GPU;

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 4** 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 [50, 53] or after [184, 180, 65] 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 [184, 180, 65] 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.

### 3.6.2 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


Detect error in panel
A11
A12
A21
A22
Correct errors directly
U
L
A11
A12
A21
A22
Detect multiple errors in one row of panel
Correct error directly in trailing matrix

Detect memory (DRAM) errors right after TMU.

(a)

Detect on-chip memory errors in the next iteration before PD/PU.

(b)

Figure 3.4: Heuristic checking for TMU.

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. 3.4**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. 3.4**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

**Algorithm 4** 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
```

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.

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

### 3.6.4 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 3.4**, 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 3.3. 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 [180, 65] 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.

### 3.6.5 Fault Tolerance Overhead Analysis

To compare the ABFT checking overhead, **Table 3.6** compared the number of matrix blocks needed to be checked for correctness in one iteration. We assume the size of current un-decomposed 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.

Table 3.6: 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** |

## 3.7 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 re-calculate checksum for each ABFT check. The most common choice of implementation[50] is to use general matrix-matrix multiplication (GEMM) in highly optimized linear algebra libraries [7, 18]. 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.

### 3.7.1 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: $c(A) = \begin{bmatrix} v_1 v_2 \end{bmatrix}^T A$ 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.

### 3.7.2 Memory Access Optimization

Optimization for memory access has been one of the most important aspects for GPU computing [120, 117, 119]. 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 [149, 118, 115] 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 [116], 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. 3.5** shows the checksum encoding on the first two tiles. By adjusting the tile size, we can achieve good latency hiding effect and overall performance.

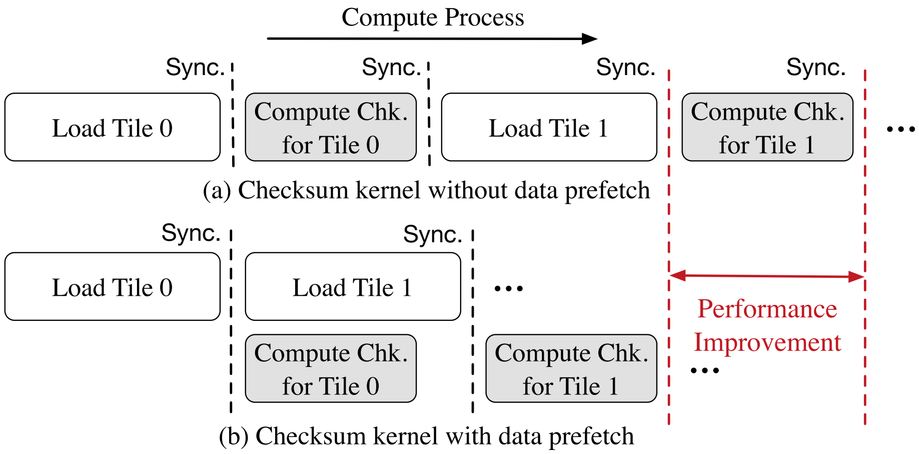


Figure 3.5: Checksum encoding w/ and w/o data prefetch.

## 3.8 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.

### 3.8.1 Performance Overhead

**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:

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

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

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

### 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:

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

### Checksum Verification

Derived from **Table 3.6**, we compute the relative verification overhead of our new ABFT scheme:

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

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

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

### Overall Performance Overhead

By summarize the above calculation, we derive the overall relative overhead of each decomposition as shown in **Table 3.7**. When matrix size is large, the relative overhead is close to a small constant.

Table 3.7: Overall Overhead.

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

Table 3.8: ABFT protection strength and overhead comparison based on LU decomposition.

| Fault→ | Mem.$^{\triangle}$ | | PD (CPU) | | | Panel broadcast$^{\otimes}$ | Mem.$^{\triangle}$ | | PU (GPU) | | | Panel broadcast$^{\otimes}$ | Mem.$^{\triangle}$ | | TMU (GPU) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Comp.$^{\cap}$ | Mem.$^{\dagger}$ | | | | | Comp.$^{\cap}$ | Mem.$^{\dagger}$ | | | | | Comp.$^{\cap}$ | Mem.$^{\dagger}$ | |
| | Ref. | Upd. | | Ref. | Upd. | | Ref. | Upd. | | Ref. | Upd. | | Ref. | Upd. | | Ref. | Upd. |
| Prior (single) | - | Y$^{\diamond}$ | R, 3% | - | R, 3% | R, 5% | Y$^{\diamond}$ | Y$^{\diamond}$ | N | N | N | N | N | Y$^{\diamond}$ | Y$^{\diamond}$ | N | Y$^{\diamond}$ |
| Post (single) | - | R, 3% | R, 3% | - | R, 3% | R, 8% | R, 8% | N | N | N | N | N | N | Y$^{\diamond}$ | Y$^{\diamond}$ | N | Y$^{\diamond}$ |
| 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$^{\diamond}$ | Y$^{\diamond}$ | Y, 3% | Y$^{\diamond}$ |
| **Ours (full)** | **-** | **Y**$^{\diamond}$ | **R, 3%** | **-** | **R, 3%** | **Y** $^{\diamond}$ | **Y**$^{\diamond}$ | **Y** $^{\diamond}$ | **Y**$^{\diamond}$ | **R, 8%** | **Y**$^{\diamond}$ | **Y**$^{\diamond}$ | **Y**$^{\diamond}$ | **Y**$^{\diamond}$ | **Y**$^{\diamond}$ | **Y, 3%** | **Y**$^{\diamond}$ |

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$^{\diamond}$, 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.

### 3.8.2 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:

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

## 3.9 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 [17] 2.3.0 that is linked with cuBLAS 9.0 [7] and Intel MKL 2018.1.163 [18], 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.

### 3.9.1 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 [89]. 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 [50], single-side checksum post-operation check ABFT [180, 65], full checksum post-operation check ABFT [184], 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 3.8** due to space limit considering the evaluation with each shows very similar result (omitted results can be provided upon request).

**Table 3.8** 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 3.8** 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.

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

Table 3.9: 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. |
| $M_{OP\_U \text{ or } R}(n, nb)$ | 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) $P(C) = (1-R_2(A_{OP}(n, nb)))^{M_{OP\_U \text{ or } R}(n,nb)}$;

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. 3.6, 3.7, 3.8** 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. 3.9, 3.10, 3.11** 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.

### 3.9.3 Performance Boost of Checksum Encoding

Our specialized designed kernel boosts the performance of checksum encoding significantly and thus also reduces the 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. 3.12** shows our kernel achieves 1.7x speedup on average and up to 1.9x speedup.

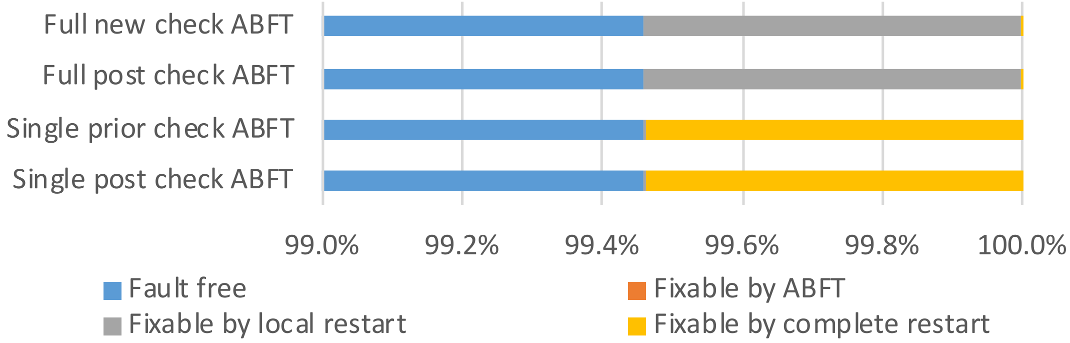


Figure 3.6: Probability of four possible outcomes of PD.

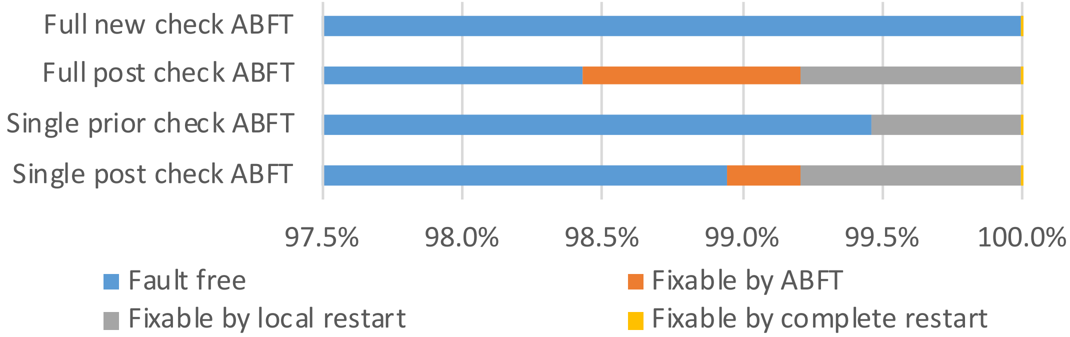


Figure 3.7: Probability of four possible outcomes of PU.

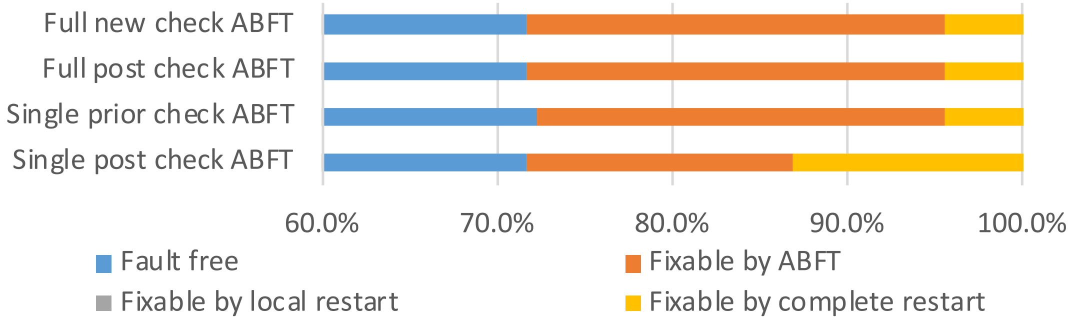


Figure 3.8: Probability of four possible outcomes of TMU.

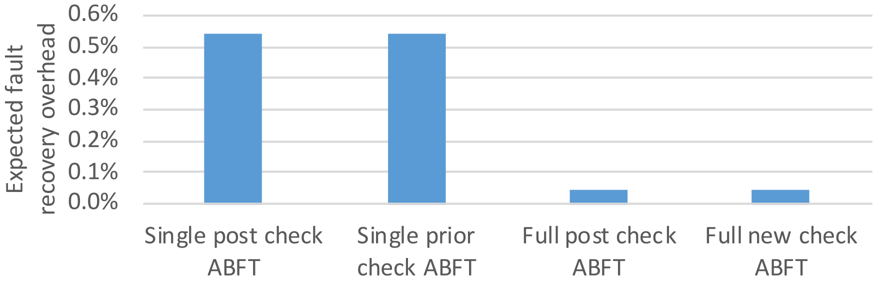


Figure 3.9: Expected Recovery Overhead of PD.

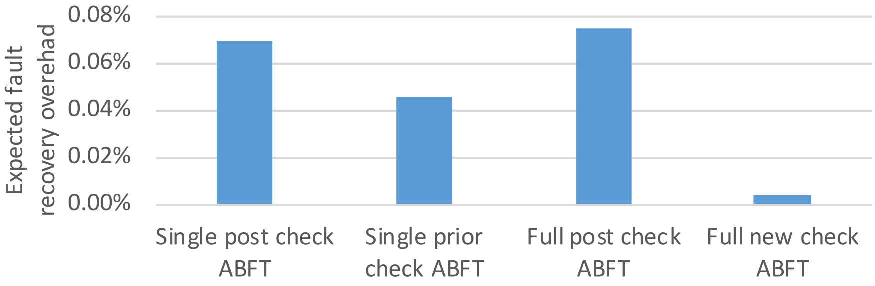


Figure 3.10: Expected Recovery Overhead of PU.

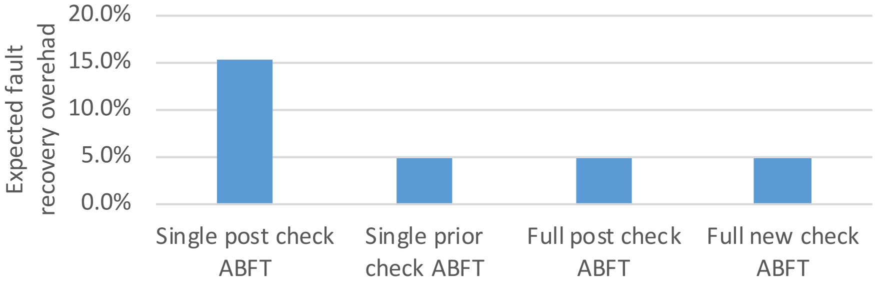


Figure 3.11: Expected Recovery Overhead of TMU.

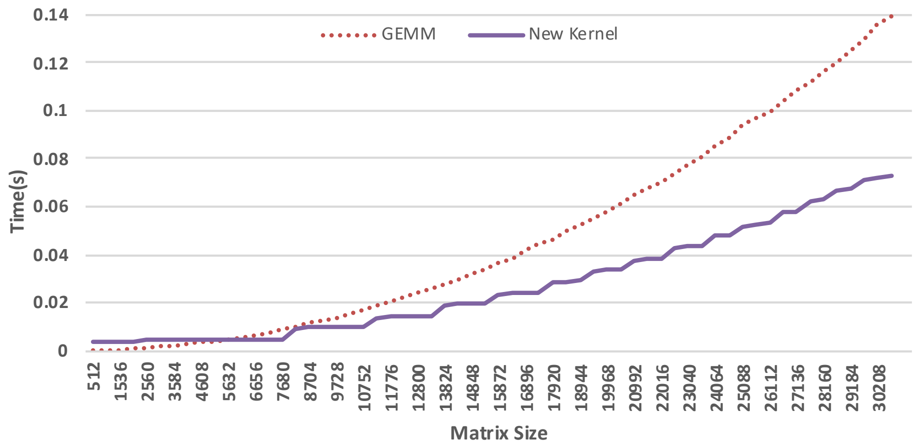


Figure 3.12: Performance of new checksum encoding kernel vs. default (GEMM).

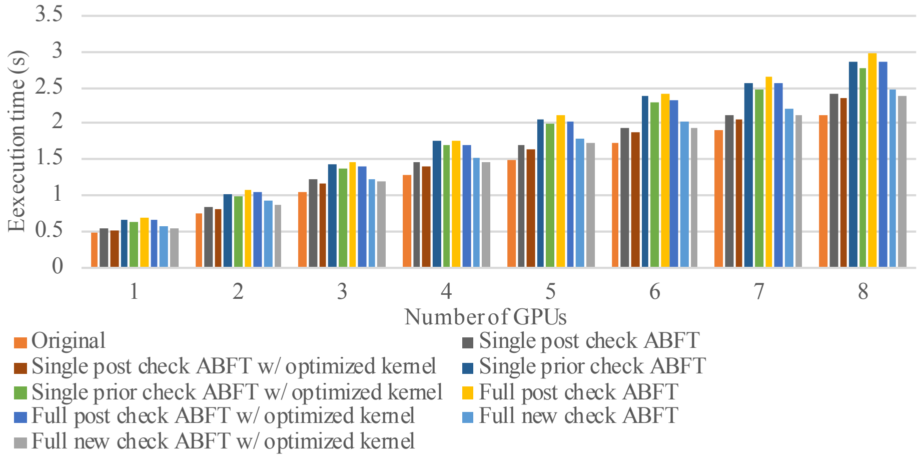


Figure 3.13: Overhead comparison of Cholesky decomposition.

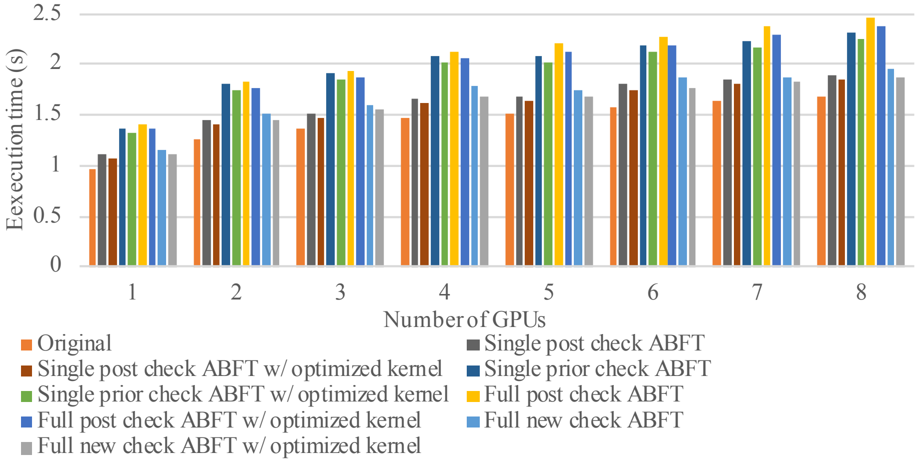


Figure 3.14: Overhead comparison of LU decomposition.

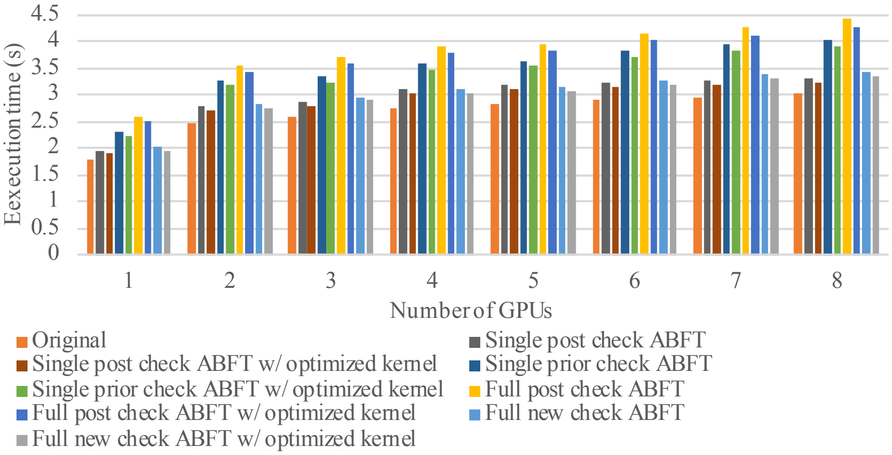


Figure 3.15: Overhead comparison of QR decomposition.

This positive result demonstrates our kernel makes far more efficient use of GPU to encode checksums.

### 3.9.4 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 [184] 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. 3.13, 3.14**, and **3.15** 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.

## 3.10 Summary

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.

# Chapter 4

# Energy Efficient One-sided Matrix Decompositions with Algorithmic Slack Reclamation

## 4.1 Introduction

Improving the energy efficiency of commonly used libraries is an effective approach to energy efficient scientific computing. Unfortunately, existing libraries are focused on performance, inconsiderate of energy savings opportunities that do not adversely impact performance. For example, MAGMA decomposes a program to tasks and schedules sequential and less parallelizable tasks on CPU and larger more parallelizable ones on GPU. Consequently, MAGMA achieves better performance than its counterpart libraries for homogeneous CPU computing. Yet, inherent in the DAG-based task scheduling in MAGMA, processing units scheduled with tasks on non-critical paths unavoidably experience idle time, i.e., slack. The slack can be further exploited for energy savings by leveraging hardware power-aware techniques including Dynamic Voltage and Frequency Scaling (DVFS). DVFS has been used to save energy on CPU by scaling down CPU speed during underused execution phases [84] [145] [144] [158], and now is also available on memory [63] [66] and GPU cards [129] [24].

Numerical linear algebra libraries are used in a wide spectrum of high performance scientific applications. These libraries solve systems of linear equations, linear least square problems, and eigenvalue/eigenvector problems. Among numerical linear algebra operations, dense matrix factorizations can sometimes take a large portion of execution time or even dominate the whole scientific application execution.

This paper presents GreenLA - an energy efficient linear algebra software package for heterogeneous scientific computing on GPU-accelerated systems. At the initial stage of the project, we analyzed highly optimized dense matrix factorization algorithms including Cholesky, LU and QR factorizations. Then we developed GreenLA to exploit algorithmic knowledge of linear algebra operations to predict slack on CPU and GPU, and use application-level DVFS strategies to reclaim the slack for energy savings. Compared to OS level solutions that rely on online learning and prediction for DVFS scheduling decisions, GreenLA accurately pinpoints and fully reclaims the slack, achieving more energy savings with less overhead. As a software package, GreenLA can work in place of existing numerical linear algebra library MAGMA. Moreover, GreenLA can be freely enabled or disabled, less intrusive than the OS level solutions.

The main contributions of this paper are:

- This paper develops GreenLA - an energy efficient linear algebra software package that effectively leverages the algorithmic characteristics of the linear algebra operations to maximize energy savings. GreenLA exploits linear algebra algorithmic knowledge combined with light-weight online profiling to accurately predict the length of tasks and slacks, and hence can maximize the reclamation of slacks via algorithm-based DVFS scheduling.
- GreenLA saves up to three times more energy than the best existing energy saving approaches that do not modify the library source codes;
- GreenLA achieves comparable performance to the highly optimized linear algebra library MAGMA but needs less energy than MAGMA.
- GreenLA is transparent to applications. With the same programming interface as the existing library MAGMA, existing MAGMA users do not need to modify their source codes to benefit from GreenLA.

The remainder of this paper is organized as follows. Section II introduces background knowledge. We present GreenLA design in Section III and its implementation in Section IV. Evaluation methodology and experimental results are provided in Sections V and VI respectively. Section VII discusses the related work and Section VIII concludes the paper.

## 4.2 Dense Matrix Factorizations on CPU-GPU Heterogeneous Systems

Dense matrix factorizations solve systems of linear equations, linear least square problems, and eigenvalue/eigenvector problems, etc. The commonly used algorithms include Cholesky, LU and QR factorizations. These algorithms decompose the coefficient matrix $A$ in a linear system $Ax = b$ into simpler forms, such as $LL^T$ and $PLU$. Consequently the solution $x$ can be calculated using forward and backward substitutions. Widely accepted heterogeneous computing libraries including MAGMA [17] provide routines of these matrix factorizations as standard functionality.

Although each suitable for different problem classes, Cholesky, LU and QR factorizations share similar algorithmic characteristics. Figure 4.1 illustrates the main steps of highly-optimized dense matrix factorizations on CPU-GPU heterogeneous systems in a global view. Factorizing a panel matrix is a Level 2 BLAS [4] operation and involves diagonal matrices factorization and panel factorization. Due to the low computational complexity and high sequentiality, panel factorization is performed on the CPU, shown as the green/yellow boxes. Updating a trailing matrix is a Level 3 BLAS operation with high computation complexity and high degrees of parallelism. It is performed on the GPU for performance efficiency, shown as the white boxes.

Figure 4.1 also demonstrates how a dense matrix factorization proceeds on a CPU-GPU platform with data movement between CPU and GPU in a local view. As mentioned, factorizing the panel matrices is executed on the CPU; and updating the trailing matrix is massively parallelized on the GPU. The *panel matrices* calculated on the CPU are of-

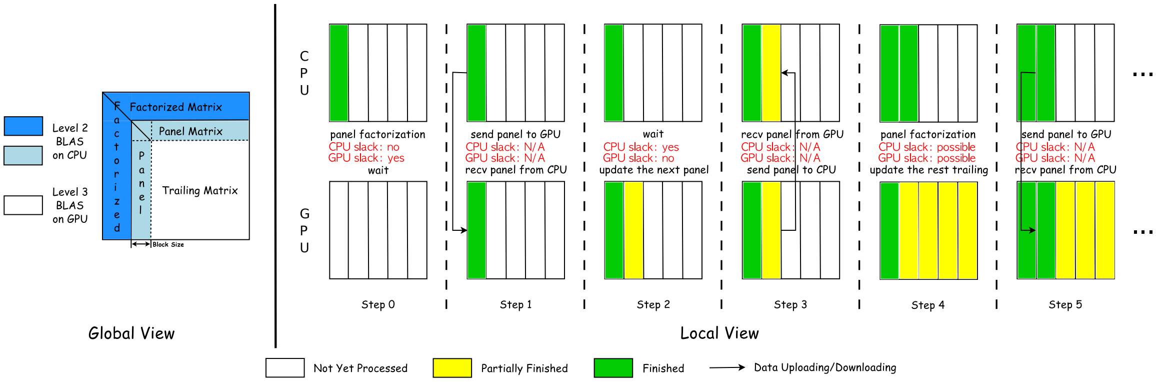


Figure 4.1: Global View and Local View of Dense Matrix Factorizations on CPU-GPU Heterogeneous Systems.

floaded to the GPU and used by the GPU to update the trailing matrices. For the sake of performance, the next panel matrix that is updated on the GPU is immediately copied back to the CPU before the entire trailing matrix finishes. As such, panel factorization is simultaneously executed on the CPU as the rest of trailing matrix is updated on the GPU. These processes proceed by a submatrix block, starting from the left upper corner of the global matrix and finishing when the whole global matrix is fully factorized.

## 4.3 GreenLA Energy Saving Methodology

At the coarse level, the matrix factorization algorithms repeatedly assign two dependent types of tasks to CPU and GPU respectively. In each iteration slack presents on both CPU and GPU. GreenLA reclaims the slack for energy savings with three main components. First, it identifies the critical path and slack on the non-critical paths on the CPU and GPU by analyzing the heterogeneous algorithms. Second, it uses algorithmic knowledge to predict and quantify the slack. Third, it exploits DVFS on the CPU and GPU to fully reclaim the slack on the non-critical paths for energy savings. The following subsections present detailed design of each components.

### 4.3.1 Critical Path and Slack Analysis

For task-parallel applications, a *slack* refers to a time period when one computer component idly waits for another. Typical causes of slack include load imbalance, inter-task or interprocess communication, and memory access stalls. Due to the pervasive presence in applications and systems, slack has been recognized as important energy saving opportunities in HPC. In a task-parallel application, a Critical Path (CP) is a particular sequence of tasks spanning from the beginning to the end of the execution where the total slack amounts to zero. While slack on the non-critical paths is usually exploited for energy savings, it is non-trivial to fully reclaim them without impacting application performance [160].

As shown in Figure 4.1, slack is present on the CPU and GPU in the heterogeneous matrix factorization algorithms. Specifically, the CPU must wait for the next updated panel from the GPU, and the GPU must wait for the factorized panel from the CPU. In addition, either the CPU waits for the GPU to finish updating the trailing matrix or the GPU waits for the CPU to finish factorizing the panel matrix. Moreover, the slack varies over iterations as the task sizes change. Being able to accurately quantify and predict the slack is necessary before reclaiming them for optimal energy savings.

### 4.3.2 Online Algorithmic Slack Prediction

For energy saving purposes, we must first know where and when the slack occurs and for how long they last. Given an application, one method to obtain such knowledge is to instrument the source code with timing functions and run the instrumented program with various problem sizes to collect the profiles. Alternatively, OS-level profiling can be performed with hardware performance counters on processors. Neither method is portable, and both methods require extensive profiling. In GreenLA, we investigate an *online algorithmic slack prediction* approach that accurately predicts the varying slack at runtime with minimum profiling.

Given the heterogeneous matrix factorization algorithms, the slack on the CPU and GPU is mainly impacted by the software and hardware parameters.

- *Problem size*: the sizes of the panel matrix and trailing matrix scheduled on the CPU and GPU respectively based on the algorithmic characteristics;

- *CPU compute capacity*: the number of floating point operations the CPU are able to perform in one second for the assigned tasks;

- *GPU compute capacity*: the number of floating point operations the GPU are able to performance in one second for the assigned tasks.

- *Data transfer speed*: the number of bytes the GPU are able to transfer between CPU memory and GPU memory.

Figure 4.2 plots the difference between CPU and GPU task execution time for the first 100 iterations of LU factorization with various problem sizes and compute rates. For instance, 10240 and 20480 are two global matrix sizes, and high_low means that the CPU runs at the highest speed and the GPU runs at the lowest speed. A non-zero difference between the execution time indicates that either CPU or GPU waits for the other in the

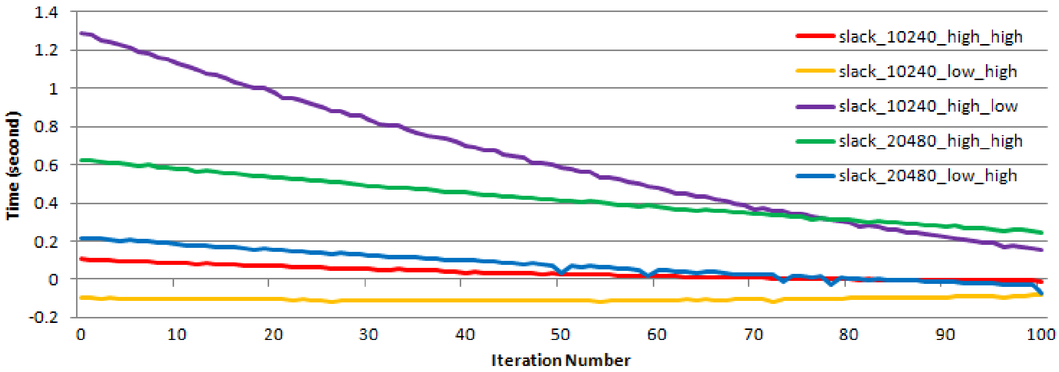


Figure 4.2: The difference between CPU and GPU execution time for the first 100 iterations of LU Factorization. The difference varies over iterations, and is a function of problem size and the CPU and GPU compute rates.

current iteration. Appropriately adjusting the compute rate of CPU or GPU can narrow and even eliminate the time difference.

In GreenLA, we leverage our prior knowledge about the factorization algorithms to quantitatively predict the slack between CPU and GPU in each iteration. Two major factors that will affect the behavior of slack are the CPU/GPU execution time and the data copy time between CPU and GPU.

We first focus on the execution time. We use LU factorization as an example here. It would be similar for Cholesky and QR factorization. Assuming the execution time of the first iteration of a $N \times N$ with block size $nb$ factorization are known, we denote the CPU time for the $N \times nb$ *panel factorization* as $T_0^{CPU}$, and the GPU time for the $N' \times N'$ *trailing updating* as $T_0^{GPU}$ where $N' = N - nb$. We can use the algorithmic information to predict the execution time of the remaining iterations. For now we consider fixed compute capacity for the CPU and GPU. As the time complexities of *panel factorization* and *trailing updating* are $O(N^3)$ for a matrix size $N$, the execution time for iteration $k$ and $k+1$ have the following relation.

$$\frac{T_{k+1}^{CPU}}{T_k^{CPU}} = \frac{O(N_{k+1}^3)}{O(N_k^3)} = \frac{(N-(k+1)\times nb)\times nb^2}{(N-k\times nb)\times nb^2}$$
$$= 1 - \frac{nb}{N-k\times nb} \tag{4.1}$$

$$\frac{T_{k+1}^{GPU}}{T_k^{GPU}} = \frac{O(N_{k+1}^3)}{O(N_k^3)} = \frac{(N-(k+1)\times nb)^2\times nb}{(N-k\times nb)^2\times nb}$$
$$= \left(1 - \frac{nb}{N-k\times nb}\right)^2 \tag{4.2}$$

Similarly, we can also predict the data copy time between CPU and GPU. Assuming the data copy time of the first iteration is $T_0^{COPY}$ for transferring a panel to GPU from CPU and then transfer it back. Assuming the data transfer speed is constant, we can use $T_0^{COPY}$ and the size change of panel over iterations to predict future data copy time. In LU factorization, the space complexity of panel needed to be transferred is $O(N^2)$, the data copy time for iteration $k$ and $k+1$ have the following relation:

$$\frac{T_{k+1}^{COPY}}{T_k^{COPY}} = \frac{O(N_{k+1}^2)}{O(N_k^2)} = \frac{(N-(k+1))\times nb^2}{(N-k)\times nb^2}$$
$$= 1 - \frac{1}{N-k} \tag{4.3}$$

Since, CPU must wait for data copy is done before it can starts it computation, the slack during iteration $k$ can be quantified as follows:

$$slack_k = \left|\ T_k^{CPU} + T_k^{COPY} - T_k^{GPU}\right|;\ 0 \le k < \frac{N}{nb} \tag{4.4}$$

A positive value indicates that the GPU have slack time to wait for the CPU, while a negative value indicates that the CPU has slack to wait for the GPU. Value zero means that the CPU and GPU finish the current iterations at the same time. The initial slack is the time difference for the first iteration, i.e., $k = 0$.

$$slack_0 = \left|\ T_0^{CPU} + T_0^{COPY} - T_0^{GPU} \right| \tag{4.5}$$

From Equations 5.1-5.6, we can quantify the slack for the rest CPU-GPU tasks, provided with the execution time of the first iteration. This algorithmic slack prediction is accurate and lightweight compared to OS level slack prediction. By using the prior algorithmic knowledge, only minimal profiling is necessary.

### 4.3.3 CP-Aware Slack Reclamation

With predicted slack over the iterations, we explore *CP-aware slack reclamation* to save energy [104] [142] [160]. To adjust the time it takes for CPU or GPU, we can either adjust the computation time or data copy time. However, since it is usually hard to accurately adjust data transfer speed and it also brings much less energy saving benefit than computation, we focus on adjusting the execution time. Specifically, we employ properly reduced compute capability on the processing units on the non-critical path such that the total slack amount to zero. We exploit DVFS, an effective power-saving hardware technology available on the CPU and GPU for this purpose. DVFS-capable processing units have multiple performance/power states. Prior studies [85] [36] [33] show that running these processing units at lower states during slack significantly reduces power and energy without impacting overall application performance. As shown in Figure 4.2, with different inputs and power states, even within one run, slack can reside at either CPU side or GPU side. We apply accordingly either CPU DVFS or GPU DVFS depending on the source of slack.

In order to quantitatively evaluate `GreenLA`, we theoretically analyze the upper bound of possible energy savings for heterogeneous matrix decomposition algorithms. Theoretically, the maximum energy savings is as follows:

$$\Delta E_{sys} = \frac{E_{new}}{E_{old}} = \frac{(P_s^{new} + P_d^{new} + P_{other}) \times T_{new}}{(P_s^{old} + P_d^{old} + P_{other}) \times T_{old}} \tag{4.6}$$

$$T_{new} = T'_{new} + T_{DVFS} \tag{4.7}$$

Here, $P_s$ and $P_d$ are the static power and dynamic power respectively consumed by the CPU and GPU, where $P_d$ is a function of processor frequency for DVFS-capable components. $P_{other}$ is the power consumption of other computer components. $T_{new}$ is the execution time of `GreenLA` including the overhead brings by the DVFS, while $T_{old}$ is the execution time of the original heterogeneous factorization. We denote $P_d = nP_{total}$, $P_s + P_{other} = (1-n)P_{total}$, where $n$ is a ratio of dynamic CPU/GPU power $P_d$ within the total system power costs. By assuming $T_{new} = T_{old}$ and adopting $P_d = \propto f^{2.4}$ from [73], we can simplify Equation 4.7 as:

$$\begin{aligned}\Delta E_{sys} &= \frac{nP_{total}\frac{f_{new}}{f_{old}}^{2.4} + (1-n)P_{total}}{nP_{total} + (1-n)P_{total}} = 1 - n\left(1 - \frac{f_{new}}{f_{old}}^{2.4}\right) \\ &= 1 - n\left(1 - \frac{nb}{N}\sum_{1}^{\frac{N}{nb}}\left(\frac{\mathsf{min}(T_k^{CPU}, T_k^{GPU})}{\mathsf{max}(T_k^{CPU}, T_k^{GPU})}\right)^{2.4}\right)\end{aligned} \tag{4.8}$$

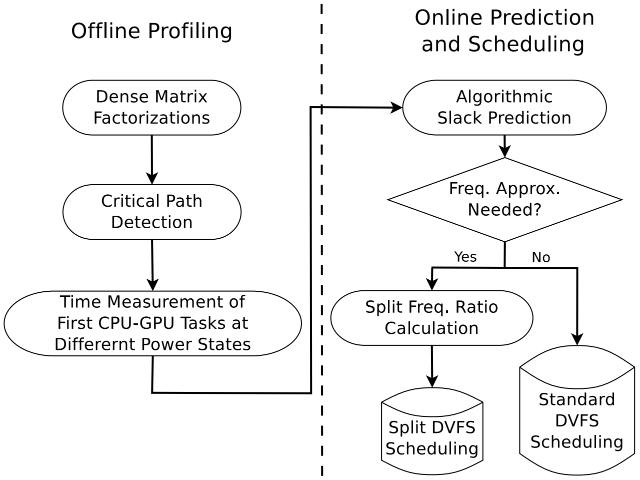

Offline Profiling
Online Prediction and Scheduling
Dense Matrix Factorizations
Critical Path Detection
Time Measurement of First CPU-GPU Tasks at Differernt Power States
Algorithmic Slack Prediction
Freq. Approx. Needed?
Yes
No
Split Freq. Ratio Calculation
Split DVFS Scheduling
Standard DVFS Scheduling

Figure 4.3: Offline and Online Framework of `GreenLA`.

## 4.4 GreenLA Design and Implementation

We present the design and implementation details of `GreenLA`, including strict and relaxed slack reclamation and coupled GPU DVFS.

Figure 4.3 illustrates the architecture and main components of `GreenLA`. It minimally profiles the application offline to obtain the execution time of the first CPU-GPU tasks at different power states and first data copy time. Such data is used to derive the slack time during the first iteration. `GreenLA` uses online algorithmic slack prediction to accurately obtain the slack for the rest of the CPU-GPU tasks. In cases where an available frequency is unable to eliminate a slack, we split the slack and use two consecutive available frequencies.

### 4.4.1 Strict Slack Reclamation

With strict slack reclamation, we apply DVFS on the non-critical path in each iteration of the factorization algorithm. By properly lowering the frequency of the processing

Table 4.1: Notation in Algorithms and Formulation.

| | |
|---|---|
| $task$ | One task of CPU-GPU dense matrix factorizations |
| $f_l$ | The lowest CPU/GPU core frequency set by DVFS |
| $f_h$ | The highest CPU/GPU core frequency set by DVFS |
| $f_l'$ | The lowest GPU core frequency paired with the highest GPU memory frequency set by DVFS |
| $f_{ideal}$ | The optimal ideal frequency to eliminate slack |
| $T$ | Execution time of a task running at $f_h$ |
| $T_x$ | Execution time of a task running at $f_x$ |
| $slack$ | Amount of time that a task can be delayed by w/o increasing the total runtime of the application |
| $f_{lower}$ | The neighboring frequency smaller than $f_{ideal}$ |
| $f_{upper}$ | The neighboring frequency greater than $f_{ideal}$ |
| $fSet$ | The frequency set consisting of all used frequencies |
| $CP$ | One task trace consisting of tasks to finish the application with the total slack of zero |
| $LastFreq$ | Frequency used after the last frequency scaling |
| $r$ | Ratio between two durations at split frequencies |

units to just eliminate the slack, we can reduce power consumption without impacting performance for the current iteration.

Prior studies have shown that the execution time of compute-intensive workloads is proportional to the frequency of processing units. Based on this observation, we derive the ideal target frequency for the processing units on the non-critical path for given current and targeting execution time. In GreenLA, we take into account the available discrete frequencies provided by CPU/GPU DVFS. In cases that the ideal target frequency is not equivalent to an available frequency, we use a weighted sum of two available neighboring frequencies and run the processing units at each frequency for a ratio of duration. Table 4.1 lists the notation used in the algorithms and formulation henceforth.

Algorithm 1 details the selection of the CPU or GPU frequency if slack occurs and Algorithm 2 presents the frequency approximation [179] [144] with *CP-aware* energy efficient DVFS scheduling. For simplicity and readability, we use five helping functions in these two algorithms: SetFreq(), GetApproxRatio(), GetSlack(), GetCurFreq(), and GetOptFreq().

**Algorithm 1** *CPU/GPU DVFS Scheduling*

```
CPU_GPU_DVFS($CP$, $fSet$, $task$, $k$)
if ($task \in CP$) then
    SetFreq($f_h$)
else
    $slack \leftarrow$ GetSlack($task$, $k$)
    if ($slack > 0$) then
        $f_{ideal} \leftarrow$ GetOptFreq($task$, $slack$)
        $LastFreq \leftarrow$ GetCurFreq()
        $fSet \leftarrow fSet \cup f_{ideal} \cup LastFreq$
    if ($T_{CPU} < T_{GPU}$) then /* CPU DVFS */
        Call CP_SSR($slack$, $fSet$) or CP_RSR($slack$, $fSet$)
    else if ($T_{CPU} > T_{GPU}$) then /* GPU DVFS */
        Call GPU_DVFS($slack$, $fSet$)
end if
```

**Algorithm 2** *Strict CP-aware Slack Reclamation*

```
CP_SSR($slack$, $fSet$)
if ($f_l \leq f_{ideal} \leq f_h$) then
    if ($f_{ideal} \notin fSet$) then
        $r \leftarrow$ GetApproxRatio($T_{lower}$, $T_{upper}$, $slack$)
        SetFreq($f_{lower}$, $f_{upper}$, $r$)
    else SetFreq($f_{ideal}$)
else if ($f_{ideal} < f_l$) then
    SetFreq($f_l$)
end if
```

Of these, SetFreq() is a wrapper of CPU/GPU DVFS APIs that set specific CPU/GPU frequencies and GetCurFreq() is used to inquire the current frequency in use. The other three functions are more complex and will be detailed next.

These three functions use prior knowledge of the mapping between frequency and execution time for the CPU and GPU tasks. Specifically, GreenLA records runtime of the first CPU-GPU tasks at different frequencies by instrumenting timestamps in the source codes, and use them to estimate the runtime and slack for the rest of the CPU-GPU tasks

**Algorithm 3** *Relaxed CP-aware Slack Reclamation*

```
CP_RSR($slack$, $fSet$)
1: if ($f_l \leq f_{ideal} \leq f_h$) then
2:   if ($f_{ideal} \notin fSet$) then
3:     $r \leftarrow$ GetApproxRatio($T_{lower}$, $T_{upper}$, $slack$)
4:     if ($r < RlxFctr$) then
5:       if ($LastFreq \neq f_{upper}$) then
6:         SetFreq($f_{upper}$)
7:         $LastFreq = f_{upper}$
8:       else Do Nothing
9:     else SetFreq($f_{lower}$, $f_{upper}$, $r$)
10:  else SetFreq($f_{ideal}$)
11: else if ($f_{ideal} < f_l$) then
12:   SetFreq($f_l$)
13: end if
```

using Equations 5.1-5.6. Given $T$, $T_{lower}$, $T_{upper}$, and $slack$ of each pair of CPU-GPU tasks, we split frequency with ratio $r$ and the ideal frequency $f_{ideal}$ can be solved as follows:

$$\begin{aligned}
&T + slack = T_{lower} \times r + T_{upper} \times (1 - r) \\
&f_{ideal} \times (T + slack) = f_h \times T \\
&\text{where } T \mapsto f_h, T_{lower} \mapsto f_{lower}, T_{upper} \mapsto f_{upper} \\
&\text{subject to } f_l' \leq f_{lower} < f_{ideal} < f_{upper} \leq f_h
\end{aligned} \tag{4.9}$$

The resulting target frequency $f_{ideal}$ is compared against the available frequencies (i.e., line 2 in Algorithm 2). If it matches an available frequency, the matched available frequency can be used directly. Otherwise, neighboring frequencies $f_{lower}$ and $f_{upper}$ are assigned in accordance with the ratios $r$ and $1 - r$ individually. In case that $f_{ideal}$ is lower than the lowest available frequency, the lowest available frequency is adopted, as sketched in Algorithm 2.

### 4.4.2 Relaxed Slack Reclamation

In contrast to strict slack reclamation that applies DVFS at each iteration of the algorithms, relaxed slack reclamation forms multiple iterations into groups and apply DVFS at the group level. Relaxed slack reclamation offers two advantages. First, it reduces the time and energy overhead incurred by frequent DVFS scheduling [158]. Second, it reclaims extra slack on the CPU and GPU caused by data dependencies, in addition to the slack caused by workload imbalance that strict reclamation targets. In each iteration, the GPU waits for the factorized panel from the CPU, and the CPU waits for the updated panel from the GPU. The slack is reclaimed by relaxed slack reclamation but not by strict slack reclamation.

We use a *relaxation factor* ($RlxFctr$ in Algorithm 3) to determine the number of iterations in a group for scheduling decisions. As shown in Algorithm 3, if the calculated split frequency ratio $r$ is less than $RlxFctr$ (e.g., 0.05), the duration at $f_{lower}$ is negligible according to Equation 4.9. In this case, we run the processing units at $f_{upper}$. The selection of $RlxFctr$ is based on algorithmic characteristics. Slack varies with iterations and the variation rate provides us the criteria of choosing an appropriate $RlxFctr$ for the optimized energy efficiency. Note that even with a constant $RlxFctr$, the number of iterations in a group may vary over time during a run.

### 4.4.3 Coupled GPU Core and Memory DVFS

DVFS can be applied to power-scalable hardware components, including CPU, GPU, and memory. It is noteworthy that on today's architectures such as GPU, core and memory frequencies are coupled and have to be switched simultaneously as a combination

**Algorithm 4** *GPU Core/Memory DVFS Scheduling*

```
GPU_DVFS($slack$, $fSet$)
1: if ($f_l' \le f_{ideal} \le f_h$) then
2:     Call CP_SSR($slack$, $fSet$) or CP_RSR($slack$, $fSet$)
3: else if ($f_l \le f_{ideal} < f_l'$) then
4:     $r \leftarrow$ GetApproxRatio($T_l$, $T_l'$, $slack$)
5:     SetFreq($f_l$, $f_l'$, $r$)
6: else if ($f_{ideal} < f_l$) then
7:     SetFreq($f_l$)
8: end if
```

[24], which differs from CPU DVFS where only core frequency is scaled. Table 4.2 lists memory-core frequency pairs for two NVIDIA GPU.

Table 4.2: GPU Mem.-Core Freq. Pairs (Unit: MHz).

| NVIDIA Kepler Tesla K20c | | NVIDIA Kepler Tesla K40c | |
|---|---|---|---|
| Memory Freq. | Core Freq. | Memory Freq. | Core Freq. |
| 2600 | 758<br>705<br>666<br>640<br>614 | 3004 | 875<br>810<br>745<br>666 |
| 324 | 324 | 324 | 324 |

As discussed earlier, in our scenario, slack may occur either on the CPU side or on the GPU side. Algorithm 1 strategically makes CPU/GPU DVFS decisions depending on the source of slack. In particular, for the case of eliminating slack from GPU side, due to the coupled core and memory frequencies of GPU, a combined GPU core/memory DVFS scheduling strategy is necessary. Line 3-7 in Algorithm 4 details the combined strategy, where split frequency ratio $r$ is calculated similarly as Equation 4.9. Equation 4.10 shows the calculation, and the difference is that scaling down to $f_l$ pairs with memory frequency reduction to the lowest.

$$\begin{aligned}
&T + slack = T_l \times r + T_l' \times (1 - r) \\
&f_{ideal} \times (T + slack) = f_h \times T \\
&\text{where } T \mapsto f_h, T_l \mapsto f_l, T_l' \mapsto f_l' \\
&\text{subject to } f_l \leq f_{ideal} < f_l'
\end{aligned} \tag{4.10}$$

In our experiments, GPU tasks that update the trailing matrices involve considerable computation and memory accesses. Therefore simultaneously decreasing GPU core/memory frequencies has dual performance impact on computation and memory accesses. Our approach takes care of these scenarios since the dual slowdown has been recorded in $T'$, which is the runtime of a task at the lowest core and memory frequencies.

## 4.5 Evaluation

In this section we detail the evaluation of GreenLA on a GPU-accelerated heterogeneous system: a linear algebra library of dense matrix factorizations (Cholesky, LU and QR) with an energy efficient CPU/GPU DVFS co-scheduling approach via online algorithmic slack prediction.

### 4.5.1 Evaluation Methodology

For comparison purposes, we present a state-of-the-art OS level method for slack prediction, and another type of classic DVFS scheduling strategy for saving energy. We stress the difference in other approaches against ours, and argue that our solution can outperform in both the accuracy of slack prediction and the amount of energy savings in our scenario.

*OS Level Slack Prediction*. As another important slack prediction method, *OS level slack prediction* either work for a specific type of applications sharing similar features, e.g., with stable/slowly-varying execution characteristics, or require considerable training to obtain accurate prediction results. Online prediction mechanism presented in [128] [145] [144] is based on a simple assumption that task behavior is identical every time a task is executed. It is however defective for applications with variable workloads, such as matrix factorizations, where the remaining unfinished matrices become smaller as the factorizations proceed. Execution time shrinks and slack varies as the workloads become lighter, which invalidates the above prediction mechanism.

Regardless of the simplest prediction above, several enhanced history-based workload prediction algorithms have been proposed to handle the variation in HPC runs and produce more accurate prediction results [175] [62] [99] [86]. The RELAX algorithm employed in CPU MISER [86] exploits both prior predicted profiles and current runtime measured profiles: $W'_{i+1} = (1-\lambda)W'_i + \lambda W_i$, where $\lambda$ is a relaxation factor for adjusting the percentage of dependent information on the current measurement. This enhanced prediction can also be error-prone for dense matrix factorizations, since using a fixed $\lambda$ cannot handle length variation of iterations of the core loop due to the shrinking remaining unfinished matrices. The use of 2-D block cyclic data distribution further brings complexity to the prediction. Moreover, statistical predictive models have been adopted for accurate workload prediction, e.g., Hidden Markov Models (HMM) used in [186] and Predictive Bayesian Network (PBN) used in [125]. Using offline training and learning based on historical records, results with high accuracy were achieved (average prediction error 3.3% via HMM and 0.43% via

PBN). Although effective offline, online slack prediction for HPC applications using statistical models can be costly: Considerable amount of execution traces are required to train the statistical predictive models for accurate slack prediction. For instance, the training dataset in [186] was obtained by running applications on one server and evaluated on one different server, which can be impractical for HPC runs as discussed earlier.

*Race-to-halt Energy Saving*. As the name suggests, *race-to-halt* (or *race-to-idle*) is an energy saving strategy that enforces power-scalable processors (e.g., CPU and GPU) to *race* when workloads are ready for processing, and to *halt* when no tasks are present and the processors are idle/waiting. In other words, *race* refers to executing the workloads at the highest frequency and voltage of the processors for the peak performance until the finish of the workloads, and *halt* implies that processor frequency and voltage are switched to the lowest level from the end of the last executed workload to the start of the next workload. This straightforward solution can effectively save energy without incurring performance loss due to the following inferences: (a) The peak performance of processors is guaranteed during computation as in original runs; (b) the peak performance of processors is not needed when no tasks are being executed and processors are waiting for data. As discuss earlier, in our scenario, slack can arise at either CPU side or GPU side, depending on various factors. In either case, the peak performance of the idle/waiting processors is not necessary. Per *race-to-halt*, we apply to them the lowest power state during the slack and switch back to the highest one until the next workload is available. *race-to-halt* is *CP-free* such that no CP detection is required before any energy saving decisions are made. Thus it is generally lightweight and easy to implement.

We implemented OS and library level approaches using *race-to-halt* and online HMM-enabled statistical slack prediction individually. Further, we implemented relaxed slack reclamation to compare with the default strict slack reclamation. Evaluated metrics include slack prediction accuracy, and energy and performance efficiency. For readability, we henceforth denote different test cases as follows:

- `MAGMA`: The original MAGMA runs of different-scale CPU-GPU Cholesky, LU and QR factorizations without any energy saving approaches;
- `OS_r2h`: The OS level implementation [6] based on a CPU race-to-halt workload prediction algorithm similar to the RELAX algorithm;
- `lib_r2h`: The library level implementation based on algorithmic race-to-halt on both CPU and GPU;
- `OS_cpsr_str`: The OS level implementation based on online HMM-enabled statistical slack prediction, with strict slack reclamation for each iteration;
- `OS_cpsr_rlx`: The OS level implementation based on online HMM-enabled statistical slack prediction, with relaxed slack reclamation ($RlxFctr = 0.05$) for blocked iterations;
- `lib_cpsr_str`: The library level implementation based on online algorithmic slack prediction, with strict slack reclamation for each iteration;
- `lib_cpsr_rlx`: The library level implementation based on online algorithmic slack prediction, with relaxed slack reclamation ($RlxFctr = 0.05$) for blocked iterations.

Among all energy saving solutions, `lib_cpsr_rlx` empirically achieves the optimal energy efficiency with negligible performance loss, and thus we adopt it as our `GreenLA` in the comparison against MAGMA later.

### 4.5.2 Experimental Setup

We applied all above test scenarios to CPU-GPU Cholesky, LU and QR factorizations (MAGMA version 1.6.1) with multiple global matrix sizes each (ranging from 5120 to 20480). However, due to limit space, we only show the result for input size of $20480 \times 20480$. For other input matrix sizes, the results are similar. All experiments were performed on a power-aware many-core CPU-GPU server. Table 5.3 lists hardware configuration of the experimental platform. The total system dynamic and static/leakage energy consumption of the above runs was measured using nvidia-smi tool [24] provided by NVIDIA, and following [131], we used PowerPack [87], an integrated software/hardware framework for profiling and analysis of power/energy costs of HPC systems and applications. A separate meter node with PowerPack deployed was used to collect power/energy costs of all hardware components of the system, and the data was recorded in a log file and accessed after the above runs.

## 4.6 Results

Next we present experimental results of our evaluation via fine-grained comparison. We first demonstrate the performance and energy efficiency of our approach by comparing to the widely used numerical linear algebra MAGMA library.

Table 4.3: Hardware Configuration for Experiments.

| Component | CPU | GPU |
|---|---|---|
| Processor | 2×10-core Intel Xeon Ivy Bridge E5-2670 | 2496 CUDA-core NVIDIA Kepler GK110 Tesla K20c |
| Peak Perf. | 0.4 TFLOPS | 1.17 TFLOPS |
| Core&Mem. Freq. Gear | Core:1.2-2.5(↑by0.1)GHz<br>Mem.:Not DVFS-capable | See left column of Table 4.2 |
| Memory | 64 GB RAM | 5 GB RAM |
| Cache | 64 KB L1, 256 KB L2, 25.6 MB L3 | 13 SMX units, 64 KB and 48 KB read-only d-cache |
| OS | Fedora 21, 64-bit Linux kernel 3.17.4 | |
| Pwr. Meter | PowerPack | nvidia-smi with -ac option |

Table 4.4: Average Error Rates of Slack Prediction for Four Runs Each of Cholesky/LU/QR Factorization.

| Benchmarks & Test Scenarios | OS Level Statistical Slack Prediction | | Library Level Algorithmic Slack Prediction |
|---|---|---|---|
| | Base Iter. (First 10%) | Base Iter. (First 20%) | |
| Cholesky (5120 - 20480) | 10.51% | 6.62% | 0.96% |
| LU (5120 - 20480) | 9.95% | 5.45% | 0.16% |
| QR (5120 - 20480) | 11.29% | 5.77% | 0.52% |

### 4.6.1 Average Error Rate of Slack Prediction

We first showcase the accuracy of slack prediction of the OS level statistical approach with two training datasets and our library level algorithmic approach. Table 4.4 summarizes the average error rate of slack prediction of the two approaches. As stated in section 4.3, HMM-based statistical slack prediction requires a group of *base* iterations (usually the first few iterations of a HPC run) to serve as an online training dataset. The prediction accuracy is highly associated with the size of the training dataset according to Table 4.4: The more *base* iterations are used, the more accuracy is achieved. However, the highest accuracy of the OS level approach is 5.45% for LU, while our library level approach can be as accurate as having a 0.16% error rate for LU. This low error rate of slack prediction can greatly facilitate forthcoming energy saving. For further comparison, we select the OS level statistical approach with higher slack prediction accuracy for more experiments.

### 4.6.2 Total Energy Saving Comparison

As shown in Figure 4.4, our library level CP-aware slack reclamation approaches could save more energy than current state-of-the-art approaches. Current approaches could only either save less energy or even costs extra energy. For example, OS level race-to-halt only slows down CPU when CPU utilization is below a threshold, while library level race-to-halt reduces both CPU and GPU speed when no corresponding workloads are running according to algorithmic characteristics. Due to high online probing overhead, the OS level race-to-halt approach incurs even more energy consumption, while the more lightweight library level race-to-halt approach can save minor energy savings (up to 3.6%) as shown in Figure 4.4. Other two approaches we compared are OS level approaches statistical slack prediction `OS_cpsr_str` and `OS_cpsr_rlx`. Since the two solutions produced inaccurate slack prediction, the inaccuracy results in inappropriate timing and duration of DVFS, which cannot eliminate possible slack – saving less energy than the optimal, or incurs performance loss due to overdue or overdone DVFS – consuming even more energy than the original run. As shown in Figure 4.4, those two approaches consumed more energy(1%–2%). Even if the OS level slack prediction can achieve the same accuracy as our library level approaches, the OS level solutions can waste much more energy saving opportunities than our library level approaches due to the considerable amount of iterations used for training, compared to only execution information of the first iteration needed by our library approach. On the other hand, our library level CP-aware slack reclamation approaches could save several times more energy. specifically, the energy saved from our approach is 2.5x of the energy saved using current best approach in Cholesky factorization, 1.5x in LU factorization and 3x in QR factorization. Moreover, different than our strict slack reclamation, which tries to reclaim all slacks using DVFS, our relaxed slack reclamation only tries to apply DVFS when split frequency ratio is larger than the $RlxFctr$, which eliminated some unnecessary power state adjustments. The reduced number of power adjustment brings less DVFS performance overhead, which further saves more energy for the overall application. Note that, the Cholesky, LU and QR factorizations are very compute intensive. Based on the energy efficiency model in [61], their heterogeneous implementations in MAGMA have really high computation efficiency, which make them hard to save more energy. Although there are many hardware components involve the execution process, we only focus on reducing the energy consumption of CPU and GPU.

### 4.6.3 CPU Energy Saving Comparison

Now, we focus on the energy saving on the CPU side. Note that since we only focus on reducing the energy cost of CPU, the energy measurement here does not include RAM energy consumption. We can see from Figure 4.5 the single core CPU energy comparison of Cholesky, LU and QR factorization using different energy saving solutions. As mentioned before, OS level race-to-halt only adjust the performance/power state of the CPU, which also introduce more probing overhead, it cost more energy on the CPU side (7%–8%). As for the OS level online HMM-enabled statistical slack prediction approaches, they need first 10%–20% iterations to do online training on the CPU, which not only brings more overhead, but also wastes valuable slack reclamation(energy saving) opportunities. However, thanks to

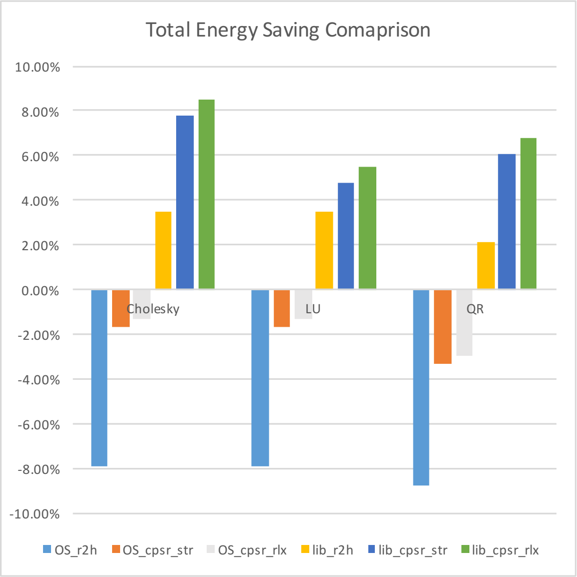


Figure 4.4: The total amount of energy saved in percentage using several different energy saving solutions. Right two bars in each group show the result of our approach, which can save up to 3x more energy than the current best solution.

the high accurate algorithmic slack prediction, our library level CP-aware slack reclamation approaches could save more energy on the CPU side when the slack resides on CPU.

### 4.6.4 GPU Energy Saving Comparison

Next, we focus on the energy saving on the GPU side. As we can see from Figure 4.6, even GPU is assigned more computation tasks, it usually finish its tasks faster than the CPU, so slacks are more likely to occur on the GPU side, and thus it can save more energy. For library level race-to-halt approach, since it rely on algorithmic execution time prediction, it can save energy to some degree. But it still waste some energy in ”halt” state, so it saves less energy than our approach. As for OS level race-to-halt, it does not adjust the performance/power of the GPU at all and poor CPU side adjustment brings more performance overhead, so the overall execution time is prolonged, which results higher GPU

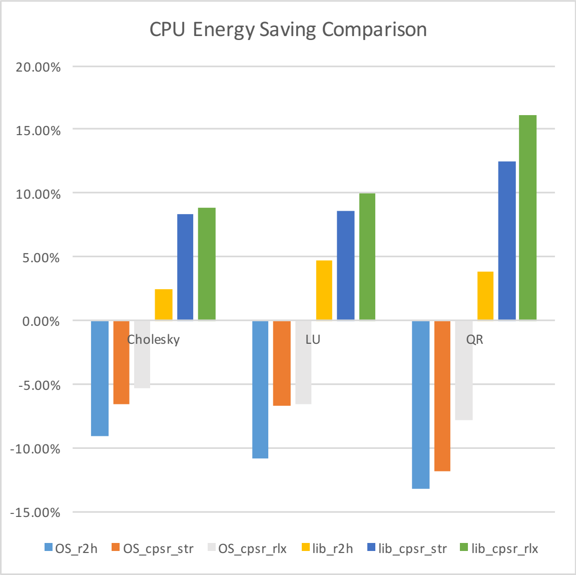


Figure 4.5: The amount of energy saved in percentage on single core CPU using several different energy saving solutions. Positive values indicate energy saving. Negative values indicate extra energy cost. Our approaches(right two bars in each group) shows more energy saving on CPU than existing state-of-the-art solutions.

energy consumption. Similar as on the CPU, OS level online prediction approach suffers from inaccurate slack prediction, which leads to higher GPU energy consumption. Our approaches, on the other hand, could save up to 16% energy on the GPU.

### 4.6.5 Time Overhead

All kinds of performance loss is observed from the experiments as shown in Figure 4.7. Typical performance degrading factors for OS level solutions consist of dynamic monitoring overhead (`OS_r2h`), online training overhead (`OS_cpsr_str` and `OS_cpsr_rlx`), DVFS overhead (all solutions), and performance loss from overdue or overdone DVFS due to inaccurate slack prediction (`OS_cpsr_str` and `OS_cpsr_rlx`). On the other hands, observed performance loss for library level solutions is from frequency approximation errors

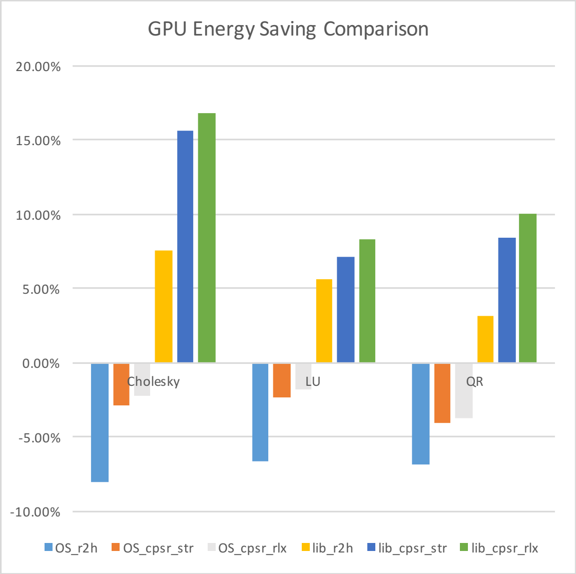


Figure 4.6: The amount of energy saved in percentage on GPU using several different energy saving solutions. Positive values indicate energy saving. Negative values indicate extra energy cost. Our approaches(right two bars in each group) shows more energy saving on GPU than existing state-of-the-art solutions.

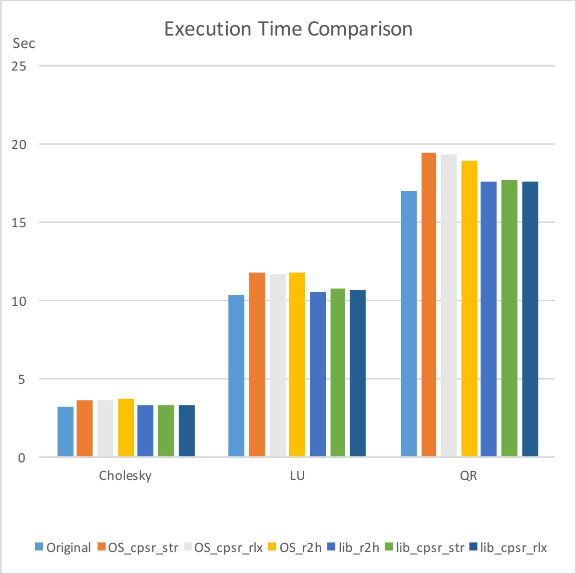


Figure 4.7: Execution time of Cholesky, LU and QR factorization using several different energy saving solutions. Right two bars in each group show our results, which have similar performance than existing state-of-the-art solutions.

and DVFS overhead. Among all approaches, `OS_cpsr_str` has the highest performance loss (up to 14.4%, due to overdone DVFS from inaccurate slack prediction), while our `lib_cpsr_str`/`lib_cpsr_rlx` incurs minor performance loss (as low as 1.2%).

## 4.7 Summary

Energy efficiency is becoming a critical factor of concern when achieving parallelism in high performance scientific computing in this era. The growing prevalence of heterogeneous architectures nowadays brings more concerns on saving energy for the emerging systems. Essentially fulfilling energy efficiency requires accurate slack prediction with minor performance degradation. Existing energy efficient approaches span from OS level to application level, which can either be inaccurate or cost-inefficient due to variable execution patterns of the target applications and lengthy training of the employed prediction model. In this paper, we propose a lightweight energy efficient approach for widely used numerical linear algebra software that utilizes algorithmic characteristics to obtain accurate slack prediction and thus gain the optimal energy savings. Experimental results on a many-core CPU-GPU platform demonstrate that our library level solution can achieve up to 8.5% energy saving than original implementation with negligible performance loss (as low as 1.2%), which 3x more energy savings compared to classic race-to-halt and workload prediction approaches.

Although the currently achieved energy savings are moderate, provided a limited amount of slack for the target applications, more energy can be saved by reducing the minor performance loss incurred by our approach. It is practical and worthwhile since careful and fine-grained DVFS analysis is able to further decrease the number of DVFS switches and errors of frequency approximation. Possible energy savings can also be obtained from improved application characteristics that facilitate power reduction, such as CPU workload centralization and idle/unused core isolation, etc. We are also interested in investigating the energy impact of matrix factorization block sizes. It is possible that the optimal block size for performance differs from the optimal block size for energy costs. There may exist a trade-off between them. We further plan to extend the work to more scientific applications on other emerging hardware and architectures in the near future.

# Chapter 5

# Energy Efficient and Fault Tolerant One-sided Matrix Decompositions with Bi-directional Algorithmic Slack Reclamation and ABFT

## 5.1 Introduction

In previous chapters, we introduced several effective approaches for optimizing energy efficiency [54] and reliability [50, 52] for one-sided matrix decompositions on heterogeneous systems with GPUs. However, none of the previous work is able to optimize both at the same time. One major challenge is that many energy efficiency and fault tolerance optimizations are not compatible with each other. For example, fault tolerance approach such as ABFT brings performance overhead, which in turns decrease energy efficiency. Energy saving approach as such overclocking or undervolting can decrease system reliability and can cause serious error propagation in matrix decompositions, which is beyond tolerable for ABFT.

In this work, we propose PowerLA, enhanced one-sided matrix decompositions that are both energy efficient and fault tolerance. Specifically, our contributions are listed as follows:

- We extend clock frequency range used in DVFS based slack reclamation energy-saving approach to further include overclocking frequencies. This enables us to get more energy saving.
- We carefully design an optimization to incorporate overclocking and ABFT to ensure execution correctness.
- We propose a novel bi-directional slack reclamation, which allows slacks to be reclaimed by both processors at the same time. This greatly improves flexibility when reclaiming slacks.
- We implement our optimization on three core one-sided matrix decompositions and evaluate our implementation on our energy aware heterogeneous computing systems with GPUs. Results show that our proposed work can obtain more energy saving with fault tolerance capability at the same time.

## 5.2 PowerLA Design

### 5.2.1 Challenges of Building Fault Tolerant and Energy Efficient Matrix Decompositions

As mentioned before, both reliability and energy efficiency are important for one-sided matrix decompositions on heterogeneous systems with GPU. However, many of the approaches in energy saving and fault tolerance are not naturally compatible with each

Table 5.1: Notation in algorithms and formulations in this chapter

| | |
|---|---|
| $T_k^{OP}$ | Prediced/measured execution of $OP$ in $k^{th}$ iteration. |
| $slack_k$ | Slack length in $k^{th}$ iteration. |
| $PD$ | Panel Decomposition. |
| $PU$ | Panel Update. |
| $TMU$ | Trailing Matrix Update. |
| $CHK\_UPD$ | Checksum update. |
| $CHK\_VRF$ | Checksum verification. |
| $f_{BASE}^{DEVICE}$ | The baseline clock frequency on $DEVICE$. |
| $f_{MIN}^{DEVICE}$ | The minimum clock frequency on $DEVICE$. |
| $f_{MAX\_CORRECT}^{DEVICE}$ | The maximum clock frequency on $DEVICE$ that does not affect execution correctness. |
| $f_{SAFE}^{DEVICE}$ | The maximum clock frequency on $DEVICE$ that does not cause system/process crash. |
| $P_{dyn\_base}^{DEVICE}$ | The dynamic power of $DEVICE$ at $f_{BASE}^{DEVICE}$ clock frequency. |
| $nb$ | Matrix blocks size ($nb \times nb$). |

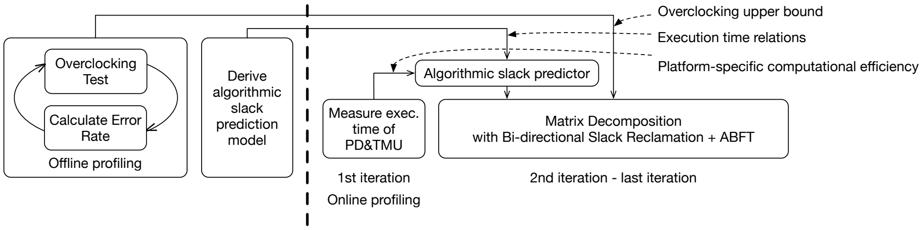

Overclocking upper bound
Execution time relations
Platform-specific computational efficiency
Overclocking Test
Calculate Error Rate
Offline profiling
Derive algorithmic slack prediction model
Algorithmic slack predictor
Measure exec. time of PD&TMU
Matrix Decomposition with Bi-directional Slack Reclamation + ABFT
1st iteration
2nd iteration - last iteration
Online profiling

Figure 5.1: Overview of PowerLA

other. For example, many works have been done to utilize slacks to improve the energy efficiency of applications such as race-to-halt and slack reclamation-based energy-saving approaches [105, 143, 165, 54]. They have been proven that their proposed approaches can save considerable energy compared with the original design without any energy saving optimizations. However, none of them can be integrated with fault tolerance techniques, such as ABFT without brings more energy cost or performance overhead. The key reason is that fault tolerance needs to be added to tasks that are on both critical and non-critical path, which prolong the total execution time and further makes an impact on energy and performance. On the other hand, other energy saving approach as such overclocking or undervolting can enable more control on energy saving and performance, but they can also decrease system reliability, which can cause serious error propagation in matrix decompositions making them beyond tolerable for ABFT if not carefully tuned.

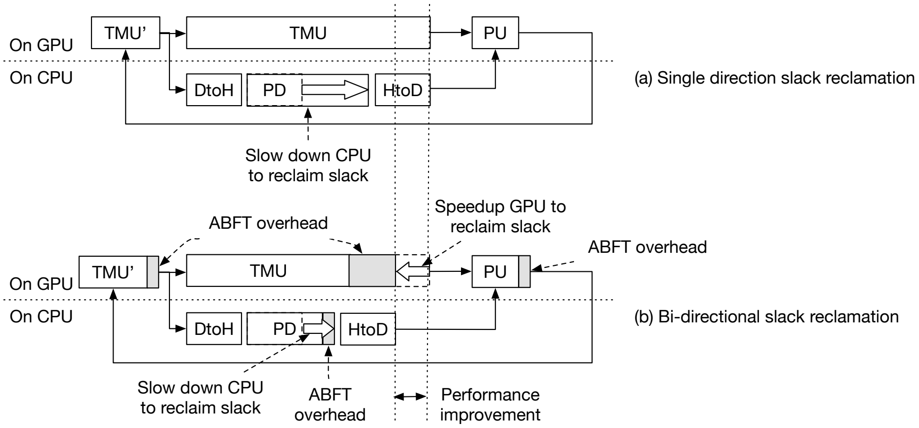

TMU'
TMU
PU
On GPU
On CPU
DtoH
PD
HtoD
(a) Single direction slack reclamation
Slow down CPU to reclaim slack
Speedup GPU to reclaim slack
ABFT overhead
ABFT overhead
TMU'
TMU
PU
On GPU
On CPU
DtoH
PD
HtoD
(b) Bi-directional slack reclamation
Slow down CPU to reclaim slack
ABFT overhead
Performance improvement

Figure 5.2: Single direction slack reclamation vs. Bi-direction slack reclamation.

### 5.2.2 Bi-directional Slack Reclamation: Bringing Fault Tolerance and Energy Saving Together

In this work, we aim to build one-sided matrix decompositions that are both fault tolerant and energy efficient. The core part of our work is the novel bi-directional slack reclamation. Similar to original slack reclamation proposed in previous works, our bi-direction slack reclamation also aims to minimize slacks by adjusting the task execution length. However, unlike previous works, which only adjust tasks on non-critical paths towards one direction, our bi-directional slack reclamation (**Fig. 5.2**) adjust tasks on both critical and non-critical path in two directions (i.e., speed up or slow down). Since tasks are already running on the highest processor power state, speedup tasks furthermore usually are not applicable since most DVFS strategies do not allow setting clock frequencies beyond the highest. For our bi-directional slack reclamation, to enable more flexible adjustment we allow both DVFS and overclocking as available power state selections for slack reclamation.

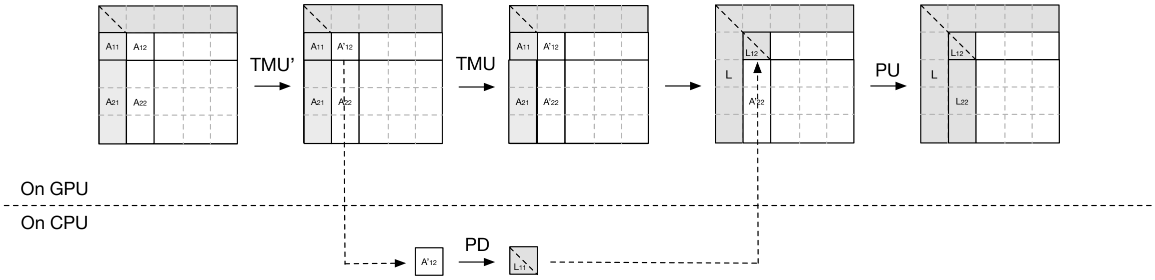


Figure 5.3: Workflow of Cholesky Decomposition.

In this way, we can allow tasks execution length to be adjusted in both directions even if it was already running on the high power state. Compared with original slack reclamation that was introduced in previous works, our bi-directional slack reclamation brings two major advantages: (1) It enables us to speed up tasks on critical path, which brings potential performance improvement for the entire execution of matrix decompositions that can potentially allow us to add fault tolerance without bringing high fault tolerance overhead; (2) Bi-directional slack reclamation brings more energy saving than single direction slack reclamation since reclaiming slacks in two directions gives more flexibility, which allows us to save energy on both CPU and GPU when reclaiming a single slack.

Bi-directional slack reclamation cannot work by itself to enable fault tolerant and energy saving for matrix decomposition. It needs to work with three other key components that will be discussed in the following sections: slack identification and prediction (section 5.5), bi-directional slack reclamation strategies (section 5.4), and overclocking and fault tolerance integration (section 5.5 and 5.5.2).

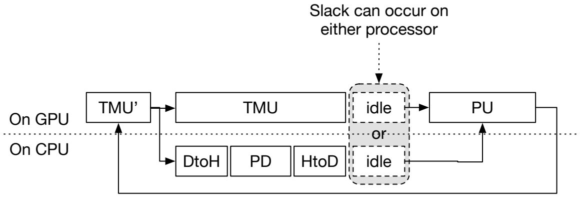
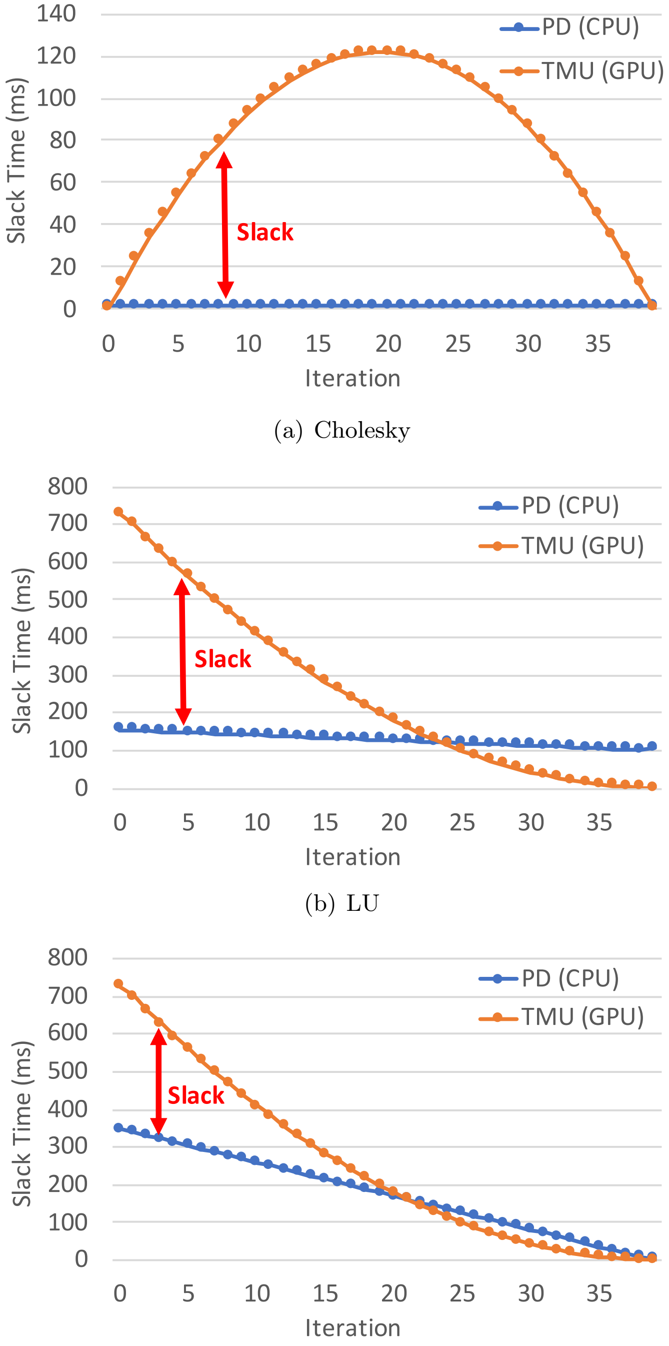


Figure 5.4: Slack can occur on either side during matrix decompositions.

## 5.3 Slacks in One-sided Matrix Decompositions on Heterogeneous Systems with GPUs

To enable slack reclamation, we need to identify slacks in matrix decompositions first. To utilize the concurrent execution feature between CPUs and GPUs, in the current design of MAGMA, part of the workload of TMU that PD will depends on later is done in ahead of rest part. In this way, PD on CPU can be done in concurrent with the rest part of TMU. The execution overlap between CPU and GPU can potentially benefit the overall performance of matrix decompositions.

Figure 5.10 demonstrates how a dense matrix decomposition proceeds on a CPU-GPU platform with data movement between CPU and GPU in a local view. As mentioned, decomposing the panel matrices is executed on the CPU; and updating the trailing matrix is massively parallelized on the GPU. The *panel matrices* calculated on the CPU are offloaded to the GPU and used by the GPU to update the trailing matrices. For the sake of performance, the next panel matrix that is updated on the GPU is immediately copied back to the CPU before the entire trailing matrix finishes. As such, panel decomposition is simultaneously executed on the CPU as the rest of trailing matrix is updated on the GPU.



(a) Cholesky

(b) LU

(c) QR

Figure 5.5: Slacks in Matrix Decompositions (Matrix size: 10240*10240)

These processes proceed by a sub-matrix block, starting from the left upper corner of the global matrix and finishing when the whole global matrix is fully decomposed.

Since the workload and the computation power of CPU and GPU can vary, they may finish assigned computation tasks at different times. With implicit synchronizations (i.e., data copies between CPU and GPU), idle may occur to processors that finish earlier. This kind of idle time is called slack. **Fig. 5.5** show how slacks change as matrix Cholesky, LU, and QR decompositions make processes on our test platform. Slack brings potential energy waste if not carefully taken care of.

### 5.3.1 Slack Prediction

To reclaim slack, we first need to know where slacks can occur. We have examined the workflow of matrix decompositions in state-of-the-art MAGMA library and figure out that slack can occur to either CPU and GPU after panel decompositions and trailing matrix update. In addition, we also need to know when slacks occur and how long they can last. One method to obtain such knowledge is to instrument the source code with timing functions and run the instrumented program with various problem sizes to collect the profiles. Alternatively, OS-level profiling can be performed with hardware performance counters on processors. Neither method is portable, and both methods require extensive profiling. In this work, we propose an algorithmic slack prediction and clock frequency adjustment strategy similar to our previous work [54].

Given the heterogeneous matrix decomposition algorithms, the slack on the CPU and GPU is mainly impacted by the software and hardware parameters.

- *Problem size*: the sizes of the panel sub-matrix and trailing sub-matrix scheduled on the CPU and GPU respectively based on the algorithmic characteristics;
- *CPU compute capacity*: the number of floating point operations the CPU is able to perform in one second for the assigned tasks;
- *GPU compute capacity*: the number of floating point operations the GPU is able to perform in one second for the assigned tasks.
- *Data transfer speed*: the number of bytes the GPU are able to transfer between CPU memory and GPU memory.

Our algorithmic slack prediction aims to leverage algorithmic knowledge to accurately predict the slack on CPU/GPU. The algorithmic knowledge that we need to consider for matrix decompositions including CPU/GPU execution time and data copy time.

We first focus on the execution time. Assuming the execution time of the first iteration of a $N \times N$ matrix with block size $nb$ decomposition are known, we denote the CPU time for the panel decomposition as $T_0^{PD}$, and the GPU time for trailing matrix updating as $T_0^{TMU}$. We can use the algorithmic information to predict the execution time of the remaining iterations. For now, we consider fixed compute efficiency for the CPU and GPU. As the time complexities of panel decomposition and trailing matrix updating are $O(N^3)$ for a matrix size $N$, the execution time for iteration $k$ and $k+1$ have the following relation.

Table 5.2: The ratio of execution time and data transfer time of panel decompositions and trailing matrix update between $k^{th}$ and $k+1^{th}$ iterations of the three core matrix decompositions.

| | Computation & Checksum Update | Data Transfer (CPU$\Longleftrightarrow$GPU) | Checksum Verfication |
|---|---|---|---|
| PD-Cholesky | $1$ | $1$ | $1$ |
| TMU-Cholesky | $(1+k)(1-\frac{1}{B-k-1})$ | N/A | $1-\frac{1}{B-k-1}$ |
| PD-LU | $1-\frac{1}{B-k}$ | $1-\frac{1}{B-k}$ | $1-\frac{1}{B-k}$ |
| TMU-LU | $1-\frac{1}{B-k}-\frac{1}{B-k-1}+\frac{1}{B^2-(2k+1)B+k(k+1)}$ | N/A | $1-\frac{1}{B-k}-\frac{1}{B-k-1}+\frac{1}{B^2-(2k+1)B+k(k+1)}$ |
| PD-QR | $1-\frac{1}{B-k}$ | $1-\frac{1}{B-k}$ | $1-\frac{1}{B-k}$ |
| TMU-QR | $1-\frac{1}{B-k}-\frac{1}{B-k+1}+\frac{1}{B^2+(2k+1)B+k(k+1)}$ | N/A | $1-\frac{1}{B-k}-\frac{1}{B-k+1}+\frac{1}{B^2+(2k+1)B+k(k+1)}$ |

$$\frac{T_{k+1}^{PD}}{T_k^{PD}} = \frac{O(N_{k+1}^3)}{O(N_k^3)} \tag{5.1}$$

$$\frac{T_{k+1}^{TMU}}{T_k^{TMU}} = \frac{O(N_{k+1}^3)}{O(N_k^3)} \tag{5.2}$$

Based on the design of Cholesky, LU, and QR decomposition on the heterogeneous system with GPU [17], we are able to derive the theoretical ratio of the execution time of panel decompositions and trailing matrix update. As shown in **Table 5.2**, we derive the ratio between two neighbor iteration, where $B = \frac{n}{nb}$. We can see that most of the ratios are not constant and their values not only depend on application settings (i.e., $nb$) but also vary between different iterations (i.e., depend on $k$). So, it is very hard to capture this information from system-level efficiently.

Similarly, we can also predict the data copy time between CPU and GPU. Assuming the data copy time of the first iteration is $T_0^{COPY}$ for transferring a panel to GPU from CPU and then transfer it back. Assuming the data transfer speed is constant, we can use $T_0^{COPY}$ and the size change of panel over iterations to predict future data copy time. The data copy time for iteration $k$ and $k+1$ have the following relation. We also show that in **Table 5.2**.

$$\frac{T_{k+1}^{COPY}}{T_k^{COPY}} = \frac{O(N_{k+1}^2)}{O(N_k^2)} \tag{5.3}$$

Finally, we also prediction the time overhead brings by ABFT in a similar way. The ABFT overhead usually comes with two parts: updating checksum, and verifying checksum. Since their algorithms are usually different, we use two different relation equations to do prediction. Assume the execution time of checksum updating operation and checksum verification operation are $T_0^{CHK\_UPD}$ and $T_0^{CHK\_VRF}$. The data copy time for iteration $k$ and $k+1$ have the following relation. We also show that in **Table 5.2**.

$$\frac{T_{k+1}^{CHK\_UPD}}{T_k^{CHK\_UPD}} = \frac{O(N_{k+1}^3)}{O(N_k^3)} \tag{5.4}$$

$$\frac{T_{k+1}^{CHK\_VRF}}{T_k^{CHK\_VRF}} = \frac{O(N_{k+1}^2)}{O(N_k^2)} \tag{5.5}$$

Since, CPU must wait for data copy is done before it can starts it computation, the slack during iteration $k$ can be quantified as follows:

$$slack_k = \left|\ T_k^{CPU} + T_k^{COPY} - T_k^{GPU}\right|;\ 0 \leq k < \frac{N}{nb} \tag{5.6}$$

where $T_k^{CPU} = T_k^{PD} + T_k^{CHK\_UPD} + T_k^{CHK\_VRF}$ and $T_k^{CPU} = T_k^{TMU} + T_k^{CHK\_UPD} + T_k^{CHK\_VRF}$. A positive value indicates that the GPU has slack time to wait for the CPU, while a negative value indicates that the CPU has slack to wait for the GPU. Value zero means that the CPU and GPU finish the current iterations at the same time. The initial slack is the time difference for the first iteration, i.e., $k = 0$.

$$slack_0 = \left| \, T_0^{CPU} + T_0^{COPY} - T_0^{GPU} \right| \tag{5.7}$$

From Equations 5.1-5.6, we can quantify the slack for the rest CPU-GPU tasks, provided with the execution time of the first iteration. This algorithmic slack prediction is accurate and lightweight compared to OS level slack prediction. By using the prior algorithmic knowledge, only minimal profiling of the first iteration is necessary.

## 5.4 Slack Reclamation Strategies

### 5.4.1 Clock Frequency Adjustment

Once slacks are predicted the next step is to adjust the clock frequency of CPU and GPU to reclaim slacks. Since we adjust both CPU and GPU clock frequency to reclaim slack, we introduce term reclaim split ratios $r_{CPU}$ and $r_{GPU}$. It is used to decide how much of the slack is reclaimed by CPU or GPU. Since most matrix operations in matrix decompositions are computed intensive, we can assume that execution time is inversely proportional to the processor clock frequency: $T \propto \frac{1}{f}$, so we can derive the ideal clock frequencies for CPU and GPU used to reclaim the slack of length $slack_k$ at $k^{th}$ iteration with reclaim split ratio of $r_{CPU}$ and $r_{GPU}$.

$$f_{ADJUST}^{CPU} = \frac{T_k^{CPU}}{T_k^{CPU} + r^{CPU} \cdot slack_k} \cdot f_{BASE}^{CPU}$$

$$f_{ADJUST}^{GPU} = \frac{T_k^{GPU}}{T_k^{GPU} + r^{GPU} \cdot slack_k} \cdot f_{BASE}^{GPU}$$

In the next subsection, we will discuss the detail about how to adjust those two ratios to improve energy saving and performance.

### 5.4.2 Strict Slack Reclamation

In this subsection, we discuss how to adjust slack reclamation ratios to optimize energy saving. We introduce two strategies: strict slack reclamation and relaxed slack reclamation.

Our first slack reclamation strategy is strict slack reclamation. This strategy aims to reclaim the slack as much as possible to minimize processor idle cycles. In other word, our goal is to adjusting $r^{CPU}$ and $r^{GPU}$ such that their sum as close as possible to 100%. For example, assuming the slack is on the CPU side (similar for slack residing on the GPU side), the task on GPU is on the critical path. In this case, we can only adjust $r^{GPU}$ above zero (speedup). Otherwise, it may impact performance. $r^{CPU}$, on the other hand, can be adjusted both above zero (slow down) or below zero (speedup).

Note that $r^{CPU}$ and $r^{GPU}$ can only be adjusted in a way such that the resulting clock frequencies are available (within hardware allowable adjusting range) on target processors. Clock frequencies also need to be applicable, which means they need to be lower than $f_{max\_correct}$ if fault tolerance is not applied and lower than a certain frequency if fault tolerance is not applied (will be discussed later). In either case, we refer $r^{CPU}_{min/max}$ and $r^{GPU}_{min/max}$ as their range, which can be easily obtained from the applicable clock frequency range of CPU and GPU.

If $r^{CPU}_{max} + r^{GPU}_{max} < 100\%$, the slack cannot be completely reclaimed. In this case, the only choice would be having $r^{CPU} = r^{CPU}_{max}$ and $r^{GPU} = r^{GPU}_{max}$ to reclaim slacks as much as possible. The theoretical energy saving for $k^{th}$ iteration can be calculated as:

$$\Delta E = (P^{GPU}_{dyn\_base} + P^{GPU}_{static} + P^{CPU}_{static}) * r^{GPU}_{max} * slack_k + P^{CPU}_{dyn\_base} * T^{CPU}_k (1 - (\frac{f^{CPU}_{ADJUST}}{f^{CPU}_{BASE})}^{1.4})$$

The theoretical performance (execution time) improvement for $k^{th}$ iteration can be calculated as:

$\Delta T = r^{GPU}_{max} * slack_k$

If $r^{CPU}_{max} + r^{GPU}_{max} 100\%$, the slack can be completely reclaimed. In this case, set $r^{CPU} + r^{GPU} = 100\%$ to eliminate the whole slack, where $max(1 - rmax^{CPU}, r^{GPU}_{min}) r^{GPU} min(r^{GPU}_{min}, 1 - r^{CPU}_{min})$. The theoretical energy saving for $k^{th}$ iteration can be calculated as:

$\Delta E = (P^{GPU}_{dyn\_base} + P^{GPU}_{static} + P_{static})^{CPU} * r^{GPU} * slack_k + max(P^{CPU}_{dyn\_base} * T^{CPU}_k (1 - (\frac{f^{CPU}_{ADJUST}}{f^{CPU}_{BASE}})^{1.4}), P^{CPU}_{dyn\_base} * (100\% - r^{GPU}) * slack_k)$

To optimize energy saving, we can adjust $r^{GPU}$ and pick the value that leads to maximum energy saving. The theoretical performance (execution time) improvement for $k^{th}$ iteration can be calculated as:

$\Delta T = r^{GPU} * slack_k$

### 5.4.3 Relaxed Slack Reclamation

A more aggressive energy saving strategy is to release that requirement of reclaiming slacks as much as possible. In terms of tuning $r^{CPU}$ and $r^{GPU}$, this eliminates the restriction of $r^{CPU} + r^{GPU} = 100\%$. There is no guarantee that this strategy can lead to better energy-saving or performance, but relaxed slack reclamation does enable wider optimization search space for $r^{CPU}$ and $r^{GPU}$.

If choosing optimization in favor of energy saving, we can adjust $r^{CPU}$ and $r^{GPU}$ and pick the value that leads to maximum energy saving.

$\Delta E = (P^{GPU}_{dyn\_base} + P^{GPU}_{static} + P_{static})^{CPU} * r^{GPU} * slack_k + max(P^{CPU}_{dyn\_base} * T^{CPU}_k (1 - (\frac{f^{CPU}_{ADJUST}}{f^{CPU}_{BASE}})^{1.4}), P^{CPU}_{dyn\_base} * r^{CPU} * slack_k)$

The theoretical performance (execution time) improvement for $k^{th}$ iteration can be calculated as:

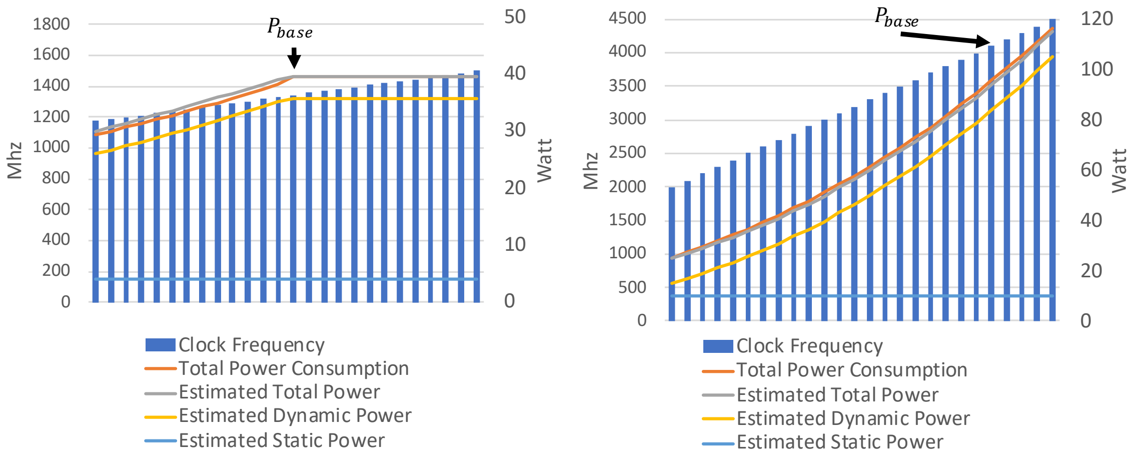
$$\Delta T = min(r_{max}^{GPU} * slack_k, (1 - r_{min}^{CPU}) * slack_k$$

In order to tune our bi-directional slack reclamation for energy saving, we need to know the dynamic power and static power consumed by the processors. However, we cannot directly measure those power values except the total power consumption of the processors. So, we choose to estimate the dynamic power and static power by solving the unknown variables in $P_{dyn} = C \cdot V^2 \cdot f + P_{static}$ given different pairs of real total power consumption and clock frequencies measurements. The estimation results are also shown in **Fig. 5.6**, in which we achieve 98.2% prediction accuracy on the GPU and 97.1% prediction accuracy on the CPU.

## 5.5 Integrating Overclocking and Fault Tolerance

### 5.5.1 Overclocking on CPUs and GPUs

Common slack reclamation based energy saving approaches are based on reducing the speed (i.e., clock frequency) of processors. This is based on the approximate relation between power and clock frequency of processors: $P_d = \propto f^{2.4}$ [73]. For example, assuming execution time is inversely linearly proportional to frequency, if we reduce the clock frequency by half, the execution time is doubled. However, the power consumption is only about 1/5 of the original, which compensates the extra energy consumption cost by extra execution time. The total dynamic energy is reduced by about 60%. On the other hand, it is easy to see that if we increase the clock frequency of a processor, more energy will be consumed.



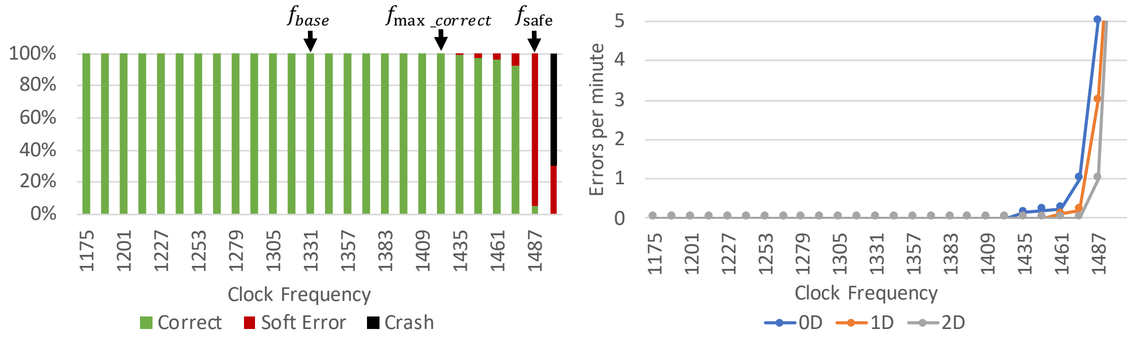
(a) GPU

(b) CPU

Figure 5.6: Processor power consumption under different clock frequencies

However, if we further push the clock frequency beyond a certain level, the power consumption will no longer increase due to the power limit set by the hardware or user settings. **Fig. 5.6(a)** shows the power change pattern as we increase the clock frequency on our testbed GPU. It is easy to see that energy consumption will decrease given that power is fixed and execution time reduces with the increase of clock frequency. To maintain the higher clock frequencies with fixed power consumption, the voltage supply will decrease according to the relation: $P_{dyn} = C \cdot V^2 \cdot f$, where P is power, C is dynamic capacitance (usually near constant), V is supplied voltage, and f is clock frequency. However, limited by the hardware we cannot apply the power limit for our testbed CPU as shown in **Fig. 5.6(b)**.

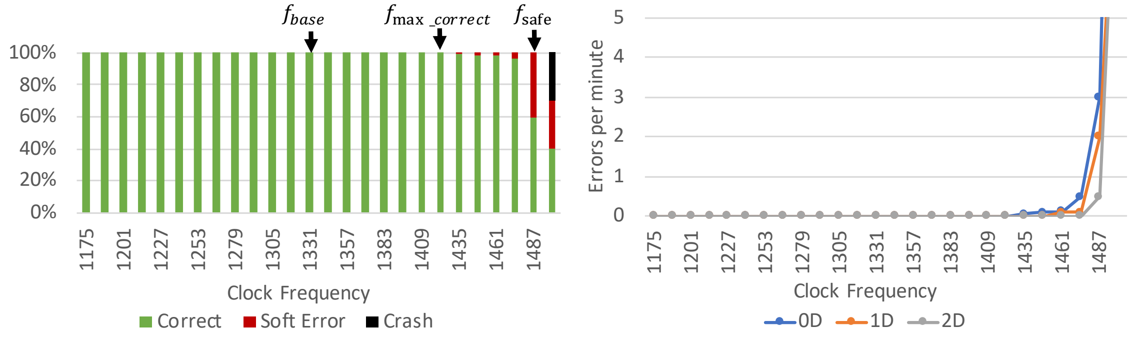
A proper voltage supply is key to maintain system stability at a certain clock frequency. Usually, there is a threshold voltage – $V_{min}$. Once the voltage drops below that



(a) Error Type Distribution

(b) Error Rate

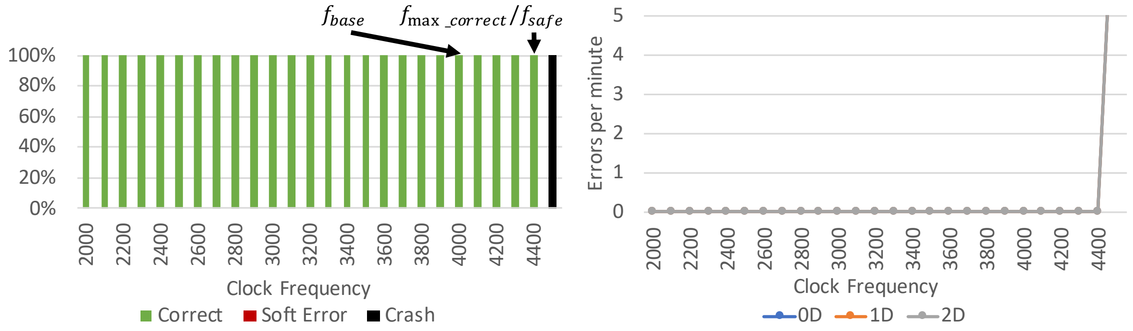
Figure 5.7: Error during TMU under different clock frequencies on GPU.



(a) Error Type Distribution

(b) Error Rate

Figure 5.8: Error during PU under different clock frequencies on GPU.



(a) Error Type Distribution

(b) Error Rate

Figure 5.9: Error during PD under different clock frequencies on CPU.

the system could not perform correctly. According to previous studies, [113, 114], $V_{min}$ is determined by both the characteristics of hardware and the program running on it.

An unreliability processor can have multiple kinds of hardware fault. One of the most common kinds of hardware fault is the transient fault. It can randomly occur to any arbitrary component of a processor in a certain frequency depending on the degrees of overclocking. Transient fault rate can increase as we push the processor to higher clock frequencies or decrease if we lower the clock frequency since transient faults are usually not caused by permanent hardware damage.

If hardware transit faults occur and cause errors in the software level, they are usually classified into two categories: hard error and soft error. If the error causes processes or system crash, it is classified as a hard error. If the error causes incorrect calculation results, it is classified as a soft error. **Fig. 5.7 (a) - 5.9 (a)** show the CPU/GPU error type distribution under different clock frequencies for TMU, PU, and PD. The results are obtained from testing each operation with a small-sized input 10000 times given different clock frequencies. The reason we choose a small input size is that it is easier for us to distinguish different error type since shorter execution time can reduce the possibility that multiple errors occur during a single run. The distribution indicates how many runs obtain correction results, contain soft errors, or crashed. We adjust the clock frequency from the lowest possible frequency at $f_{min}$. $f_{base}$ is the baseline clock frequency used throughout this work and it is also the highest clock frequency that the processor will be automatically adjusted to. Beyond that, the clock frequencies are considered at overclocking frequencies. As long as the clock frequency is lower than $f_{max\_correct}$, the processor can still operate normally without have hardware faults. Beyond $f_{max\_correct}$ and below $f_{safe}$ the processor begins to have soft errors but does not leads to process or system crash. However, beyond $f_{safe}$ we would experience process or system crash. Our GPU shows these clock frequencies as different value, however, our CPU does not exhibit any soft error before it begins to have crashes.

In this work, we only aim to handle soft errors caused by overclocking. For doing that, we restrict the processor clock frequency to be under a certain frequency threshold, so that possibility of hard error (e.g., crash) is insignificant or negligible.

Depending on where the transient fault occurs, it may manifest itself as different kinds of soft error. For example, the calculation error is usually caused by faults in the logic part of ALU or FPU. Memory storage error is usually caused by faults (e.g., bit flips) in the storage cells of DRAM, cache, or registers. For matrix operations, matrix elements sometimes can be repeatedly accessed in order to obtain final results. If an element whose value is corrupted by software gets repeatedly referenced, it may cause error propagation i.e., accumulated error in the result matrix. Depending on the cause of the error and the computation pattern (i.e., how data is used/reused) of a matrix operation, the error pattern can be different. Here we define three degrees of error pattern: 0D, 1D, and 2D.

- **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. So, here we further study the error rate of those three type of error propagation in CPU and GPU still using our previous testing method. The results are shown in **Fig. 5.7 (b) - 5.9 (b)**.

### 5.5.2 Choosing Suitable Clock Frequencies

To handle soft errors, we choose to use ABFT since it brings much lower fault tolerance overhead compared with more general application-level duplication based approaches such as DMR (100% overhead) or TMR (200% overhead). ABFT is an application-specific approach and it is usually designed for matrix operations including matrix decompositions focused in this work. It is based on the idea that if we encode a certain amount of matrix information in checksums before a matrix operation and apply the same matrix operation also to checksums, the checksum relation would still hold for the result matrix. By verifying the checksum relations, we are able to detect or even correct errors in the result matrix.

Depending on how much information is encoded in checksums, the error tolerance strength is different. Currently, there are two common ways to do checksum encoding (i.e., checksum scheme). (1) Single side checksum scheme encodes matrices along either rows or columns (i.e., calculate the sum of elements of every row or column). Since it only encodes matrix in one dimension, it brings relative lower overhead. However, it can only efficiently tolerate 0D error pattern. (2) Full checksum scheme encodes matrices along both rows and column at the same time. Since it encodes matrices in both dimensions, it brings stronger protection i.e., both 0D and 1D error pattern. However, it also brings higher fault tolerance overhead.

Since there is a limitation on how many errors it can correct for a certain checksum design and the number of errors can increase with the raising of the processor clock frequency, it is important that we determine clock frequencies and checksum designs that can ensure all errors can be detected and corrected with high confidence. Otherwise, an undetected or uncorrected error would cause serious error propagation later, which requires recovery with high overhead. In this work, we find that it is useful to estimate the max possible processor clock frequencies given two checksum designs. In order to do that, we first define an error rate function $R$ given clock frequency that can be derived from **Fig. 5.7 (b) - 5.9 (b)**:

$$\lambda_{f,error\_type} = R(f, error\_type)$$

where $\lambda$ is the error rate of a certain *error_type*. The *error_type* can be 0D, 1D, or 2D. $f$ is the processor clock frequency. Assuming the rate is constant for a given clock frequency, we can treat the distribution of probability errors occur during a period of time as Poisson distribution. So, we can calculate the probability of have error less or equal than $k$ during a period of time $T$ using the CDF of Poisson distribution function:

$$P(\#\ of\ errors\ in\ error\_typek) = \sum_{i=0}^{k} \frac{e^{-\lambda_{f,error\_type}T}(\lambda_{f,error\_type}T)^i}{i!} \tag{5.8}$$

Based on equation 5.8, we can define a function $F$ that output the max possible clock frequency that leads to less to equal to $k$ errors *error_type* during time period of $T$ with at least probability of $p$:

$$f_{max} = F(k, error\_type, T, p)$$

For single side checksum, it can only tolerate up to one 0D error propagation. Since we are encoding checksum for each $nb \times nb$ matrix block individually, it can tolerate up to $\frac{m}{nb} \times \frac{n}{nb}$ 0D error propagations for matrix size of $m \times n$. However, it cannot tolerate any 1D or 2D error propagations.

So, the max possible clock frequency is estimated as:

$$f_{max\_single} = min(F(\frac{m}{nb} \times \frac{n}{nb}, 0D, T_{OP}, p_0), \tag{5.9}$$

$$F(0, 1D, T_{OP}, p_1), F(0, 2D, T_{OP}, p_2)) \tag{5.10}$$

Similarly, we can also estimate the max possible clock frequency for full checksum as:

$$f_{max\_full} = min(F(\frac{m}{nb} \times \frac{n}{nb}, 0D, T_{OP}, p_0), \tag{5.11}$$

$$F(\frac{m}{nb} \times \frac{n}{nb}, 1D, T_{OP}, p_1), \tag{5.12}$$

$$F(0, 2D, T_{OP}, p_2)) \tag{5.13}$$

where $T_{OP}$ is the execution time of this matrix operation, which can be obtained from our algorithmic prediction model. $p_0, p_1, p_2$ can be set as desired. Higher probabilities such as 99% indicate that from 99% of the case ABFT can handle all errors. However, higher probabilities also impose a stricter rule on clock frequency adjustment, which brings a trade-off here.

## 5.6 Experiments

In this section we detail the evaluation of PowerLA on a GPU-accelerated heterogeneous system: a linear algebra library of dense matrix decomposition (Cholesky, LU, and QR).

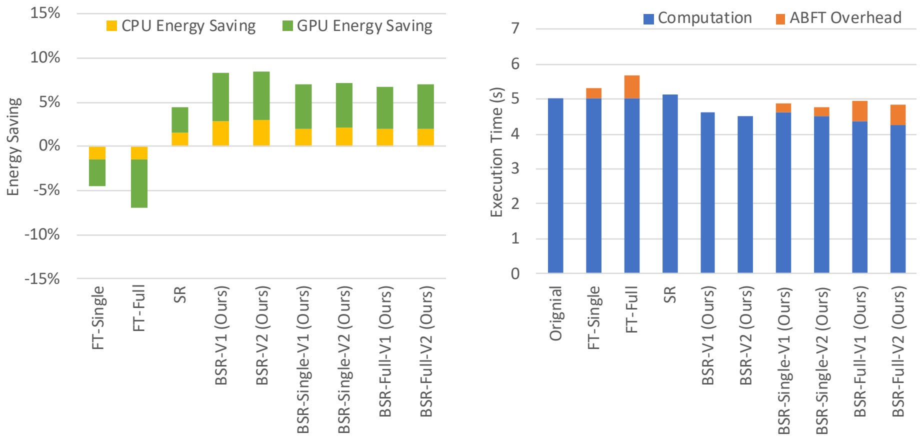
Table 5.3: Hardware Configuration for Experiments.

| Component | CPU | GPU |
|---|---|---|
| Processor | Intel Core i7-6700K | NVIDIA GeForce GTX 750 Ti |
| Peak Perf. | 92 GFLOPS (single) | 1389 GFLOPS (single) |
| Original Clock | 0.8-4.2(↑by0.1)GHz | 980-1331(↑by 13)MHz |
| Overclocking | 4.2-4.4(↑by0.1)GHz | 1331-1500(↑by 13)MHz |
| Memory | 8 GB RAM | 2 GB RAM |

### 5.6.1 Evaluation Methodology

Since PowerLA brings performance, fault tolerance, and energy efficiency optimization, we compare our works with the state-of-the-art works among three aspects.

- Original MAGMA: The matrix decompositions with the state-of-the-art performance on heterogeneous systems with CPUs and GPUs without fault tolerance and energy saving optimizations.
- FT-Single: Matrix decompositions with single side checksum ABFT.
- FT-Full: Matrix decompositions with full checksum ABFT.
- SR: The matrix decompositions with algorithmic slack reclamation.
- BSR-V1(ours): Matrix decompositions with our strict bi-directional algorithmic slack reclamation without applying fault tolerance.



(a) Energy consumption comparison

(b) Performance comparison

Figure 5.10: Energy consumption and performance comparison for Cholesky decomposition.

- `BSR-V2(ours)`: Matrix decompositions with our relaxed bi-directional algorithmic slack reclamation without applying fault tolerance.

- `BSR-Single-V1(ours)`: Matrix decompositions using our strict bi-direction algorithmic slack reclamation with single side checksum ABFT.

- `BSR-Single-V2 (ours)`: Matrix decompositions using our relaxed bi-direction algorithmic slack reclamation with single side checksum ABFT.

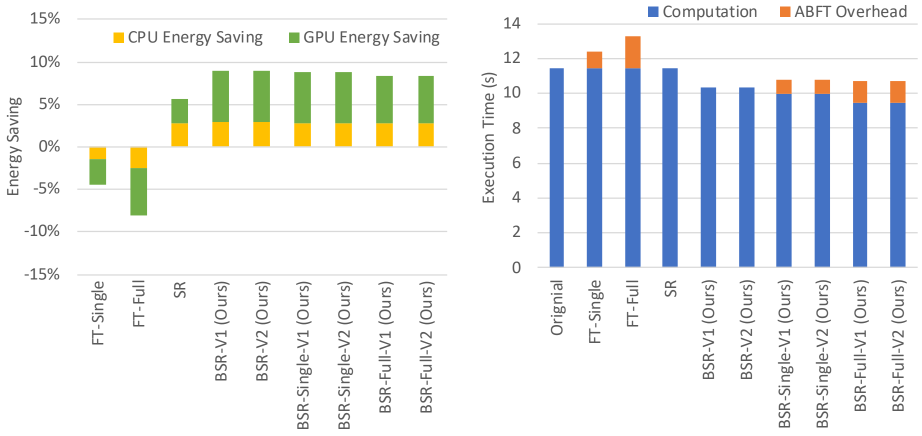
- `BSR-Full-V1 (ours)`: Matrix decompositions using our strict bi-direction algorithmic slack reclamation with full checksum ABFT.

- `BSR-Full-V2 (ours)`: Matrix decompositions using our relaxed bi-direction algorithmic slack reclamation with full checksum ABFT.

### 5.6.2 Experimental Setup

All experiments were performed on a power-aware CPU-GPU server with Ubuntu 16.04 running on it. Table 5.3 lists hardware configuration of the experimental platform. The CPU overclocking is achieved by: 1) Setting the maximum overclocking frequency in BIOS; 2) Setting the CPU power state governor to keep it always at the highest clock frequency (`sudo cpupower frequency-set --governor performance`); 3) Adjusting the max CPU clock frequency (`CPU set max clock - sudo cpupower frequency-set -u <clock>`, where `<clock>` is in Hz). The CPU and DRAM energy and power consumption are measured by reading the associated register counters in Intel CPUs via MSR (Linux `perf` API). GPU overclocking is achieved by 1) Setting a max graphics clock frequency offset in `NVIDIA X Server Settings GUI`; 2) Setting the power management to keep GPU always running at the highest clock frequency (also done in `NVIDIA X Server Settings GUI`); 3) Querying the available GPU clock frequencies (`nvidia-smi -q -d SUPPORTED_CLOCKS`); 4) Setting the graphics clock frequency (`nvidia-smi -ac <graphics_clock>, <memory_clock>`, where the clock unit is in Mhz). The GPU and DRAM energy and power consumption are measured using NVIDIA Management Library (NVML) API. We keep memory clock frequencies of CPU and GPU at their default value in our experiments and only modify the core clock frequencies.

We applied all above test scenarios to Cholesky, LU, and QR decomposition on the MAGMA library with multiple global matrix sizes each (ranging from 5120 to 10240). However, due to limit space, we only show the result for the input size of 10240 × 10240. For other input matrix sizes, the results are similar.



(a) Energy consumption comparison

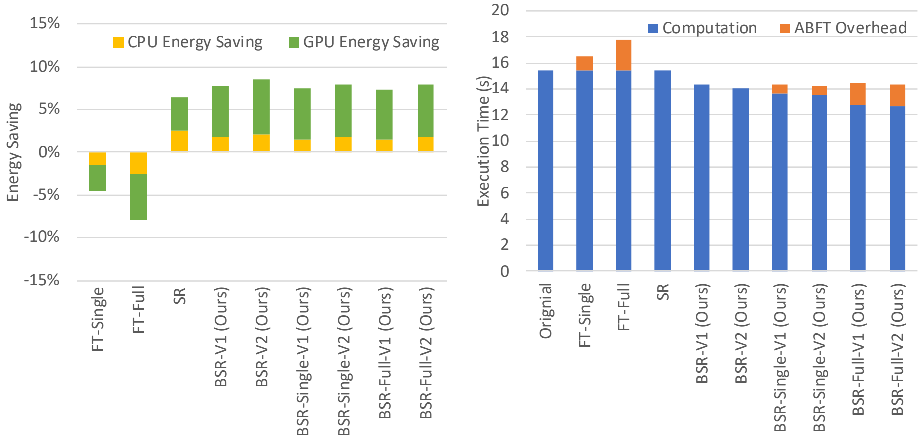
(b) Performance comparison

Figure 5.11: Energy consumption and performance comparison for LU decomposition.

### 5.6.3 Results

#### Energy Consumption Comparison

We first compare the energy consumption of each design. **Fig. 5.10(a) - 5.12(a)** show the energy impact bring by different designs on Cholesky, LU, and QR decomposition. In these figures, positive values represent the percentage of energy saving and negative values represent the percentage energy of more energy cost compared with the baseline. The baseline is chosen as the original MAGMA version running on a clock frequency of $f_{base}$ for CPU and GPU during the entire execution. Comparing with fault tolerant only designs (i.e., `FT-single` and `FT-full`), our new designs can save 11.0% - 15.9% total energy. In addition, comparing with energy saving optimization only design (i.e., `SR`), our new designs can save 2.8% - 3.6% extra energy, which is equal to 51.1% - 84.4% more energy saving.



(a) Energy consumption comparison

(b) Performance comparison

Figure 5.12: Energy consumption and performance comparison for QR decomposition.

### Performance Comparison

**Fig. 5.10(b) - 5.12(b)** show the performance impact bring by different designs on Cholesky, LU, and QR decomposition. Comparing with fault tolerant only designs (i.e., `FT-single` and `FT-full`), our new designs can achieve 8.2% - 15.0% performance improvement. In addition, comparing with energy saving optimization only design (i.e., `SR`), our new designs can achieve 2.2% - 9.8% performance improvement.

## 5.7 Summary

Energy efficiency and fault tolerance are becoming critical factors of concern when achieving parallelism in high performance scientific computing in this era. Matrix decompositions on heterogeneous systems with GPU have become key components in many scientific applications. Many works have been proposed for improving their energy efficiency and reliability. However, not much work has been focused on optimizing both energy efficiency and reliability at the same time since they do not naturally compatible with each other.

In this work, we proposed a novel bi-directional slack reclamation, which allows both fault tolerance and energy saving to be applied to matrix decompositions at the same time. Experiment results show that compared with original slack reclamation, our work not only enables fault tolerance but also bring higher energy saving.

# Chapter 6

# Related Work

## 6.1 Algorithm Based Fault Tolerance for Soft Errors

Today's computing systems have been used widely in academic and industrial fields to solve various challenging problems. However, many kinds of faults can occur in computing systems, which may negatively affect the process of computation or the correctness of calculation results. One of the most challenging error to tolerate is soft error. Soft error leads to incorrect results yet without aborting the computation process, which makes it hard to be detect and correct.

[83, 45] have shown future supercomputers will be highly susceptible to soft errors, especially the ones that lead to silent data corruption (SDC). In [94], a large-scale study of GPU error rate shows that two-thirds of the tested GPU hardware exhibit pattern-sensitive soft errors in GPU memory or logic parts. In [155], it is shown that GPUs exhibit high soft error susceptibility and soft errors arise more frequently as the workload increases. In addition, many GPU undervolting and overclocking based energy saving approaches [113] greatly impacts the system reliability and thus cause the frequent occurrences of soft errors.

One common general appraoch to tolerate soft error is to use Triple Modular Redundancy (TMR) [133]. 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%). In order to address this issue, fault tolerance built at application level is desirable, since it can leverage the semantics and structure of a specific application with low fault tolerance overhead.

Algorithm-based fault tolerance (ABFT) technique represents a middle ground between application-level fault tolerance and system-level fault tolerance. It was first proposed by Huang and Abraham [102], which tolerate 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.

In recent years, ABFT has been extended by many researchers in recent years. For example, in [39], Banerjee et al. proposed an ABFT scheme that works on hypercube multiprocessor. In [146], Sao and Vuduc explored a self-stabilizing fault tolerance approach for iterative methods. Among them the most relevant related works to our work are the ABFT protected one-sided matrix decomposition [184, 180, 50, 53, 65, 185].

## 6.2 Energy Saving for Matrix Operations

The growing prevalence of heterogeneous architectures has motivated a large body of energy efficient approaches[136], but few of them were designed specifically for numerical linear algebra operations, such as dense matrix factorizations extensively used in HPC. Some efforts presented next can also be applied to heterogeneous systems with similar techniques.

**OS-level Scheduling**. Liu *et al.* [129] proposed several power-aware techniques for a CPU-GPU heterogeneous system including two static/dynamic mapping algorithms and one aggressive voltage reduction scheme. Decent power and energy savings were achieved (more than 20%) towards several matrix workloads, but our work targets different matrix algorithms with less slack and thus less energy saving opportunities. Their work focused on time-sensitive applications that require significant computational capacity, such as real-time scoring of bank transactions, live video processing, etc. Our work can also meet fine-grained timing requirements of applications, which is guaranteed by respecting tasks on the critical path. Hong *et al.* [97] proposed an integrated power and performance model to statically determine the optimal number of processors for a given application running on GPU, based on the intuition that using more cores is not necessary for applications reaching the peak memory bandwidth. By using fewer GPU cores, average 11% energy savings can be achieved for memory bandwidth limited applications. The proposed system can be used by a thread scheduler for online energy saving decision-making. This approach was also evaluated on different applications than us – the more energy savings do not demonstrate their strength.

**Library-level Scheduling**. Alonso *et al.* [34] incorporated two energy saving techniques at library level to schedule the computation of dense linear algebra operations on a hybrid platform of a multicore CPU and multiple GPU. Specifically, idle threads were blocked when no tasks to process, and busy-waiting threads were also blocked by synchronization primitives when waiting for a device to finish its work. Due to lack of consideration of algorithmic characteristics of dense linear algebra operations, the reported average energy cost reduction was around 4% for Cholesky and 7% for LU. Anzt *et al.* [38] applied energy efficient techniques on GPU-accelerated iterative linear solvers for memory-intensive sparse linear systems, and demonstrated that considerable energy savings (17.8% on average) can be fulfilled without harming performance noticeably, by setting CPU to a low power state during the time when GPU is running while CPU is busy-waiting. However, the proposed solution cannot work for our scenario where CPU and GPU frequently interact with data movement. Note that there exists more slack for sparse linear algebra operations and more energy savings are expected compared to dense ones.

**Online and Offline Workload Prediction**. There exist numerous solutions that predict workload and slack, facilitating energy saving decision-making, spanning from online to offline. Zhu *et al.* [186] proposed a power-aware consolidation scheme of scientific workflow tasks for energy and resource cost optimization. The pSciMapper framework consists of online consolidation and offline analysis for resource usage prediction (e.g., CPU utilization) using Hidden Markov Model (HMM), with reported average prediction error of moderate 3.3%. However, the drawback of this approach is considerable slowdown around 15%, which is unacceptable in HPC nowadays. For comparison purposes, we also adopt

HMM-based slack prediction in an online fashion instead, and experimental results indicate a higher prediction error (up to 11.29%) can be incurred. Li *et al.* [125] applied a Predictive Bayesian Network to identify daily workload patterns and adjust resource provisioning accordingly for cloud datacenters. The prediction algorithm was evaluated to be considerably effective (only 0.43% average prediction error was observed). Our work differs from this offline workload prediction – `GreenLA` is able to achieve energy savings online for HPC runs using negligible amount of training dataset from the earlier stage of the runs. Tse *et al.* [174] proposed a novel Monte Carlo simulation framework that supports multiple types of hardware accelerators (FPGA and GPU) and provided scheduling interfaces to adaptively perform load balancing at runtime for performance and energy efficiency. The energy savings achieved is however from performance gain obtained from the collaborative simulation framework, not from an energy efficient strategy.

# Chapter 7

# Conclusions

To enabled fault tolerance for one-sides matrix decomposition on heterogeneous systems with GPUs, we design the first ABFT for one-sided matrix decompositions on heterogeneous systems with GPUs – Enhanced On-line ABFT. By designing a novel ABFT checksum maintaining algorithm for GPUs, our Enhanced On-line ABFT effectively tolerates errors during matrix decompositions. It is also the first Online-ABFT scheme that can correct both computing and storage errors. To further enable stronger error propagation protection, we propose Full checksum ABFT. Full checksum ABFT enables also much wider and stronger error protection for matrix decompositions benefited from our novel full checksum scheme specially designed for one-side matrix decompositions. This stronger error protection can efficiently reduce and tolerate error propagations during matrix decompositions. To reduce fault tolerance overhead, we give the first systematic study of error propagation pattern caused by computation, memory system, and communication error that occurs in all major operations of matrix decompositions. Based on the study results, we provide an efficient ABFT checking scheme by prioritizing the checksum verification according to the sensitivity of matrix operations, which leads to strong error protection with low fault tolerance overhead. To accurately predict the slacks in one-sided matrix decompositions and maximize energy saving, we propose a novel algorithmic slack reclamation energy saving approach for one-sided matrix decompositions on GPUs. Our work exploit algorithmic knowledge of linear algebra operations to predict slack on CPU and GPU, and use application-level DVFS strategies to reclaim the slack for energy savings. Finally, we propose `PowerLA`, enhanced one-sided matrix decompositions that are both energy efficient and fault tolerance.

# Bibliography

[1] ACML:developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/.

[2] *ASC Sequoia Benchmark Codes*. https://asc.llnl.gov/sequoia/benchmarks/.

[3] *Automatically Tuned Linear Algebra Software (ATLAS)*. http://math-atlas.sourceforge.net/.

[4] *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/.

[5] *CPUFreq - CPU Frequency Scaling*. https://wiki.archlinux.org/index.php/CPU_Frequency_Scaling.

[6] *CPUSpeed*. http://carlthompson.net/Software/CPUSpeed/.

[7] *CUBLAS*. developer.nvidia.com/cuBLAS.

[8] *CUDA Best Practices Guide:*. docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[9] *CULA*. www.culatools.com.

[10] *DPLASMA: Distributed Parallel Linear Algebra Software for Multicore Architectures*. http://icl.cs.utk.edu/dplasma/.

[11] *Green500 Supercomputer Lists*. http://www.green500.org/.

[12] *HPL - High-Performance Linpack Benchmark*. http://www.netlib.org/benchmark/hpl/.

[13] *Intel Hyper-Threading Technology*. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[14] *Intel Xeon Phi Coprocessors*. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner/.

[15] *LAPACK - Linear Algebra PACKage*. http://www.netlib.org/lapack/.

[16] *Linux File Copy Command*. http://www.linux.org/.

[17] *MAGMA*. icl.cs.utk.edu/magma.

[18] *MKL*. software.intel.com/en-us/mkl.

[19] *NAS Parallel Benchmarks.* http://www.nas.nasa.gov/publications/npb.html.

[20] *NVIDIA Fermi Compute Architecture Whitepaper.* https://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[21] *NVIDIA Kepler Compute Architecture Whitepaper.* https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[22] *NVIDIA Maxwell Compute Architecture Whitepaper.* https://international.download.nvidia.com/geforce-com/international/pdfs/ GeForce_GTX_980_Whitepaper_FINAL.PDF.

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

[24] *NVIDIA System Management Interface (nvidia-smi).* https://developer.nvidia.com/nvidia-system-management-interface/.

[25] *NVIDIA Tesla GPU Accelerators for Servers.* http://www.nvidia.com/object/tesla-servers.html.

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

[27] *Parallel MPI BZIP2 (MPIBZIP2).* http://compression.ca/mpibzip2/.

[28] *ScaLAPACK - Scalable Linear Algebra PACKage.* http://www.netlib.org/scalapack/.

[29] *Streams And Concurrency Webinar.* on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf.

[30] *TOP500 Supercomputer Lists.* http://www.top500.org/.

[31] *Watts up? Meters.* https://www.wattsupmeters.com/.

[32] *Performance of the high performance LINPACK benchmark for distributed memory computers.* http://icl.cs.utk.edu/graphics/posters/files/SC13-HPL.pdf, 2013.

[33] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. DVFS-control techniques for dense linear algebra operations on multi-core processors. *CSRD*, 27(4):289–298, November 2012.

[34] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Reducing energy consumption of dense linear algebra operations on hybrid CPU-GPU platforms. In *Proc. ISPA*, pages 56–62, 2012.

[35] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Saving energy in the LU factorization with partial pivoting on multi-core processors. In *Proc. PDP*, pages 353–358, 2012.

[36] P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control. In *Proc. HPCS*, pages 463–470, 2011.

[37] P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *CSRD, Special Issue*, August 2012.

[38] H. Anzt, V. Heuveline, J. Aliaga, M. Castillo, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Proc. IGCC*, pages 1–6, 2011.

[39] Prithviraj Banerjee, Joe T Rahmeh, Craig Stunkel, VS Nair, Kaushik Roy, Vijay Balasubramanian, and Jacob A Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *Computers, IEEE Transactions on*, 1990.

[40] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[41] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[42] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA. Technical Report UT-CS-10-660, September 2010.

[43] Aurelien Bouteiller, Thomas Herault, George Bosilca, Peng Du, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Transactions on Parallel Computing*, 2015.

[44] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 155–164. ACM, 2008.

[45] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[46] M. Castillo, J. C. Fernández, R. Mayo, E. S. Quintana-Ortí, and V. Roca. Analysis of strategies to save energy for message-passing dense linear algebra kernels. In *Proc. PDP*, pages 346–352, 2012.

[47] Sandra Catalán Pallarés, José Ramón Herrero Zaragoza, Enrique S Quintana Ortí, and Rafael Rodríguez Sánchez. Energy balance between voltage-frequency scaling and resilience for linear algebra routines on low-power multicore architectures. 2017.

[48] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proc. PPoPP*, pages 2–11, 2006.

[49] G. Chen, K. Malkowski, M. Kandemir, and P. Raghavan. Reducing power with performance constraints for parallel sparse applications. In *Proc. IPDPS*, pages 1–8, 2005.

[50] 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.

[51] Jieyang Chen and Zizhong Chen. Cholesky Factorization on Heterogeneous CPU and GPU Systems. In *Frontier of Computer Science and Technology (FCST), 2015 Ninth International Conference on*, pages 19–26. IEEE, 2015.

[52] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, et al. Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 68. IEEE Press, 2018.

[53] Jieyang Chen, Sihuan Li, and Zizhong Chen. GPU-ABFT: Optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs. In *Networking, Architecture and Storage (NAS), 2016 International Conference on*, 2016.

[54] Jieyang Chen, Li Tan, Panruo Wu, Dingwen Tao, Hongbo Li, Xin Liang, Sihuan Li, Rong Ge, Laxmi Bhuyan, and Zizhong Chen. Greenla: green linear algebra software for gpu-accelerated heterogeneous computing. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 667–677. IEEE, 2016.

[55] L. Chen, Z. Chen, P. Wu, R. Ge, and Z. Zong. Energy efficient parallel matrix-matrix multiplication for DVFS-enabled clusters. In *Proc. PASA, with ICPP*, pages 239–245, 2012.

[56] Longxiang Chen, Dingwen Tao, Panruo Wu, and Zizhong Chen. Extending checksum-based abft to tolerate soft errors online in iterative methods.

[57] Zizhong Chen. Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. 2013.

[58] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *Parallel and Distributed Systems, IEEE Transactions on*, 2008.

[59] Hyungmin Cho, Chen-Yong Cher, Thomas Shepherd, and Subhasish Mitra. Understanding soft errors in uncore components. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[60] J. Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *Proc. HPC-Asia*, pages 224–229, 1997.

[61] JeeWhan Choi, Daniel Bedard, Robert J. Fowler, and Richard W. Vuduc. A roofline model of energy. In *IPDPS*, pages 661–672. IEEE Computer Society, 2013.

[62] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proc. DATE*, pages 18–28, 2004.

[63] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power mamagement via dynamic voltage/frequency scaling. In *Proc. ICAC*, pages 31–40, 2011.

[64] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proc. ICS*, pages 162–171, 2011.

[65] Teresa Davies and Zizhong Chen. Correcting soft errors online in lu factorization. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 2013.

[66] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: Active low-power modes for main memory. In *Proc. ASPLOS*, pages 225–238, 2011.

[67] Chong Ding, Christer Karlsson, Hui Liu, Teresa Davies, and Zizhong Chen. Matrix multiplication on gpus with on-line fault tolerance. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, 2011.

[68] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, February 2013.

[69] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with GPUs. *Numerical Computations with GPUs*, pages 1–26, 2014.

[70] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 2012.

[71] Peng Du, Piotr Luszczek, and Jack Dongarra. High performance dense linear system solver with soft error resilience. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 272–280. IEEE, 2011.

[72] Peng Du, Piotr Luszczek, and Jack Dongarra. High performance dense linear system solver with resilience to multiple soft errors. volume 9, pages 216–225. Elsevier, 2012.

[73] R. Efraim, R. Ginosar, C. Weiser, and A. Mendelson. Energy aware race to halt: A down to EARtH approach for platform energy management. *IEEE Computer Architecture Letters*, 13(1):25–28, January 2014.

[74] M. Elgebaly and M. Sachdev. Efficient adaptive voltage scaling system through on-chip critical path emulation. In *Proc. ISLPED*, pages 375–380, 2004.

[75] James Elliott, Mark Hoemmen, and Frank Mueller. Evaluating the impact of sdc on the gmres iterative solver. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014.

[76] Bo Fang, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman, and Matei 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*, pages 117–130. ACM, 2017.

[77] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva 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*, pages 221–230. IEEE, 2014.

[78] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. A systematic methodology for evaluating the error resilience of GPGPU applications. *IEEE Trans. Parallel Distrib. Syst.*, 27(12):3397–3411, 2016.

[79] Bo Fang, Jiesheng Wei, Karthik Pattabiraman, and Matei Ripeanu. Evaluating error resiliency of GPGPU applications. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1502–1503. IEEE, 2012.

[80] Bo Fang, Jiesheng Wei, Karthik Pattabiraman, and Matei Ripeanu. Towards building error resilient GPGPU applications. *SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.

[81] A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proc. ICS*, pages 393–402, 2005.

[82] E. Feller, C. Rohr, D. Margery, and C. Morin. Energy management in IaaS clouds: A holistic approach. In *Proc. CLOUD*, pages 204–212, 2012.

[83] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High*

*Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.

[84] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proc. PPoPP*, pages 164–173, 2005.

[85] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proc. SC*, page 34, 2005.

[86] R. Ge, X. Feng, W.-C. Feng, and K. W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *Proc. ICPP*, page 18, 2007.

[87] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron. PowerPack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, May 2010.

[88] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong. Effects of dynamic voltage and frequency scaling on a K20 GPU. In *Proc. PASA*, pages 826–833, 2013.

[89] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[90] L Bautista Gomez, Franck Cappello, Luigi Carro, Nathan DeBardeleben, Bo Fang, Sudhanva Gurumurthi, Karthik Pattabiraman, Paolo Rech, and M Sonza Reorda. GPGPUs: how to combine high computational power with high reliability. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 341. European Design and Automation Association, 2014.

[91] John Gunnels, Daniel S Katz, Enrique S Quintana-Orti, RA Van de Gejin, et al. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Dependable Systems and Networks*, 2001.

[92] Doug Hakkarinen and Zizhong Chen. Algorithmic cholesky factorization fault recovery. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.

[93] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *Parallel and Distributed Systems, IEEE Transactions on*, 2015.

[94] Imran S Haque and Vijay 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.

[95] T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time MPI broadcast algorithm for large-scale InfiniBand clusters with multicast. In *Proc. CAC, with IPDPS*, pages 232–239, 2007.

[96] J. D. Hogg. A dag-based parallel cholesky factorization for multicore systems. Technical Report RAL-TR-2008-029, October 2008.

[97] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proc. ISCA*, pages 280–289, 2010.

[98] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *Proc. IPDPS*, 2006.

[99] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Proc. SC*, page 1, 2005.

[100] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proc. PLDI*, pages 38–48, 2003.

[101] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proc. ISLPED*, pages 275–278, 2001.

[102] Kuang-Hua Huang, Jacob Abraham, et al. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 1984.

[103] S. Hunold and T. Rauber. Automatic tuning of PDGEMM towards optimal performance. In *Proc. Euro-Par*, pages 837–846, 2005.

[104] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. ISLPED*, pages 197–202, 1998.

[105] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202. ACM, 1998.

[106] Yulu Jia, Piotr Luszczek, George Bosilca, and Jack 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*, 2013.

[107] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Proc. SC*, page 33, 2005.

[108] C. Karlsson, T. Davies, C. Ding, H. Liu, and Z. Chen. Optimizing process-to-core mappings for two dimensional broadcast/reduce on multicore architectures. In *Proc. ICPP*, pages 404–413, 2011.

[109] A. Karwande, X. Yuan, and D. K. Lowenthal. CC–MPI: a compiled communication capable MPI prototype for ethernet switched clusters. In *Proc. PPoPP*, pages 95–106, 2003.

[110] K. H. Kim, R. Buyya, and J. Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters. In *Proc. CCGrid*, pages 541–548, 2007.

[111] H. Kimura, M. Sato, Y. Hotta, T. Boku, and D. Takahashi. Empirical study on reducing energy of parallel programs using slack reclamation by DVFS in a power-scalable high performance cluster. In *Proc. CLUSTER*, pages 1–10, 2006.

[112] 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 *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 294–307, Dec 2015.

[113] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, and Vijay Janapa Reddi. Safe limits on voltage reduction efficiency in gpus: a direct measurement approach. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pages 294–307. IEEE, 2015.

[114] Jingwen Leng et al. *Guardband management in heterogeneous architectures.* PhD thesis, 2016.

[115] Ang Li, Shuaiwen Leon Song, Eric Brugel, Akash Kumar, Daniel Chavarria-Miranda, and Henk Corporaal. X: A comprehensive analytic model for parallel machines. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 242–252. IEEE, 2016.

[116] Ang Li, Shuaiwen Leon Song, Akash Kumar, Eddy Z Zhang, Daniel Chavarría-Miranda, and Henk Corporaal. Critical points based register-concurrency autotuning for gpus. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 1273–1278. EDA Consortium, 2016.

[117] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk 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*, pages 297–311. ACM.

[118] Ang Li, YC Tay, Akash Kumar, and Henk Corporaal. Transit: A visual analytical model for multithreaded machines. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 101–106. ACM, 2015.

[119] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-grained synchronizations and dataflow programming on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 109–118. ACM, 2015.

[120] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17. ACM, 2015.

[121] D. Li, B. R. de Supinski, M. Dolz, D. S. Nikolopoulos, and K. W. Cameron. Strategies for energy efficient resource management of hybrid programming models. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):144–157, January 2013.

[122] D. Li, B. R. de Supinski, M. Schulz, K. W. Cameron, and D. S. Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *Proc. IPDPS*, pages 1–12, 2010.

[123] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S Vetter. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[124] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proc. HPCA*, page 14, 2004.

[125] J. Li, K. Shuang, S. Su, Q. Huang, P. Xu, X. Cheng, and J. Wang. Reducing operational costs through consolidation with resource prediction in the cloud. In *Proc. CCGrid*, pages 793–798, 2012.

[126] Xin Li, Kai Shen, Michael C Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *USENIX Annual Technical Conference*, pages 275–280, 2007.

[127] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. Correcting Soft Errors Online in Fast Fourier Transform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 30. ACM, 2017.

[128] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *Proc. SC*, page 107, 2006.

[129] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin. Power-efficient time-sensitive mapping in heterogeneous systems. In *Proc. PACT*, pages 23–32, 2012.

[130] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin. Exploiting barriers to optimize power consumption of CMPs. In *Proc. IPDPS*, page 5a, 2005.

[131] Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. Profiling high performance dense linear algebra algorithms on multicore architectures for power and energy efficiency. *Computer Science - R&D*, 27(4):277–287, 2012.

[132] Y. Luo, V. Packirisamy, W.-C. Hsu, and A. Zhai. Energy efficient speculative threads: Dynamic thread allocation in same-ISA heterogeneous multicore systems. In *Proc. PACT*, pages 453–464, 2010.

[133] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[134] Timothy C May and Murray H Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, 1979.

[135] Sarah E Michalak, Kevin W Harris, Nicolas W Hengartner, Bruce E Takala, and Stephen 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*, 5(3):329–335, 2005.

[136] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *CoRR*, abs/1404.4629, 2014.

[137] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proc. ICS*, pages 35–44, 2002.

[138] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. The soft error problem: An architectural perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243–247. IEEE, 2005.

[139] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[140] Eugene Normand. Single event upset at ground level. *IEEE transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[141] Heather M Quinn, Dolores A Black, William H Robinson, and Stephen P Buchner. Fault simulation and emulation tools to augment radiation-hardness assurance testing. *IEEE Transactions on Nuclear Science*, 60(3):2119–2142, 2013.

[142] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya. Some observations on optimal frequency selection in DVFS-based energy consumption minimization. *Journal of Parallel Distributed Computing*, 71(8):1154–1164, August 2011.

[143] Nikzad Babaii Rizvandi, Javid Taheri, and Albert Y Zomaya. Some observations on optimal frequency selection in dvfs-based energy consumption minimization. *Journal of Parallel and Distributed Computing*, 71(8):1154–1164, 2011.

[144] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making DVS practical for complex HPC applications. In *Proc. ICS*, pages 460–469, 2009.

[145] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale MPI programs. In *Proc. SC*, pages 1–9, 2007.

[146] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2013.

[147] Alexander Scholl, Claus Braun, Michael A Kochte, and Hans-Joachim Wunderlich. Efficient on-line fault-tolerance for the preconditioned conjugate gradient method. In *On-Line Testing Symposium (IOLTS),21st International*, 2015.

[148] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM international conference on Supercomputing*, 2012.

[149] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 214–227. ACM, 2018.

[150] Daniel Siewiorek and Robert Swarz. *Reliable computer systems: design and evaluatuion*. Digital Press, 2017.

[151] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, 2012.

[152] Jared C Smolens, Brian T Gold, Jangwoo Kim, Babak Falsafi, James C Hoe, and Andreas G Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *ACM SIGPLAN Notices*, volume 39, pages 224–234. ACM, 2004.

[153] S. W. Son, K. Malkowski, G. Chen, M. Kandemir, and P. Raghavan. Reducing energy consumption of parallel sparse matrix applications through integrated link/CPU voltage scaling. *Journal of Supercomputing*, 41(3):179–213, September 2007.

[154] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In *Proc. PPoPP*, pages 230–238, 2006.

[155] Jingweijia Tan, Nilanjan Goswami, Tao Li, and Xin Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, 2011.

[156] Jingweijia Tan, Shuaiwen Leon Song, Kaige Yan, Xin Fu, Andres Marquez, and Darren Kerbyson. Combating the reliability challenge of GPU register file at low supply voltage. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 3–15. IEEE, 2016.

[157] L. Tan, L. Chen, Z. Chen, Z. Zong, R. Ge, and D. Li. Improving performance and energy efficiency of matrix multiplication via pipeline broadcast. In *Proc. CLUSTER*, pages 1–5, 2013.

[158] L. Tan, L. Chen, Z. Chen, Z. Zong, R. Ge, and D. Li. HP-DAEMON: High performance distributed adaptive energy-efficient matrix-multiplication. In *Proc. ICCS*, pages 599–613, 2014.

[159] L. Tan and Z. Chen. Algorithmic energy saving for parallel Cholesky, LU, and QR factorizations. *Journal of Parallel Distributed Computing*, 2015.

[160] L. Tan and Z. Chen. Slow down or halt: Saving the optimal energy for scalable HPC systems. In *Proc. ICPE*, 2015.

[161] L. Tan, Z. Chen, Z. Zong, R. Ge, and D. Li. A2E: Adaptively aggressive energy efficient DVFS scheduling for data intensive applications. In *Proc. IPCCC*, pages 1–10, 2013.

[162] L. Tan, M. Feng, and R. Gupta. Lightweight fault detection in parallelized programs. In *Proc. CGO*, pages 1–11, 2013.

[163] L. Tan, S. R. Kothapalli, L. Chen, O. Hussaini, R. Bissiri, and Z. Chen. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. *Parallel Computing*, 2014.

[164] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In *Proc. IPDPS*, 2015.

[165] Li Tan and Zizhong Chen. Slow down or halt: Saving the optimal energy for scalable hpc systems. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 241–244. ACM, 2015.

[166] Li Tan, Shuaiwen Leon Song, Panruo Wu, Zizhong Chen, Rong Ge, and Darren J Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing.

[167] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. Improving performance of iterative methods by lossy checkponting. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 52–65. ACM, 2018.

[168] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z. Zhang, Darren Kerbyson, and Zizhong 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.

[169] Y. Taur, X. Liang, W. Wang, and H. Lu. A continuous, analytic drain-current model for DG MOSFETs. *IEEE Electron Device Letters*, 25(2):107–109, February 2004.

[170] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe 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*, pages 331–342. IEEE, 2015.

[171] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[172] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE*

*IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[173] Timothy Tsai, Nawanol Theera-Ampornpunt, and Saurabh 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*, pages 1–8. IEEE, 2012.

[174] A. H. T. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk. Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters. In *Proc. FPT*, pages 233–240, 2010.

[175] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and J. Bruce. A control-theoretic approach to dynamic voltage scheduling. In *Proc. CASES*, pages 255–266, 2003.

[176] D. M. Wadsworth and Z. Chen. Performance of MPI broadcast algorithms. In *Proc. PDSEC, with IPDPS*, pages 1–7, 2008.

[177] X. Wang, X. Fu, X. Liu, and Z. Gu. Power-aware CPU utilization control for distributed real-time systems. In *Proc. RTAS*, pages 233–242, 2009.

[178] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. OSDI*, page 2, 1994.

[179] A. S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824–834, September 2004.

[180] Panruo Wu and Zizhong 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.

[181] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. Silent data corruption resilient two-sided matrix factorizations. In *Proceedings of the 22nd Principles and Practice of Parallel Programming*, 2017.

[182] Panruo Wu, Chong Ding, Longxiang Chen, Teresa Davies, Christer Karlsson, and Zizhong Chen. On-line soft error correction in matrix–matrix multiplication. *Journal of Computational Science*, 2013.

[183] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong 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.

[184] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong 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.

[185] Erlin Yao, Jiutian Zhang, Mingyu Chen, Guangming Tan, and Ninghui Sun. Detection of soft errors in lu decomposition with partial pivoting using algorithm-based fault tolerance. *The International Journal of High Performance Computing Applications*, 29(4):422–436, 2015.

[186] Q. Zhu, J. Zhu, and G. Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *Proc. SC*, pages 1–12, 2010.

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