FAULT TOLERANCE THROUGH INVARIANT CHECKS IN APPLICATIONS USING LINEAR ALGEBRAIC METHODS

By

Felix Da Yuan Loh

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Electrical Engineering)

at the
UNIVERSITY OF WISCONSIN–MADISON
2018

Date of final oral examination: 06/11/2018

For my parents, Leong Eam Loh and Cecilia Swee See Tan

ACKNOWLEDGMENTS

It is fairly common knowledge that earning a PhD can be a long, arduous and difficult journey. This has certainly been my experience, but I have also discovered that the experience has been fruitful in many ways. I would like to take this opportunity to acknowledge several people who have been instrumental in my life as a PhD student, for without them, I would not have been successful in completing my degree.

First and foremost, my wonderful parents, Dr. Leong Eam Loh and Cecilia Swee See Tan. Without their financial, emotional and spiritual support, I definitely would never have gotten this far. I cannot thank them enough. Many words of thanks are also in order for my sister Beatrice Loh, my brother-in-law Kendrick Ling and my nephew Jonathan Ling; I greatly appreciate their kind support and encouragement.

I am also deeply grateful to my advisors, Professor Parameswaran Ramanathan and Professor Emeritus Kewal Saluja. They have been very patient, and have provided me with invaluable advice and encouragement throughout the course of my PhD studies. I would also like to thank the remaining members of my thesis committee: Professor Yu Hen Hu, Professor Mikko Lipasti and Professor Dan Negrut. Their insightful comments and suggestions have assisted me in improving the quality of this dissertation.

I am fortunate to have been given the opportunity to work as a Teaching Assistant for several undergraduate courses under the direction of Joseph Krachey, an instructor and Faculty Associate in my department. Joseph has been helpful in mentoring and inspiring me to become a better teacher. His friendly and approachable manner made it comfortable for me to discuss issues that arose during the course of my teaching duties, and he even nominated me for a teaching award.

And last but not least, I would like to thank my colleagues, past and present: Ahmed Abulila, Rehan Ahmed, Daniel Chang, Amrita Chowdhury, Spencer Millican, Daniel Seemuth, Hsiang-Kuo Tang, Ashwin Varadarajan, Wen Yan and Nairan Zhang. Our conversations were not always centered around research, but they made my long hours in the lab interesting and enjoyable.

ABSTRACT

Graphics processing units (GPUs) have become a popular platform for scientific computing applications, many of which are based on linear algebra. As the minimum feature size of transistors decreases, GPUs are becoming more vulnerable to transient faults caused by events such as alpha particle strikes, power fluctuations and electronic noise. In addition, the likelihood of a fault increases as more GPU computing nodes are used in supercomputers to meet the increasingly demanding computational requirements of scientific applications. Consequently, there are concerns that GPU-based supercomputer systems will suffer from very high fault rates. In order to ensure reliability, it is necessary to use fault tolerance (FT) techniques.

This thesis presents low-overhead FT techniques for several commonly-used linear algebraic applications that run on GPUs, focusing mainly on applications that operate with sparse matrices. These FT techniques exploit the invariant properties of the algorithms used in these applications, and exploit the parallel execution model of GPUs to allow for low-overhead error detection.

This thesis introduces and studies efficient error checking schemes for three popular matrix factorization techniques: Householder QR factorization, left-looking Cholesky factorization, and right-looking LU factorization. It also explores lightweight invariant checking methods for the preconditioned conjugate gradient (PCG) and biconjugate gradient stabilized (BiCGSTAB) iterative solvers and introduces an efficient checking method for the Lanczos eigensolver, as well as fault injection mechanisms for NVIDIA GPUs that allow for the simulation of transient, non-instantaneous faults.

This thesis carefully evaluates these FT methods on a contemporary NVIDIA GPU platform, and the results show that the aforementioned error checking strategies have high error coverage and are significantly more efficient than prior FT techniques on a GPU system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

Linear algebra is fundamental to many computing applications, most of which are essential to modern life. These include applications in artificial intelligence, bioinformatics, computer architecture, cryptography, data science, economics, machine learning, signal processing, sociology and numerous other fields. The following paragraphs briefly describe some practical applications of linear algebra.

Signal Processing: A continuous-time linear time invariant (LTI) system can be represented as a first order vector differential equation. This system can then be analyzed using various linear algebraic techniques.

Machine Learning: Most machine learning techniques involve some form of matrix factorization, as well as numerical optimization. These numerical optimization techniques often require singular value decomposition which can be computed, in part, by using an eigensolver.

Bioinformatics: Vector spaces are used for defining biological structures in a data format that is suitable for representation on computers. For example, the National Center for Biotechnology

Information (NCBI) has a publicly available search tool, VAST+ [1], for a comprehensive archive of protein structure similarities that is stored in the Molecular Modeling Database (MMDB). This data is represented in the form of vectors.

Data Science: Natural language processing frequently uses latent semantic analysis. This involves using singular value decomposition to obtain a low-rank approximation of a given term-document matrix.

Sociology: The relationships between individuals in a social group can be represented as a graph, which can then be analyzed using the adjacency matrix of that graph.

In recent years, graphics processing units (GPUs) have become a popular platform for scientific computing applications (many of which are based on linear algebra), and are increasingly being used as the main computational units in supercomputers. This trend is expected to continue as the number of computations required by scientific applications reach petascale and even exascale range [2]. As the minimum feature size of transistors decreases, GPUs are becoming more vulnerable to transient faults caused by events such as alpha particle strikes, voltage droops and electronic noise. To make matters worse, the likelihood of a fault increases as more GPU computing nodes are used in parallel to meet the increasingly demanding computational requirements of scientific applications. Consequently, there are concerns that exascale systems will suffer from very high fault rates [3]. This necessitates the use of fault tolerance (FT) techniques.

Figure 1.1, reproduced from Shivakumar et al [4], illustrates the extent by which faults are expected to affect modern processors, including GPUs. Figure 1.1 projects that over the period from 1992 (600 nm process technology) to 2011 (50 nm process technology), the soft error rate (SER)

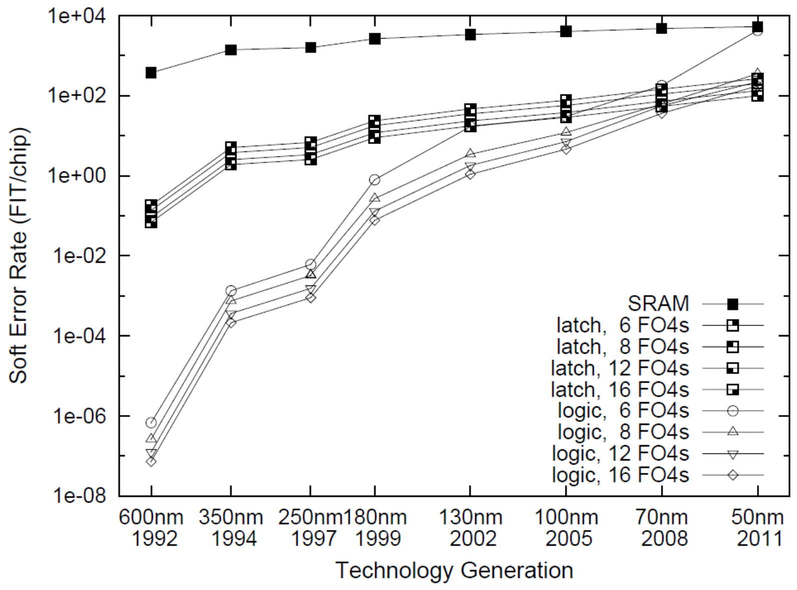


Figure 1.1 Projected soft error rate of processor chips.

per chip would have increased by nine orders of magnitude, from approximately 10^{-7} failures in time (FIT) to around 10^2 FIT, where FIT is defined as the number of failures per 10^9 hours of operation per chip. Current state-of-the-art NVIDIA GPUs (for example, the GTX 1080) are based on the Pascal architecture [5]. These use 16 nm process technology. By extrapolating from figure 1.1, one can expect the SER of this generation of GPUs to be as high as 10^8 FIT. This implies that a single GPU chip might encounter up to 2.4 errors per day.

Recently, researchers [6] have observed that the single bit error occurrence frequency on the Titan supercomputer at Oak Ridge National Laboratory, which features GPUs as its main computational units, appears to be highly correlated with users and applications. These researchers advocate the use of application-centric GPU error resilience techniques. Moreover, hardware FT techniques, such as dual modular redundancy (DMR), are typically expensive and consume valuable, limited space on a chip. In the case of GPUs, which rely on a large number of parallel logic circuits to provide fast performance, this cost can be unacceptable. Software FT techniques, including algorithm-based fault tolerance (ABFT), are an important and necessary approach for providing reliable computation on GPUs without sacrificing valuable chip space. Therefore, scientific computing applications that run on GPUs should be augmented with software-based FT mechanisms. In addition, GPUs have a large number of streaming processors (SPs) and the impact of a transient fault affecting a single SP of a GPU can be very different from the observed impact of a fault affecting a single core of a conventional multi-core CPU.

There are several existing linear algebra software collections. A prominent example is SuiteSparse [7], which is a comprehensive collection of linear algebraic applications that are optimized for sparse matrices. SuiteSparse and other packages depend on libraries that provide the basic linear algebra routines necessary for higher-level applications. One such library is LAPACK [8], which is a well-known linear algebra library that is the successor to the LINPACK and EISPACK libraries, and it contains low-level basic linear algebra subprograms (BLAS) routines commonly used in higher-level linear algebraic applications that run on contemporary CPUs. LAPACK also includes some higher-level routines, like Cholesky and QR factorization. Similarly, cuBLAS [9] is a general low-level BLAS library optimized for use on NVIDIA GPUs, while cuSPARSE [10] is a library for NVIDIA GPUs that is tailored towards low-level basic linear algebraic operations involving sparse matrices. However, to the best of our knowledge, there has been no comprehensive fault tolerant linear algebra software collection (particularly for sparse matrices) that can be utilized to provide low-overhead fault tolerance for critical applications that make significant use of certain linear algebra routines.

In this dissertation, we develop low-overhead fault tolerance techniques for several commonly-used linear algebraic applications on GPUs. We introduce and evaluate efficient error checking schemes for three widely-used matrix factorization techniques: QR factorization, Cholesky factorization, and LU factorization. We also propose low-overhead invariant checking methods for the preconditioned conjugate gradient (PCG) and the biconjugate gradient stabilized (BiCGSTAB) iterative solvers and introduce an efficient checking method for the Lanczos eigensolver. Non-FT versions of these applications are widely available in linear algebraic suites such as the aforementioned SuiteSparse and LAPACK. Our FT techniques exploit the invariant properties of the algorithms used in those applications, and exploit the parallel execution model of GPUs to allow for low-overhead error detection. We mainly focus on applications that involve sparse matrices. These applications are typically more complicated than their counterparts that work on dense matrices for several reasons. Firstly, data representations for sparse matrices are more complex (and are not standardized). Secondly, on a GPU platform, exploiting parallelism using sparse matrix data representations is more difficult, compared to dense matrix representations. Finally, in the case of applications that involve matrix factorization, fill-in of the factor matrices has to be accounted for, which is a non-trivial task. We develop our own implementations for most of the linear algebraic applications, and use open-source third-party applications for others.

1.2 Contributions of This Dissertation

The contributions of the dissertation are the following:

1. We propose fault tolerant techniques for three commonly-used matrix factorization methods: block Householder QR factorization [11], sparse supernodal left-looking Cholesky factorization and sparse right-looking LU factorization [12]. Specifically, for these matrix factorization methods, we identify invariant properties that can be used to detect errors at the end of each iteration of the factorization method. Householder QR factorization is a numerically stable matrix factorization technique suitable for square matrices. Cholesky factorization is also numerically stable, but it is restricted to symmetric positive definite matrices. LU factorization can be used with any general matrix, but it is not numerically stable and requires partial pivoting to ensure stability. For each of these factorization methods, we investigate various fault tolerant schemes and evaluate these schemes on a GPU platform, with and without the injection of faults. We show that our fault tolerant schemes have low overhead and high error coverage on GPUs.

2. We introduce and evaluate lightweight checking methods for two iterative solvers: the PCG method and the BiCGSTAB method [13]. These solvers are a popular alternative to traditional direct methods for solving a linear system $Ax = b$. PCG requires the matrix A to be symmetric positive definite, while BiCGSTAB only requires that the matrix is positive definite. We identify various invariant properties that exist in these iterative solvers, and then discuss efficient error checking mechanisms for these solvers, using the invariant properties.

We also investigate the behavior of these solvers on a GPU platform, with and without the injection of faults. We demonstrate that these methods provide robust protection from faults and have low overhead.

3. We propose an efficient orthogonal checking method for the Lanczos eigensolver, together with a simple but robust recovery scheme. The Lanczos eigensolver is a popular iterative method for finding some maximal eigenvalues of a symmetric matrix. We evaluate our fault tolerant method on a GPU platform, in the absence and presence of faults, and show that the checking overhead has very low overhead and good error coverage.

4. We develop transient fault injection mechanisms for NVIDIA GPUs that allow for the simulation of transient, non-instantaneous faults. This fault model has an advantage over prior models in that it can simulate faults that last longer than an instant, which do occur in real life.

1.3 Organization of This Dissertation

The remainder of this dissertation is organized as follows: Chapter 2 discusses important background material on GPU architecture and the nature of faults. It also provides an overview of research that is related to our work. Chapter 3 discusses our fault tolerant strategies for the block Householder QR factorization method. Chapter 4 explains our fault tolerant methods for the left-looking Cholesky factorization method, while Chapter 5 discusses our strategies for providing fault tolerance for the right-looking LU factorization method. Chapter 6 explains our fault tolerant methods for the PCG and BiCGSTAB iterative solvers. Chapter 7 describes how we provide fault tolerance for the Lanczos eigensolver. Chapter 8 concludes the dissertation and discusses future work.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Graphics Processing Units (GPUs)

A GPU is a processor that is capable of executing a very large number of threads in parallel. It functions as a co-processor to the CPU, possessing a large number of processing elements arranged in a highly parallel manner, with a wide memory bus and fast off-chip memory. The CPU communicates with the GPU via the *PCI-Express* interface [14], and through this interface, the host CPU offloads data and instructions to be executed on the GPU.

Our work is centered on NVIDIA GPUs based on the Fermi architecture [15], like the GTX 480, as well as those based on the Pascal architecture [5], such as the GTX 1080.

The GTX 480 is capable of performing about 1.3 teraFLOPS and has 15 streaming multiprocessors (SMs). Each SM possesses 32 streaming processors (SPs) and each SP can carry out two single-precision fused multiply-add operations per second. In addition, each SM has 16 LD/ST units (for memory operations), 4 special function units (for complicated operations such as sine and cosine), a 32,768 $\times$ 32-bit register file, 64KB of shared memory/L1 cache and 768KB of L2 cache. The GTX 480 also has 1.5GB of off-chip memory.

The GTX 1080 has 20 SMs and can perform about 8.2 teraFLOPS. Each SM possesses 128 SPs, for a total of 2560 SPs available in the GTX 1080 GPU. Each SM also has 32 LD/ST units, 32 special function units, a 256KB register file, 96KB of shared memory and 48KB of L1 cache, as well as 8 texture units. The GTX 1080 has 8GB of off-chip memory.

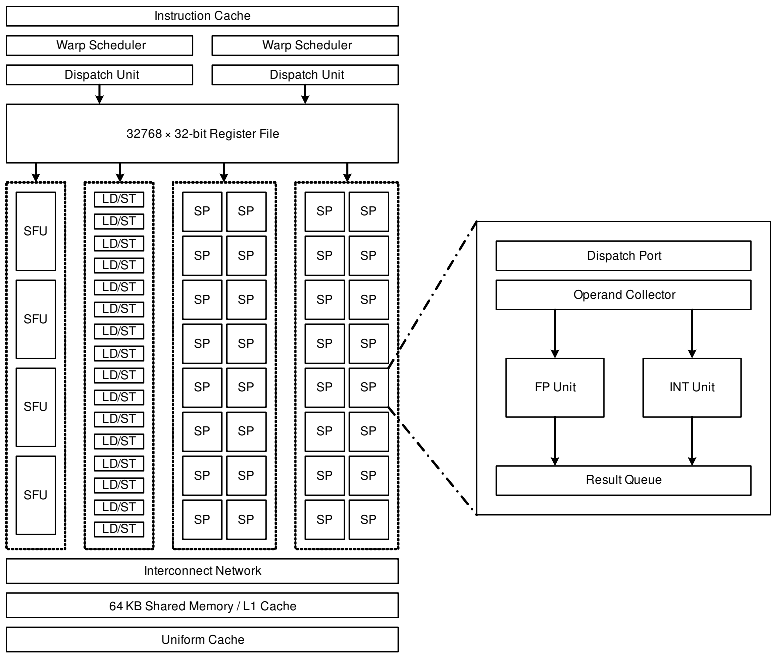


Figure 2.1 A single SM of the GTX 480.

Contemporary NVIDIA GPU families, including those based on the Fermi and Pascal architectures, have support for ECC-protected register files, caches and off-chip memory. Each kernel executing on the GPU consists of a large number of threads that are grouped into thread blocks. Threads are executed in groups of 32 threads known as *warps*. Each thread within a warp executes in sync with the rest of the threads in that warp, in a SIMD fashion. Figure 2.1 shows one such streaming multiprocessor of the GTX 480, and Figure 2.2 shows one streaming multiprocessor of the GTX 1080.

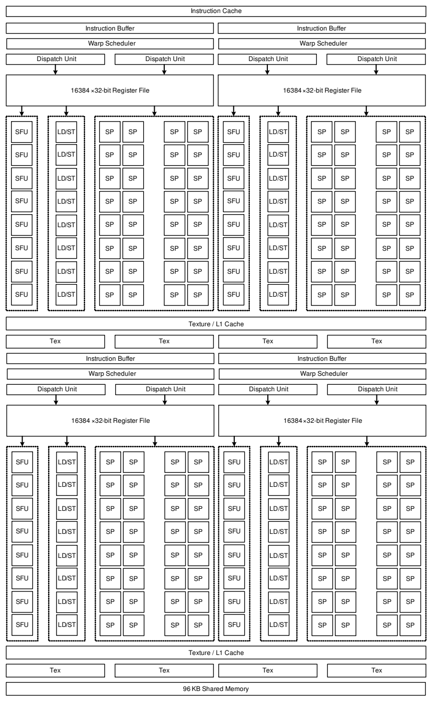


Figure 2.2 A single SM of the GTX 1080.

2.1.2 Sparse Matrix Formats

Our work in Chapter 3 uses dense matrices; however, in subsequent chapters, we deal exclusively with sparse matrices.

When the matrix is sparse and has large dimensions, it is impractical to store the matrix in a format that is more suited to dense matrices, such as the column major order format, as this wastes a significant amount of memory on elements that have zero values. Instead, it is more practical to store the matrix in a sparse matrix format, such as the compressed column storage (CCS) format or the compressed row storage (CRS) format. The CCS matrix format consists of three one-dimensional arrays. These are the *val* array, which stores the non-zero values of the matrix (grouped according to the columns of the matrix), the *row_idx* array, which contains the row indices of the non-zero values (also grouped according to the columns of the matrix), and the *col_ptr* array, whose elements point to the location of the first element of each column in the *row_idx* and *val* arrays. Figure 2.3 shows a sample matrix and its representation in the CCS format. Note that in the case of a matrix stored in the CCS format, row accesses should be avoided, as the rows of the matrix are not explicitly represented in the CCS format. Column accesses of a matrix in the CCS format, however, should be generally efficient. The reverse is true for matrices stored in the CRS format.

2.1.3 Performance of GPUs on Sparse Matrices

In general, an application that operates on sparse matrices can be expected to outperform its counterpart that operates on dense matrices, particularly if the input matrix is very large and sparse. This is mainly because the former only needs to perform computations for the non-zero elements of the matrix. However, it is important to note that there are some limitations on the performance of GPUs when the input matrix is in a sparse matrix format, such as the CCS format. This is mainly due to the fact that the actual values of the sparse matrix can only be accessed *indirectly* in memory, as opposed to dense matrix formats like the column major order format, whose values can be accessed directly. For example, in order to be able to do useful computations with the value elements of the *val* array of a sparse matrix in the CCS format, the GPU has to first access the *col_ptr* array and/or the *row_idx* array in order to determine the index of some value element. This has implications for GPUs, as their performance is memory-bound rather than compute-bound. In addition, the fact that indirect memory accesses are necessary means that the GPU is less able to exploit memory performance enhancing strategies such as memory coalescing. The performance of GPUs is also strongly dependent on the sparsity of the input matrix, as well as the extent of fill-in. In general, the sparser the input matrix (after fill-in), the better the GPU performance.

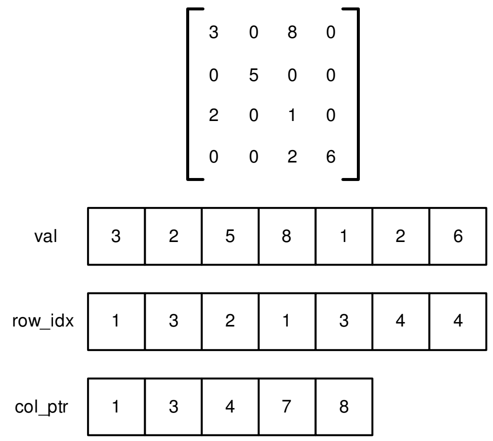


Figure 2.3 A sample matrix and its CCS representation.

2.1.4 Overview of Faults and Errors

It is important to distinguish between *faults* and *errors*, as they are not equivalent terms. A fault is a phenomenon that can lead to an error, while an error is a discrepancy between a computed value and its true value. Faults can be further classified into three categories: *transient*, *permanent* and *intermittent* [16]. A transient fault occurs during a single event and does not persist indefinitely. A permanent fault manifests at some point in time and persists from that point onwards. An intermittent fault occurs repeatedly, but does not persist continuously.

Sources of transient faults include the following: alpha particles, supply voltage droops, electromagnetic interference (EMI) from external entities, as well as EMI created by the chip itself – this is known as cross-talk. Meanwhile, sources of permanent faults include physical wear-out of the processor chip, fabrication defects in the chip during manufacture and flaws in the design of the processor. Intermittent faults can occur due to voltage and temperature fluctuations.

A non-permanent fault has an *active duration*, which is defined as the length of time that the fault manifests itself to the affected hardware. This term mainly applies to faults that are either intermittent, or are transient and *non*-instantaneous.

A fault may not necessarily result in an error, because the fault can be masked. For example, at the gate-level abstraction, a fault occurring at one of the inputs of a two-input OR gate, whose other input is set to a logical one, will never propagate to the output of that gate.

In our work, we focus on transient faults that affect the SPs of the GPU. We allow for the possibility of these faults to manifest as errors that can propagate to other processing elements and be written to memory; however, we limit the scope of our work to *exclude* cases where the errors arise from faults originating from the memory elements. Hence, we will not consider FT techniques that are specific to memory systems, such as parity bits and error-correcting code (ECC). Fortunately, those techniques are complementary to our work and can be used safely in conjunction with our proposed FT techniques. It is worth pointing out that memory-specific FT techniques, such as ECC, cannot protect against the types of errors that arise during execution in the processing units.

2.1.5 Faults and Soft Error Rate (SER) in Computational Units

There have been a few studies that investigate the nature and effects of transient faults, and estimate the likely SER of devices in the future. Ziegler et al [17] have reviewed experiments conducted at IBM in evaluating radiation-induced soft failures over a fifteen year period. Shivakumar et al [4] examine the effect of technology scaling and microarchitectural trends on the SER in memory and logic circuits. In particular, they predict that the SER per chip would have increased by nine orders of magnitude, from 10^{-7} FIT to 10^{2} FIT, over the period from 1992 to 2011. More recently, Nie et al [6] have conducted a large-scale study of soft errors on GPUs used in the Titan supercomputer, and observed that the single bit error occurrence frequency appears to be highly correlated with users and applications.

The effects of intermittent faults have also been studied. For example, Gil-Tomas et al [18] performed fault injection simulation of intermittent faults to examine their impact on microprocessors.

2.1.6 Transient Fault Injection Mechanism

Our fault injection mechanism is capable of simulating transient hardware faults that can persist for a substantial length of time (e.g. $500\mu s$), which affect the processing elements of an NVIDIA GPU and lead to silent data corruption (SDC). Our fault model considers only faults that occur in the SP cores during execution and that data received from the register files, caches and off-chip memory are initially error-free. We allow for the possibility that corrupted results can be written to memory by threads running on faulty SPs. Our fault injection mechanism utilizes CPU interrupt-based timers to simulate the active duration of a transient fault. It can also select a particular SP core (within a particular SM) to inject the fault into. Using this method, we can simulate transient faults that last somewhat longer than an instant. We use SIGRTMIN as the interrupt signal. We have two versions of this mechanism.

The strategy of the first version is as follows: At the beginning of the main program, we create and initialize timer constructs for the fault start time and fault active duration (called *faultStartTimer* and *faultActiveDurTimer* respectively). On the GPU side, we initialize the *fault active* flag in the off-chip memory to 0. We start the timer for *faultStartTimer* and once it expires, the CPU sends an interrupt signal. When this happens, the interrupt handler on the CPU sets the *fault active* flag in off-chip memory to 1 – this signifies that a fault has occurred and is now active. At the same time, we start the timer for *faultActiveDurTimer*. Upon expiration of this timer, the CPU sends another interrupt signal, upon which the interrupt handler resets the *fault active* flag in off-chip memory back to 0, indicating that the active duration of the fault has expired.

The second version works as follows: At the beginning of each iteration of the program, we pick a random number which is uniformly distributed between 0 and 1. We then compare this value against a user-defined "fault activation" threshold. This threshold represents the average number of faults expected in an iteration. For example, if the threshold is 0.001, that means on average, one can expect one fault to occur every 1000 iterations. If the random number is smaller than the threshold, then we can activate the fault in one of two ways. In the first way, we set the *fault active* flag in the GPU off-chip memory to 1 – this signifies that a fault has occurred and is now active. At the same time, we start the timer for *faultActiveDurTimer*. Upon expiration of this timer, the CPU sends an interrupt signal, upon which the interrupt handler resets the *fault active* flag in off-chip memory back to 0, indicating that the active duration of the fault has expired. Alternatively, in the second way, instead of starting a timer to trigger an interrupt signal, we can simply allow the fault to persist for the duration of that iteration. Then at the beginning of the following iteration, we pick another random number, uniformly distributed between 0 and 1, and compare it against a *different* user-defined "fault deactivation" threshold (typically this will be much larger than the "fault activation" threshold). If the random number is smaller than this threshold, we deactivate the fault by resetting the *fault active* flag. Otherwise, the fault continues to be active and in the next iteration, we repeat the process to determine whether to deactivate the fault. This second way allows us to simulate faults with an active duration that can persist for a few iterations.

In both versions, the fault arrivals model a Poisson process with a user-specified average rate. Before a kernel function that is being executed on the GPU returns, each thread of that kernel checks to see whether the *fault active* flag in off-chip memory is 1 – if so, then that thread checks whether it is being executed on the SM and SP core that the fault is supposed to be injected into. If there is a match, then that thread either subtracts a random value from its result, or flips a predetermined bit in its 32-bit or 64-bit floating point result. The affected thread will only corrupt results that will be written to the off-chip memory.

2.1.7 A Note on Our FT Techniques

We will evaluate our FT techniques using the framework and assumptions laid out in the preceding sub-sections of this chapter. However, we would like to note that our FT techniques are actually independent of the data representation, the host platform, and the fault injection mechanism. We also wish to make some comments regarding errors that originate from the off-chip main memory: In general, our FT methods will also detect errors that only occur during the writing of data to memory. There are, however, some exceptions to this statement. Our FT techniques will not detect errors in the case where the input matrix is somehow corrupted in memory before any actual computation is initiated, since this is equivalent to operating on a different input matrix, and would not violate any invariant properties. In the case of applications involving matrix factorization, if some elements of the original input matrix become corrupted in memory during execution of the application, then our error checking methods for these applications will still be able to detect such errors, but recovery might not be possible.

2.2 Related Work

Von Neumann [19] is likely the first to address the issue of general purpose, reliable computing. He proposed the idea of using redundant logic components to ensure reliable computation, as long as the probability of failure of each component does not exceed 0.0073. Vaidya et al [20] subsequently studied the effectiveness of N-modular redundancy (NMR), where multiple processing elements are used to replicate computation and provide error detection and even error correction capabilities. In general, the overhead of this fault tolerant approach is very high but other researchers such as Jeon et al [21] have exploited the large number of SPs already present in GPUs to adapt this technique to GPUs.

Checkpointing [22] is an established, general method of recovering from an error once it has been detected. The system saves its state a regular intervals during program execution. Upon detection of a fault, the system can reset back to its last known good state and resume execution from that point.

Providing fault tolerance in linear algebra is not a new idea. Huang and Abraham [23] were the first to come up with a way to correct a single error in the multiplication of two dense matrices, by augmenting the input matrices with a checksum row and/or column. The elements in the checksum row and column of the result matrix are then compared to the sum of the elements in the corresponding row or column. The element that has been computed erroneously can then be identified by finding the row and column whose sum of elements do not match their corresponding row or column checksum.

2.2.1 Fault Tolerance in Matrix Factorization

Fault tolerant (FT) techniques for various matrix factorization methods have been discussed in the literature. Most of these techniques are based on the pioneering work done by Huang and Abraham [23] in providing fault tolerance for matrix-matrix multiplication through the use of checksums. For example, Jou and Abraham [24] investigate the use of a weighted checksum encoding scheme for various matrix factorization algorithms, including LU factorization. Hakkarinen et al [25] discuss and evaluate checksum techniques for Cholesky methods on distributed memory systems for dense matrices. However, they do not evaluate the effectiveness of these techniques on GPUs. Chen et al [26] implement and evaluate a checksum FT scheme for Cholesky factorization on MAGMA, a linear algebra library for heterogeneous CPU-GPU systems that is designed to work with dense matrices. However, this scheme is unlikely to have good performance for sparse matrices, because it requires dot product operations involving the multiplication of dense checksum vectors with several rows of the input matrix. These rows will be hard to efficiently access, since the sparse matrix is typically in the CCS format, in order to maximize performance. In addition, the checksum elements impose a significant storage overhead on the GPU memory. Our work does not suffer from these issues.

Du et al [27] use row checksums to protect the right factors R and U in dense QR factorization and dense LU factorization respectively, and use checkpoints for the corresponding left factors Q and L. In a similar fashion, Davies et al [28] use local and global row checksums to protect the

U factor during dense right-looking LU factorization on distributed systems. This row checksum method is unlikely to have good performance when partial pivoting is necessary. It will also not be efficient on sparse matrices, particularly if the rows are not explicitly defined in the sparse matrix format.

Wu et al [29] have implemented a package of fault tolerant routines for Cholesky, LU and QR factorization. However, their primary strategy for fault tolerance is to use row and column checksums for the matrices. We propose to use only invariant properties for error detection in our fault tolerant software collection. Moreover, Wu's fault tolerant routines can only tolerate one error per matrix row/column at a time. This may not be sufficient for GPUs, as multiple elements within the same row or column can possibly get corrupted by a single fault. Our fault tolerant schemes are able to tolerate such errors.

With respect to fault tolerance in block Householder QR, Du et al [30] implemented a technique for a hybrid CPU-GPU system where both the CPU and GPU are actively involved in factorizing A, and up to one error in R can be corrected *post*-factorization using two checksum columns appended to A and a series of Givens rotations. Q is protected using diskless checkpointing in CPU memory. Our techniques differ from theirs in that our schemes are geared towards implementations of the block Householder QR algorithm where the major computations are performed solely on the GPU. Our techniques are also able to correct multiple errors caused by a single fault in the GPU.

2.2.2 Fault Tolerance in Iterative Solvers

Several researchers have investigated fault tolerant schemes for iterative solvers. Hoemmen et al [31] propose the concept of allowing iterative solvers to apply variable reliability. This approach allows a program to perform most computations using a less reliable computing mode and some computations in a special, high reliability mode. In subsequent work, Elliot et al [32] use this principle to evaluate a version of the generalized minimal residual (GMRES) solver that uses a combination of an unreliable inner solver with a reliable outer solver.

Chen [33] proposes a conjugate check for Krylov subspace iterative solvers. However, Chen's method of fault injection for evaluation of the check has flaws. Firstly, the author corrupts a set of intermediate variables only by an error magnitude of 1.0 each, and he does not make it clear which elements are corrupted in variables that are vectors. This means that we cannot be certain of the effectiveness of the check when the variables are corrupted by other values of magnitude. Secondly, the author does not state how the faults are injected, or the iteration(s) in which the errors are introduced. Thus, one cannot be sure whether the check would be effective if faults occur during early iterations or during later ones.

Oboril et al [34] use numerical defect correction to improve the fault tolerance of the conjugate gradient solver. Bronevetsky et al [35] examine the effectiveness of checkpointing, checksum and residual tracking methods in a variety of iterative methods while Sloan et al [36] propose optimizations that are focused on checksum based fault detection in the matrix-vector multiplication operations of iterative methods.

Shantharam et al [37] investigate the effects of a single soft error propagating through a sequence of sparse matrix-vector multiplications, which is a common process in iterative solvers, and show that such an error can result in the corruption of the entire result vector within a short sequence. In a follow-up paper [38], the authors introduce a new sparse checksum encoding for all the key operations of the preconditioned conjugate gradient (PCG) iterative solver, particularly in sparse matrix-vector multiplication.

2.2.3 Fault Tolerance in GPUs

There has been work done on evaluating whether certain fault tolerant schemes work well on GPUs. Wunderlich et al [39] investigate the efficiency of triple modular redundancy (TMR) versus checksum-protected matrices in matrix multiplication on GPUs and show that the checksum method has lower overhead than TMR.

Researchers have also focused on the effect of faults on the memory structures of the GPU, including the off-chip memory. Shah et al [40] inject faults into the active mask stack, the register file and caches of AMD GPUs and analyze the vulnerability of these structures to the injected faults. Maruyama et al [41] propose using data encoding for detecting errors in the memory subsystem on GPUs, as well as checkpointing with re-execution for recovery when such errors are detected.

There is also research that investigate architectural approaches to providing fault tolerance in GPUs. Jeon et al [21] propose architectural modifications to GPUs to provide them with fault tolerance. Their idea is to allow the GPU to utilize unused execution lanes to verify the results of threads via dual modular redundancy (DMR). To do this, they enhance the register file with register forwarding units and include a replay queue to buffer unverified threads whenever execution resources are unavailable for redundant execution. Subsequent work by Dweik et al [42] and Abdel-Majeed et al [43] extend these modifications to allow for the correction of errors using TMR.

Palframan et al [44] propose fault tolerance in GPUs by using selective gate hardening and modification of the fused multiply-add unit to include checking of the exponent bits and the most significant bits of the mantissa in the floating point format, via redundant execution. They also introduce an encoding format for data stored in the register file such that the important bits of the data are stored in the hardened cells.

2.2.4 Fault Injection in GPUs

Some researchers have come up with software fault injection schemes specifically tailored for GPUs. Braun et al [45] used a simple fault injector (for matrix multiplication) that was able to inject faults into a target SM and one of its functional units. Ours can also target a specific SM and SP, but has the added capability of allowing for variable fault start time and active duration. Yim et al [46] developed a comprehensive fault injection tool that can be integrated into any GPU application, while Fang et al [47] use a GPU-based debugger to inject faults into various applications.

Tselonis et al [48] inject permanent faults into the architectural registers of a GPU model based on the Fermi architecture, using a GPU architectural simulator. This scheme has several weaknesses: Firstly, because the fault simulated are *permanent*, the authors' fault model cannot simulate temporal faults which are far more likely to affect a GPU. Secondly, the authors inject faults only into the register files of the GPU model. This may not accurately simulate errors which affect only the execution units of the GPU. Thirdly, by choosing to inject faults into a GPU model rather than an actual GPU, the author's fault model cannot capture fault-induced behavior in the architectural structures that are not modeled by the simulator. Our fault injection method does not suffer from these shortcomings.

Other researchers used beam testing to inject real faults into GPUs. For example, Rech et al [49] utilized neutron beams to subject NVIDIA GPUs to levels of radiation that are expected at sea level. However, they do not consider any fault tolerance techniques.

2.2.5 Other Fault Tolerance Techniques for Linear Algebraic Applications

Recognition, mining and synthesis (RMS) applications make significant use of linear algebra, and process large amounts of noisy data though the use of statistical and probabilistic computation. Shanbhag et al [50] have shown that the machine learning kernels underlying these applications are inherently resilient to small magnitude errors, and they proposed using statistical error compensation techniques to exploit this property. In subsequent work, Kim et al [51] design a machine learning accelerator core that uses both statistical error compensation and approximate computing techniques, and is energy efficient and error resilient.

Chapter 3

Fault Tolerance for Block Householder QR

Block Householder QR [11] has $O(n^3)$ complexity, so a checking scheme that has complexity $O(n^2)$ or lower order is necessary to avoid high overhead. While Householder QR has a convenient invariant property that $A = Q_k R_k$ at the end of any iteration k, simply using this property naively will incur an overhead of the order of $O(n^3)$, which is unacceptable. To address this, we implemented two low overhead checking schemes for block Householder QR that checks or compares only a subset of the elements of the matrices A, Q and R, and utilizes checkpointing techniques [22]. For this work, we developed our own GPU-based block Householder QR factorization program, with dense matrices stored in the row major order format.

3.1 Overview of Block Householder QR

QR factorization is an important application commonly used in solving large and dense linear systems of equations, or in solving linear least squares problems. It involves factoring an m-by-n matrix A as a product $A = QR$, where Q is an m-by-n orthogonal matrix and R is an n-by-n upper triangular matrix [52]. In this work, we use the block Householder implementation and limit our study to real-valued, dense square matrices (i.e. $m = n$).

The Householder QR algorithm applies a sequence of Householder reflections to the matrices Q and R in place, where Q is initialized to I and R is initialized to A. A Householder reflection is an orthogonal matrix of the form

$$P = I - \beta vv^T,$$

where $\beta = \frac{2}{v^Tv}$ and v is a Householder vector that is computed from a column of the matrix R such that PA will have its corresponding column filled with zeros below the diagonal. Successive operations will result in $R = P_{n-1}...P_1P_0A$ becoming an upper triangular matrix. Q, on the other hand, is post-multiplied with successive versions of P such that $Q = P_0P_1...P_{n-1}$ is an orthogonal matrix.

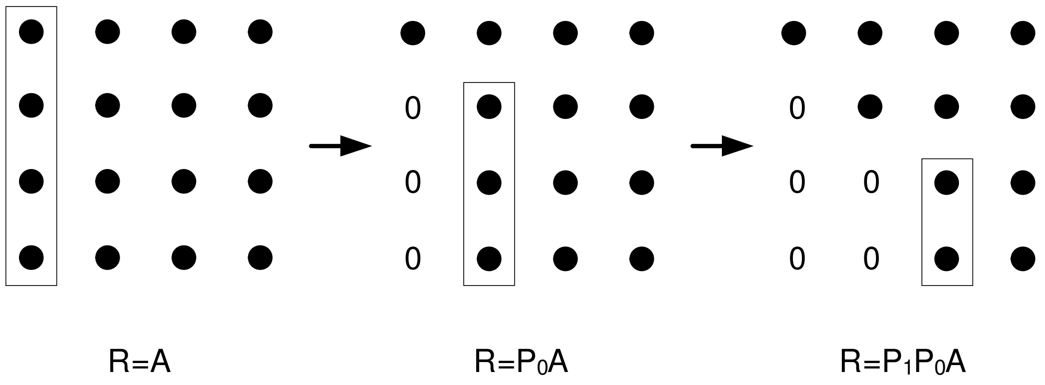


Figure 3.1 Upper Triangularization of R using Householder Reflections

Figure 3.1 shows how a series of Householder reflections are chosen from the columns of R to upper triangularize R. First, the entire leftmost column of R (initialized as A) is chosen to compute the Householder vector v_0. v_0 is then used to form the orthogonal matrix P_0 which is then pre-multiplied with R to form the matrix product P_0R (or P_0A). By overwriting the original R matrix with P_0A, zeros are placed below the diagonal in the leftmost column of R. The process is then repeated, this time selecting only the elements below (or at) the diagonal in the next column to compute the Householder vector v_1. This process continues until R assumes an upper-triangular form.

In practical implementations of the Householder QR algorithm, the Householder matrix P is *not* explicitly computed, because

$$\begin{aligned} PR &= (I - \beta v v^T)R \\ &= R - \beta v v^T R \end{aligned}$$

The order of operations is important. Multiplying v^T with R, and then multiplying v with the vector $v^T R$ results in the Householder QR algorithm having an overall computational complexity of $O(n^3)$. By contrast, if we instead choose to multiply v with v^T first and then multiply the matrix vv^T with R, the algorithm will have a computational complexity of $O(n^4)$.

The block Householder QR algorithm was designed by Bischof and van Loan [53], shown in Algorithm 1. Because the basic Householder QR algorithm is dominated by matrix-vector multiplications, it is relatively inefficient for GPU architectures due to the low amount of computation for each data fetched from the off-chip memory of the GPU. The block Householder QR algorithm addresses this shortcoming by combining r Householder reflections into a single block operation, such that it is dominated by matrix-matrix multiplications:

$$\begin{aligned} P_{WY} &= P_0 P_1 ... P_{r-1} \\ &= (I - \beta_0 v_0 v_0^T)(I - \beta_1 v_1 v_1^T)...(I - \beta_{r-1} v_{r-1} v_{r-1}^T) \\ &= I + WY^T, \end{aligned}$$

where W and Y are n-by-r matrices. Effectively, this means that the columns of R are partitioned into n/r blocks of r columns each. For each block, r Householder vectors are computed and applied to the corresponding columns of R to upper-triangularize them, then the W and Y matrices for that block are computed and applied to the *remaining* columns of R, as well as to Q. This algorithm has $O(n^3)$ complexity. For more details, we refer the interested reader to the comprehensive overview by Kerr et al [54]. Note that the block Householder algorithm utilizes the *house()* function, shown in Algorithm 2.

3.2 Fault Tolerance Schemes for Block Householder QR

Here, we discuss a redundant fault tolerant scheme and then describe a low cost method of checking, called full check, and a fault tolerant scheme, FC-O, that makes use of the full check. Finally, we discuss a partial check method and two schemes which use this method: PC-C and PC-CS.

3.2.1 Redundant Scheme

In the Redundant scheme, we duplicate all of the computations involved in factorizing A. This is similar to the R-Thread approach used by Dimitrov et al [55]. In this scheme, no checkpoints are taken. At the end of the *last* iteration, a full check is performed on the Q and R matrices. The computational complexity of this scheme is high, because additional resources will be required to handle the redundant computations. However, there is no overhead for recovery. The complexity of doing the single full check is $O(n^2)$, as discussed in the next section.

3.2.2 Full Check

Checking each and every element of the product QR with the corresponding element in A is expensive, because obtaining the product QR requires $O(n^3)$ operations. One can reduce the cost by comparing each of the n elements in the vector Ax with its corresponding element in the vector QRx, where $x = [1 \;\; 1 \, ... \, 1]^T$. When computing Ax and QRx, we use *all* elements in each row of A and QR. This is a row checksum comparison. During each comparison, if the relative difference between the corresponding elements of Ax and QRx exceeds an error bound ϵ, then an error is said to have occurred. The absolute difference is used when the magnitude of the element of Ax is below a threshold δ. Each full check has $O(n^2)$ complexity (I compute Rx before QRx), but has practically 100% error coverage. A further optimization can be made by simply adding all elements within each row of R to obtain Rx.

3.2.3 FC-O Scheme

The *Full Check Only* scheme, shown in Algorithm 3, does not take any checkpoints for Q and R, and a full check, as explained above, is performed at the end of the last iteration. If an error is detected during this check, then Q and R are reset back to I and A respectively and the QR factorization is restarted. While the computational complexity of checking is only $O(n^2)$ (since there is only one full check), the overhead of recovery is high.

3.2.4 Partial Check

The full check has excellent error coverage. However, if one were to use the full check at the end of *every* iteration i of the block Householder QR factorization, the *overall* cost of doing the checks will be $O(n^3)$, i.e. the computation cost of doing the full checks is in the same order as the block Householder QR. This overhead is unacceptable for very large values of n, particularly at the petascale and exascale range. In this section, we propose a partial check that trades off the computational cost of checking with error coverage. Instead of computing the checksums for all elements within a row and for all rows of A, the partial check computes the checksums for only a subset of elements.

The partial check operates *exactly* the same as the full check, with the following exception: The full check computes Ax and QRx using elements 0 to $n-1$ of each row of A and QR. By contrast, the partial check computes the checksums using only elements within the sth and $(s+r-1)$th rows and columns of A, along with the necessary elements from Q and R, as shown in Figure 3.2. Note that in this figure, $s = ir$. This also means that only the sth to $(s+r-1)$th elements of the vectors Ax and QRx are compared. Let us consider just the number of floating point multiplications involved in the multiplication of Q with Rx. At the end of the ith iteration, $r^2(i+1)$ multiplications are needed to do a partial check. Thus, the *overall* complexity of performing this check at the end of every iteration is $O(n^2)$, since

$$\sum_{i=0}^{\frac{n}{r}-1} r^2(i+1) = \frac{1}{2}n(n+r)$$

This is about n times better than doing a full check at the end of every iteration. Like the full check, this check is implemented as a set of GPU kernels. Similar to the full check, Rx can be computed by adding up the required subset of elements within the same row of R, for each of the rows of R that will be checked. In the next two subsections, we discuss two variations of block

Householder with checkpointing that make use of both the partial check and full check for error detection.

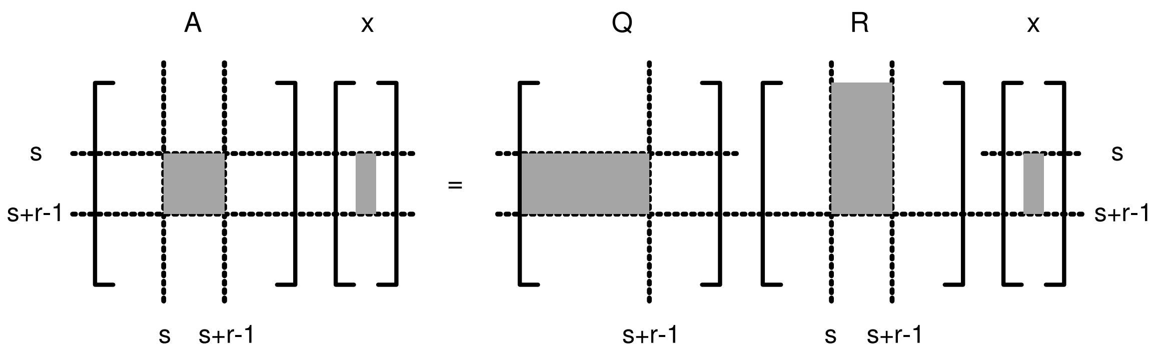


Figure 3.2 Elements involved in the computations for the partial check (shaded regions)

3.2.5 PC-C Scheme

In the *Partial Check with Checkpointing* scheme, shown in Algorithm 4, checkpoints [22] are used for Q and R. A partial check is conducted on Q and R at the end of every iteration except the last. If errors are detected by the partial check in any iteration, then Q and R are replaced by the data in their respective checkpoints, and the computation for that iteration is re-executed. If no errors are detected, then the checkpoints for Q and R are updated with the data in Q and R. Since the partial check has limited error coverage, there is a distinct possibility that there are undetected errors in Q and/or R – this means that the checkpoints will be corrupted. To mitigate this, we have a special flag that is set whenever the partial check *does* detect an error. If upon re-execution of that same iteration, errors are *again* detected, it means that in some prior iteration, the partial check for that iteration failed to detect any errors when there were actually errors in Q and/or R. By checking the value of the flag, one will know that corruption of the checkpoints have occurred. In this case, we simply reset Q and R back to I and A respectively and restart the block Householder QR computation from the beginning. Note that this flag is reset when the partial check detects no errors, or when the QR computation is restarted. In the last iteration, a full check is done to ensure that there are no undetected errors that were missed by the partial checks. The complexity of checking in this scheme is $O(n^2)$. The overhead of recovery will be low if the checkpoints are not corrupted. Otherwise, the overhead will be greater, but in the absolute worst case, no higher than the recovery overhead of FC-O.

3.2.6 PC-CS Scheme

The *Partial Check with Checkpoint Swapping* scheme is basically the same as PC-C, except that it does not explicitly take checkpoints at the end of each iteration. Instead, we use additional matrices Q_{alt} and R_{alt}. Q and Q_{alt} swap between the roles of source matrix and destination matrix after each iteration. The same process occurs for R and R_{alt}. For example, during even iterations, Q and R are used as the source matrices while Q_{alt} and R_{alt} serve as the destination matrices. If an error is detected during an even iteration, Q and R serve as checkpoint matrices. The advantage of this approach is that it avoids the need to explicitly copy the respective data in Q and R over to their checkpoint matrices. However, there are some issues worth pointing out. Because the destination matrices change between even and odd iterations, the correctly computed elements end up being distributed between Q and Q_{alt}, as well as between R and R_{alt}. This requires the ‘assembly’ of the ‘final’ Q and R after factorization is complete. Figure 3.3 shows where the correct elements are located in each of the destination matrices at the end of the last iteration. Also, in the event that an error occurs, the partial check needs to use the appropriate subset of elements from Q and Q_{alt}, as well as from R and R_{alt} for checking. During recovery, the recovery kernels also need to pick the right subset of elements from these four matrices. The overhead for this scheme is expected to be lower than that of PC-C.

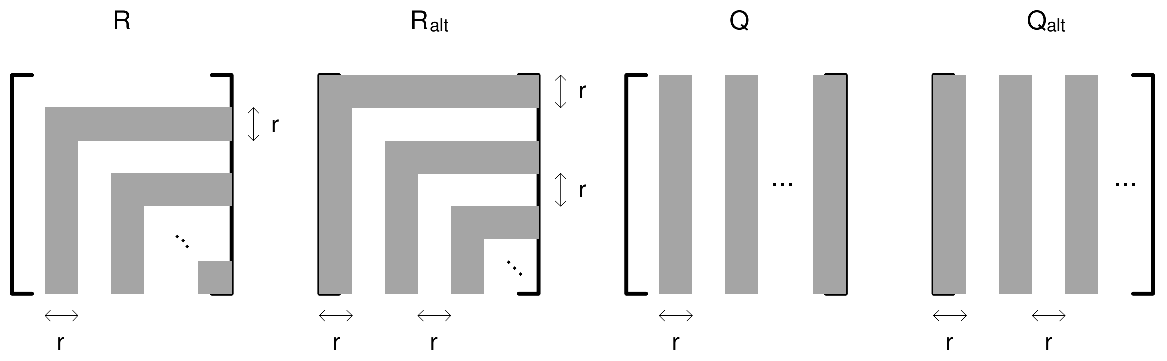


Figure 3.3 Locations of the correctly computed elements in the destination matrices (shaded regions)

3.3 Performance of Fault Tolerance Schemes

Our experimental methodology is as follows: each individual run uses a randomly generated input matrix A whose individual elements are single-precision floating point numbers uniformly distributed between -25 and 25. An additional random number is generated, uniformly distributed between 1 and 100, for determining the magnitude of the fault if it is activated. We conduct experimental runs for each of the four techniques described above. Each of these runs were conducted using four input matrix sizes ($n \times n$) with $n = 512, 1024, 2048$ and 4096. We set δ to 0.5, and the round-off error bound, ϵ, to $4 \times 10^{-6}n$, and allow at most one fault to occur during the lifetime of an experiment. The experiments can be classified into two categories: *fault activated* and *fault not activated*. We also run a series of baseline experiments to determine the average baseline execution time with *no* fault injection or error detection mechanism.

For the *fault activated* experiments, we choose six target fault start time values for each matrix size, spaced evenly within the expected run time for block Householder QR factorization. Each experiment uses a fixed target fault active duration of $100\mu s$. We run at least 1000 experiments for each technique and at each matrix size. Among all runs in these experiments, we select only those in which the fault got activated. We have a similar process for the *fault not activated* experiments, except that we set the target fault active duration to a sufficiently small value, such that the majority of the experiments has a unactivated fault. Among all runs in these experiments, we select only those in which the fault did *not* get activated.

We first examine the *fault activated* category for both the PC-C and PC-CS schemes. The number of cases in which the fault was activated (leading to at least one error) can be broken down into four main cases. We list these cases along with the number of occurrences in parenthesis: errors were detected and corrected successfully (4838), errors were detected successfully but not corrected (0), errors were not detected (0) and benign fault (7). In some rare instances, a benign fault occurred. A benign fault is one which causes errors that are indistinguishable from round-off errors. The vast majority of faults were successfully detected and corrected by either the partial check or the full check. There were no cases in which the errors went undetected or were uncorrected. Table 3.1 summarizes these results.

Table 3.1 Outcomes for PC-C and PC-CS schemes where fault was activated

Case	**Number**
Errors detected and corrected successfully	4838
Errors detected successfully but not corrected	0
Errors were not detected	0
Benign Fault	7

One can take a further look at the outcome in which errors were detected and corrected successfully, which are summarized in Table 3.2. This outcome (total of 4838 cases) can be further classified into three possible cases: The partial check successfully detected the errors in the same iteration in which they occurred (3745), the partial check successfully detected the errors but only detected these errors in a subsequent iteration (984), and only the full check detected the errors (109).

Table 3.2 Further analysis of the case where errors were detected and corrected successfully

Case	**Number**
Partial check detected within same iteration	3745
Partial check detected in a later iteration	984
Full check detected only	109

From these results, we estimate that the partial check has an error coverage of about 97.7%. While this is quite good, there are a substantial number of cases in which the partial check only detected these errors in a subsequent iteration. This results in a greater overhead for recovery, since the checkpoints would have been corrupted with errors by the time these errors are detected. If we exclude these cases, then the coverage of the partial check is about 77.4%. For further analysis, we define the *detection latency* as the number of iterations that elapse between the occurrence of errors and the subsequent detection of these errors. For example, if errors occur during iteration 5 and these errors are detected in iteration 6, then the detection latency is 1. Figure 3.4 shows a histogram of the detection latency of the partial check.

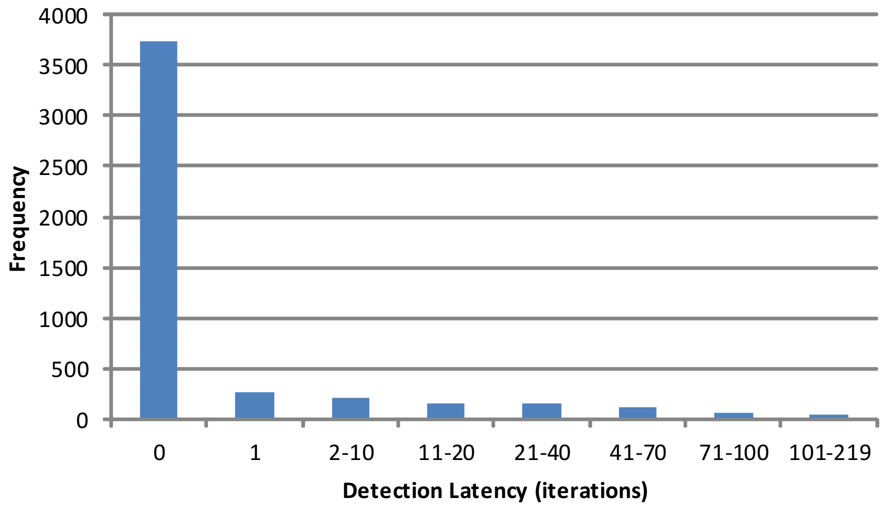


Figure 3.4 Detection latency for the partial check.

In most cases (3745), the detection latency is 0. This corresponds to the number of cases where the partial check successfully detected the errors in the same iteration in which they occurred. There are also a smaller number of cases with higher detection latencies, which total to 984 cases.

Figure 3.5 shows the timing results. The average timings for the upper two plots are normalized with respect to the average baseline runtime, while the lower two plots show the raw average runtime in seconds. The horizontal axis for all plots indicates the size (i.e. $n \times n$) of the input matrix A that was used. In the *fault not activated* cases, the Redundant scheme has a significantly greater overhead than the other three schemes, and the overhead increases as n increases. However, for smaller values of n, the normalized runtime of Redundant is less than twice that of the baseline experiments. This is likely because the highly parallel nature of the GPU is able to hide most of the overhead of the redundant computations. PC-C and PC-CS have a slightly higher overhead than FC-O, but this overhead does not scale appreciably as n increases. This indicates that the computational complexity of PC-C and PC-CS are comparable to FC-O.

FC-O has the best performance in terms of runtime when no errors are present. However, when errors are present, the overhead of FC-O increases significantly, such that its performance is even worse than the Redundant scheme. To make matters worse, this overhead increases as n increases. By contrast, the overhead of PC-C and PC-CS increase only by a much smaller degree, and the overhead for these two schemes do not scale appreciably as n increases. PC-CS always has a lower overhead than PC-C, indicating that explicit memory operations for taking checkpoints do indeed have a measurable impact on performance. The results of these experiments indicate that the PC-CS scheme possesses the best balance of performance under both error-free and error-prone environments. It always performs better than PC-C, whether the fault was activated or not activated, and provides good error coverage while having a relatively small performance overhead which remains small as n is increased. Even when faults are not activated, the performance of PC-CS is still close to that of FC-O.

Our results appear to indicate that as n is scaled to values that reflect the size of input matrices that will be encountered in petascale and exascale computing, the FC-O scheme rapidly becomes an unacceptable method for ensuring reliable block Householder QR factorization. The PC-C and PC-CS schemes on the other hand, scale reasonably well as n is increased. We believe that these two schemes – in particular the PC-CS scheme, are promising fault tolerance methods for block Householder QR factorization at the petascale and exascale range.

In the next chapter, we will discuss FT strategies for Cholesky factorization, particularly the supernodal left-looking version. From this point onwards, we will focus solely on sparse matrices, since sparse matrices are much more extensively used in real-life applications compared to dense matrices.

Algorithm 1 Compute block Householder QR of $A \in \mathbf{R}^{n\times n}$

```
1:  procedure BlockHouseholder($A$, $Q$, $R$)
2:      $Q \leftarrow I, R \leftarrow A$
3:      for $i = 0$ to $n/r - 1$ do
4:          $s = ir$
5:          for $j = 0$ to $r - 1$ do
6:              $u = s + j$
7:              $[v, \beta] =$ house$(R(u : n - 1, u))$
8:              $R(u : n-1, u : s+r-1) = R(u : n-1, u : s+r-1) - \beta vv^T R(u : n-1, u : s+r-1)$
9:              $V(0 : n - 1, j) = [zeros(u, 1); v]$
10:             $B(j) = \beta$
11:         end for
12:         $Y = V(0 : n - 1, 0)$
13:         $W = -B(0)V(0 : n - 1, 0)$
14:         for $j = 1$ to $r - 1$ do
15:             $v = V(0 : n - 1, j)$
16:             $z = -B(j)v - B(j)WY^T v$
17:             $Y = [Yv]$
18:             $W = [Wz]$
19:         end for
20:         $R(s : n-1, s+r : n-1) = R(s : n-1, s+r : n-1) + YW^T R(s : n-1, s+r : n-1)$
21:         $Q(0 : n - 1, s : n - 1) = Q(0 : n - 1, s : n - 1) + Q(0 : n - 1, s : n - 1)WY^T$
22:     end for
23:     return $Q$, $R$
24: end procedure
```

Algorithm 2 Compute Householder vector v and scalar β

```
1: procedure HOUSE(x)
2:     $n = \textbf{length}(x)$
3:     $\sigma = x(1:n-1)^T x(1:n-1)$
4:     $v = [1 \;\; x(1:n-1)^T]^T$
5:     if $\sigma = 0$ then
6:         $\beta = 2$
7:     else
8:         $\mu = \sqrt{x(0)^2 + \sigma}$
9:         if $x(0) \leq 0$ then
10:            $v(0) = x(0) - \mu$
11:        else
12:            $v(0) = \frac{-\sigma}{x(0)+\mu}$
13:        end if
14:        $\beta = \frac{2v(0)^2}{v(0)^2+\sigma}$
15:        $v = \frac{v}{v(0)}$
16:    end if
17:    return $v$, $\beta$
18: end procedure
```

Algorithm 3 FC-O scheme

```
1: procedure FC-O($A$, $Q$, $R$)
2:     do BlockHouseholder$(A, Q, R)$
3:     at end of last iteration, do FullCheck$(A, Q, R)$
4:     if errors detected then
5:         restart BlockHouseholder$(A, Q, R)$
6:     end if
7:     return $Q$, $R$
8: end procedure
```

Algorithm 4 PC-C scheme

```
 1: procedure PC-C($A$, $Q$, $R$)
 2:     do BlockHouseholder($A, Q, R$)
 3:     if this is end of last iteration then
 4:         do FullCheck($A, Q, R$)
 5:         if errors detected then
 6:             restart BlockHouseholder($A, Q, R$)
 7:         end if
 8:     else at end of each iteration
 9:         do PartialCheck($A, Q, R$)
10:         if errors detected and flag set then
11:             restart BlockHouseholder($A, Q, R$)
12:         else if errors detected and flag not set then
13:             $Q, R \leftarrow$ checkpoints, set flag, redo iteration
14:         else
15:             update checkpoints
16:         end if
17:     end if
18:     return $Q$, $R$
19: end procedure
```

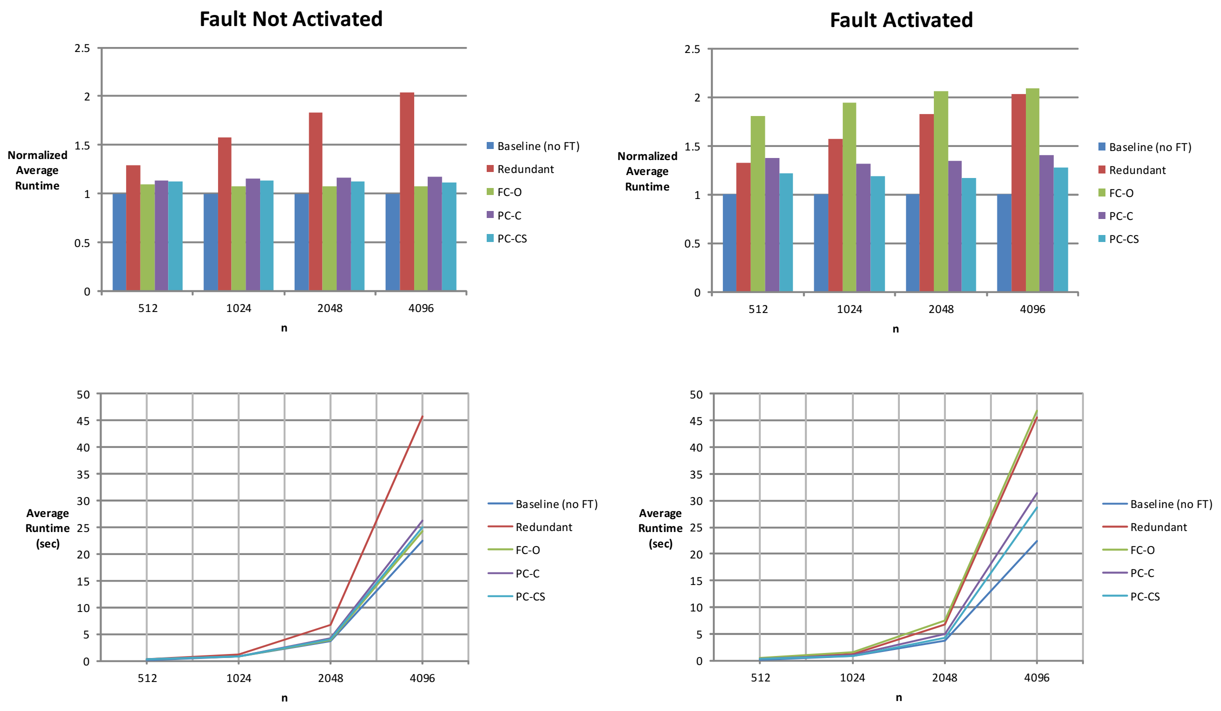


Figure 3.5 Experimental results for the FT techniques

Chapter 4

Fault Tolerance in Cholesky Factorization

4.1 Overview of Left-looking Cholesky

In this section, we first describe the basic algorithm of the Left-looking Cholesky factorization method, then explain its supernodal counterpart: the supernodal left-looking Cholesky factorization.

4.1.1 Left-looking Cholesky

The left-looking Cholesky factorization is a frequently used method of factorizing sparse symmetric positive definite matrices [56]. It is useful as the first step in solving a linear system of equations, in the form of $Ax = b$. This algorithm calculates the matrix L one column per iteration, where $A = LL^T$ and L is a lower triangular matrix. During the kth iteration, the kth column of L is calculated using the following formula, which utilizes (but does *not* modify) the previously computed elements in columns 1 to $k-1$ of L. $L_{k,k}$ refers to the element of L with row and column index k (i.e. the kth diagonal element of L), while $L_{k+1:n,k}$ refers to elements $k+1$ to n of the kth column of L. Elements 1 to $k-1$ of the kth column are zero and are not computed.

$$L_{k,k} = \sqrt{A_{k,k} - L_{k,1:k-1}L^T_{k,1:k-1}}$$

$$L_{k+1:n,k} = \frac{A_{k+1:n,k} - L_{k+1:n,1:k-1}L^T_{k,1:k-1}}{L_{k,k}}$$

When a sparse matrix A is used, only the columns of $L_{k:n,1:k-1}$ that have non-zero elements in the top row (i.e. $L_{k,1:k-1}$) are required in the sparse matrix-vector multiplication.

Algorithm 5 illustrates the sparse left-looking Cholesky factorization. The factorization proceeds in two distinct phases: *symbolic factorization* and *numerical factorization*. In symbolic factorization, the non-zero structure of L is determined. This phase is necessary because the data structures in sparse matrix factorization reserve space only for the non-zero elements. Note that L will typically have more non-zero elements than A, and this is known as *fill-in*. During numerical factorization, the non-zero elements of L are computed.

Algorithm 5 Compute sparse factorization $A \in \mathbf{R}^{n\times n} = LL^T$

```
1:  procedure LEFTCHOL($A, L$)
2:      do symbolic factorization to find non-zero struc. of L
3:      $c = [0\ 0\ \dots\ 0]^T$
4:      for $k = 1 : n$ do
5:          $c_{k:n} = A_{k:n,k} - L_{k:n,1:k-1}L^T_{k,1:k-1}$
6:          $L_{k,k} = \sqrt{c_k}$
7:          $L_{k+1:n,k} = \frac{c_{k+1:n}}{L_{k,k}}$
8:          $c = [0\ 0\ \dots\ 0]^T$
9:      end for
10:     return $L$
11: end procedure
```

4.1.2 Supernodal Left-looking Cholesky

The supernodal left-looking Cholesky method works similarly to the basic sparse left-looking Cholesky method, except that instead of computing one column of L per iteration, it computes multiple columns in one iteration. Such a group of columns is referred to as a supernode [57]. The columns in each supernode have a similar non-zero pattern and this allows the supernodal left-looking Cholesky method to maximize throughput by operating on dense blocks of sub-matrices. Figure 4.1 shows a sample L matrix divided into 5 supernodes.

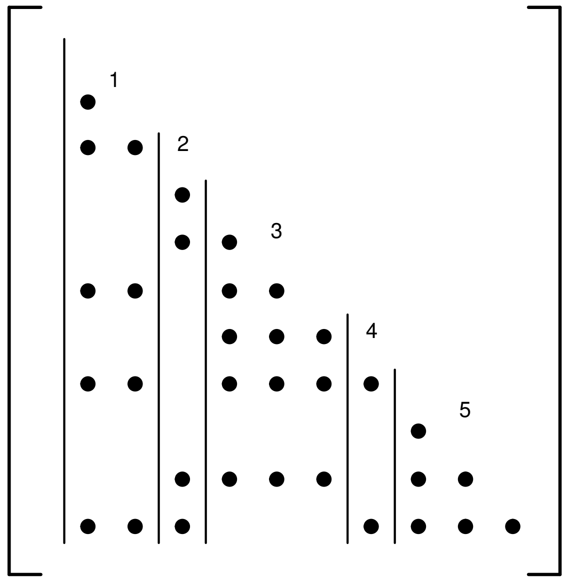


Figure 4.1 Supernodes for a sample L matrix.

During each iteration, in a process analogous to the computations performed by the basic sparse left-looking Cholesky factorization method, the supernodal left-looking Cholesky method first computes the left-looking update of all columns in the current supernode, similar to line 5 of algorithm 5. Then it factorizes the diagonal block of the current supernode using block Cholesky factorization, which is equivalent to line 6 of algorithm 5. Finally, similar to line 7 of algorithm 5, the supernodal left-looking Cholesky method scales the off-diagonal block of the current supernode, which involves solving a linear system of equations.

It is also worth noting that the supernodal left-looking Cholesky factorization modifies only the elements in the current supernode. Elements computed in previous supernodes are used in the update of these elements, while elements in successive supernodes are *not* modified in any way. We would like to direct the interested reader to Davis et al [56] for more details on the supernodal left-looking Cholesky factorization.

4.2 Necessity for Fault Tolerance in Left-looking Cholesky

In this section, we briefly describe CHOLMOD, a sparse direct solver that we use in our experiments for evaluating our FT schemes. We then illustrate that faults can cause serious errors in the factorization phase of such a solver, and motivate the need to protect solvers like CHOLMOD with FT schemes.

4.2.1 CHOLMOD

CHOLMOD [58], from SuiteSparse, is an application for sparse matrices that is able to perform two major kinds of factorizations, which are left-looking Cholesky and LDL factorization. It can also solve a specified linear system after factorization of the coefficient matrix is complete. It chooses which of these factorizations to use, depending on the characteristics of the input matrix. In most cases, if the input matrix A is symmetric positive definite, CHOLMOD will choose the left-looking Cholesky factorization. CHOLMOD can also select between the basic and supernodal versions of Cholesky factorization – this usually depends on the size and non-zero pattern of A.

The user is also able to force CHOLMOD to use a particular factorization method. CHOLMOD uses the compressed column storage format (CCS) for its sparse matrices. CHOLMOD is able to take advantage of the resources of a GPU when supernodal left-looking Cholesky factorization is specified. Note that the GPU is only utilized in the numerical factorization phase. CHOLMOD uses the CCS format to represent sparse matrices.

In our work, we use CHOLMOD to evaluate our proposed FT schemes, and provide fault tolerance for the operations of CHOLMOD that are performed on the GPU.

4.2.2 Effect of Faults on Direct Cholesky Solvers

We ran fault injection experiments to illustrate that it is important to provide fault tolerance for direct solvers that use the left-looking Cholesky method, like CHOLMOD. In these experiments, we ran the basic CHOLMOD program with no fault tolerance, using the supernodal left-looking Cholesky factorization with the nd6k matrix from the University of Florida sparse matrix collection [59]. Faults are injected into the GPU by flipping a chosen bit position using my fault injection mechanism. We determine whether the final factorization has errors by computing and comparing the L2-norm of Ax and of LL^Tx, where $x = [1 \;\; 1 \; ... \; 1]^T$.

As seen in Figure 4.2, errors that involve the more significant bits in the result of a thread lead to a vast majority of cases where there are errors in the final factorization of the input matrix. The situation is almost as bad for bit flips occurring in the less significant bit positions: even in this case, there are a significant number of incidents with errors in the final factorization. In other words, when a bit flip error affects some computation in the GPU, the matrix factorization can be expected to contain significant errors. Even worse, these errors, which result in SDC, are not apparent to

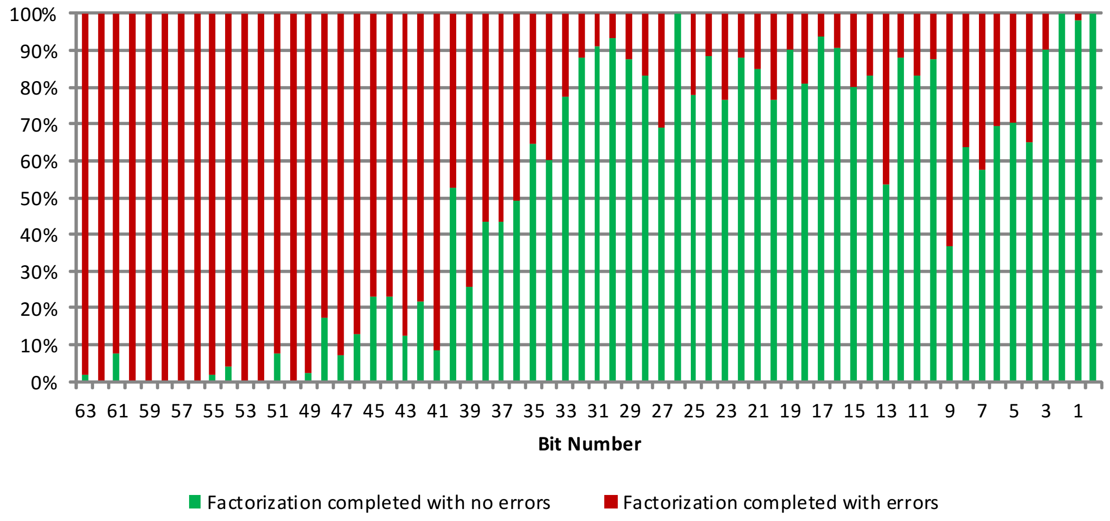


Figure 4.2 CHOLMOD factorization outcomes for various bit positions.

the user because the system typically exhibits no trace of abnormal behavior, unlike errors that cause the system to crash. This shows that fault tolerance is necessary to protect direct solvers like CHOLMOD from faults and errors that can affect the reliability of scientific and mathematical applications.

4.3 Fault Tolerance Methods for Left-looking Cholesky

4.3.1 Invariant properties for Left-looking Cholesky

All Cholesky factorization methods involve factorizing a matrix A such that $A = LL^T$, where L is a lower triangular matrix. If A is positive definite, then the factorization is guaranteed to be unique. This can be used as an invariant property at a high level; however, the invariant does *not* hold in between iterations, i.e. $A \neq L_k L_k^T$, where L_k is the currently computed version of L at the end of the kth iteration. Fortunately, a sub-portion of the matrix L_k can still be used to check against A. In the case of left-looking Cholesky, since only one column of L is modified and finalized per iteration (i.e. that column will not be further modified in future iterations), only that column needs to be checked for errors. This principle extends to the supernodal left-looking Cholesky factorization, where only one supernode is modified and finalized after each iteration.

4.3.2 Fault Detection Methods

In this subsection, we describe several FT methods for the left-looking Cholesky factorization.

4.3.2.1 xAy vs xLL^Ty Checking

An invariant property holds for the currently computed supernode in any one iteration. We exploit this property and use it as a means for error checking. Let x be a dense row vector and y a dense column vector, each of size n, where n is the width of A. Then $xAy = xLL^Ty$ and by carefully choosing the elements of x and y, one can ensure that only the columns of L that have been computed up to the current iteration are involved in calculating xLL^Ty. We choose the elements of the vectors as follows.

Suppose at the end of the kth iteration, one has computed the columns c to $c + b$ of L. Note that columns 1 to $c - 1$ of L were computed in previous supernodal iterations. Let $x_i = 1$ if $c \leq i \leq c + b$, and 0 otherwise. Let $y_i = 1$ for all i.

Let $u = xL$. Since $L_{i,j} = 0$ for all $i < j$,

$$u_i = \sum_{i=j}^{n} x_i L_{i,j}.$$

Since $x_j = 0$ for $j > c + b$, it follows that $u_j = 0$ for all $j > c + b$. Thus, the calculation of u only involves columns 1 to $c + b$ of L. Now let $v = L^Ty$. This means that $uv = xLL^Ty$. Because

$u_j = 0$ for all $j > c + b$, one does not require the values of v_j for all $j > c + b$ in order to calculate xLL^Ty. In addition, the elements v_1 to v_{c-1} only involve the rows of L^T that were computed in previous supernodal iterations. Therefore, it is unnecessary to recalculate these elements; one only needs to compute elements v_c to v_{c+b} and include them with the previously computed values of v.

With this choice of x and y, the elements u_1 to u_{c+b} can be computed in parallel on the GPU using columns c to $c + b$ of L, while the elements v_c to v_{c+b} can likewise be computed in parallel using the corresponding columns of L (i.e. the corresponding rows of L^T).

The calculation of xAy is straightforward. Ay needs to be computed only once, before the factorization starts. Then in the current supernodal iteration, the computation of xAy only involves the cth to $(c + b)$th elements of Ay. This method allows one to check the elements of the recently computed supernode with minimal overhead.

The error coverage of this method is expected to be close to 100%, because all diagonal elements of L are guaranteed to be non-zero (and positive) and therefore all columns of the currently computed supernode are accounted for in the product xLL^Ty. This means that any error affecting the elements in any of the columns of the current supernode should result in a discrepancy between the values of xAy and xLL^Ty.

An error is detected if $|xLL^Ty - xAy|/|xAy| > 10^{-6}$ for $|xAy| \geq 1$, or if $|xLL^Ty - xAy| > 10^{-6}$ for $|xAy| < 1$.

4.3.2.2 Simplified xAy vs xLL^Ty Checking

This method is very similar to the xAy vs xLL^Ty checking method described in the previous subsection; however, in this FT method we set x such that $x_i = 1$ only for $i = c + b$ (0 for all other elements of x). This sacrifices error coverage for lower overhead in the checking kernels, because only the $(c+b)$th row of L is used in the computation of xL – unlike the xAy vs xLL^Ty checking method, the product xLL^Ty does *not* account for all columns in the current supernode. In other words, xL is the $(c+b)$th row of L, which is sparse. Consequently, it cannot be guaranteed that elements c to $c+b-1$ of xL are non-zero, and any error that affects the elements in columns c to $c+b-1$ of the supernode is likely to go undetected.

Like the previous FT scheme, an error is detected if $|xLL^Ty - xAy|/|xAy| > 10^{-6}$ for $|xAy| \geq 1$, or if $|xLL^Ty - xAy| > 10^{-6}$ for $|xAy| < 1$.

4.3.2.3 Column Checksum

Wu et al [29] propose using column checksums as a means of checking for errors. They observe that the checksum invariant property holds for outer-product Cholesky. The checksum invariant property is also true for left-looking Cholesky. Note that it is not possible to simply modify A to include an extra row and column (for the checksums) and run this modified A directly on CHOLMOD, because A is no longer positive definite.

We implemented a checksum scheme similar to the one proposed by Chen et al [26], using only one column checksum each for A and L. We compared this scheme against my proposed FT schemes. Like Chen's proposed scheme, the column checksums are implemented as separate data structures (row vectors) and are computed separately from the main Cholesky factorization process. Let $A_{n+1,1:n}$ and $L_{n+1,1:n}$ be the column checksums for A and L respectively. The kth element of $L_{n+1,1:n}$ can be computed using the following formula. $L_{n+1,k}$ refers to the kth element of the column checksum vector for L, where $1 \leq k \leq n$.

$$L_{n+1,k} = \frac{A_{n+1,k} - L_{n+1,1:k-1}L^T_{k,1:k-1}}{L_{k,k}}$$

The column checksums of A can be computed in a similar fashion and this can be done before factorization begins. During each supernodal iteration, we compute the column checksums for columns c to $c + b$ of L, where columns c to $c + b$ are the columns in the currently computed supernode. Note that the kth checksum requires all previously computed checksum elements, i.e. $L_{n+1,1:k-1}$. This means that the column checksums have to be computed sequentially, and this results in significant overhead. To check for errors, we sum up the elements for each column c to $c+b$ of the current supernode of L (this can be done in parallel) and compare the sum of a column, $ColSum$, against its respective column checksum, $ColChecksum$.

Like the xAy vs xLL^Ty checking method, the error coverage of this FT method is close to 100%. In this scheme, an error is detected if $|ColChecksum - ColSum|/|ColChecksum| > 10^{-6}$ for $|ColChecksum| \geq 1$, or if $|ColChecksum - ColSum| > 10^{-6}$ for $|ColChecksum| < 1$.

4.3.3 Fault Recovery Method

In all three schemes, when an error is detected, we restart the numerical factorization process. This allows the memory overhead of checking to be small, as only the initial version of L needs to be stored. The overhead of recovery can be high. However, the occurrence of an error is generally rare, and it is expected that recovery will not be necessary most of the time.

4.4 Experimental Methodology

We run all experiments using a GPU compute cluster. It contains 14 compute nodes, 10 of which contain GTX 480 GPUs. Each experiment is run on a single node using just one of the four GTX 480 cards. We used double-precision floating point numbers. Table 7.1 shows the specifications of one such node.

Table 4.1 Specifications of a compute node

Component	Quantity
Intel Xeon E5520 CPU @ 2.26GHz	2
48GB DDR3 RAM	-
NVIDIA GTX 480 w/ 1.5GB VRAM	4
Western Digital RE4 2TB HDD	1

Each experiment runs an instance of CHOLMOD that is set to the supernodal left-looking Cholesky factorization method. We perform experiments in two main categories: performance of FT schemes without fault injection and performance of FT schemes with fault injection. We choose 5 sparse symmetric positive definite real matrices from the University of Florida sparse matrix collection [59] that represent a variety of different applications, as shown in Table 4.2.

For those experiments that perform fault injection, we inject faults into the kernels that perform the basic linear algebra operations in the numerical factorization phase of CHOLMOD. We set the active duration of a fault to $1000\mu s$ and the fault rate according to the input matrix as shown in Table 4.2. We measure the average time needed for factorization of the input matrix. We compare

Table 4.2 Input matrices used in experiments

Matrix	Width	Non-zeroes	Application	Fault Rate (s)
nasa4704	4704	104756	structural	0.2
aft01	8205	125567	acoustics	0.3
fv2	9801	87025	2D/3D	0.5
bloweybq	10001	49999	materials	0.2
bcsstk17	10974	428650	structural	1

the factorization time for any run with the corresponding factorization time for CHOLMOD with no FT. In each fault injection experiment, we pick one random SP of a randomly chosen SM and randomly choose a bit to flip, uniformly distributed from bit number 63 to 56, in the event a fault is activated. For evaluation purposes, we choose b in the CHOLMOD solver such that $x = [1 \;\; 1 \; ... \; 1]^T$ in the linear system $Ax = b$ that is to be solved. We treat any run in which $||LL^T x - b||_2/||b||_2 > 10^{-6}$ as erroneous.

4.5 Results

4.5.1 Performance of FT Schemes in the Absence of Faults

Figure 4.3 compares the various runtimes of each FT scheme with respect to the runtime of the CHOLMOD implementation with no FT, in the absence of faults. As shown in figure 4.3, the xAy vs xLL^Ty checking scheme and the simplified xAy vs xLL^Ty checking scheme have significantly lower overhead than the column checksum scheme. For each matrix, the runtime of the simplified xAy vs xLL^Ty checking scheme is always less than the runtime of the xAy vs xLL^Ty checking

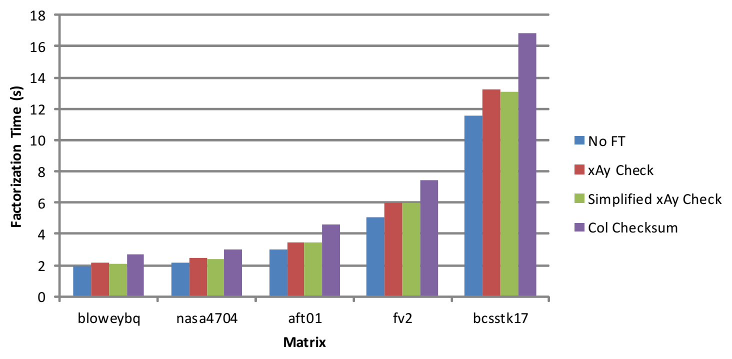


Figure 4.3 Performance of various FT schemes with no fault injection

scheme (but by only a small margin), while the column checksum scheme always has the largest overhead.

It is also interesting to note that the number of non-zero elements in the input matrix does not correlate with the runtime for factorization. For example, nasa4704 and aft01 both have more non-zero elements than fv2, but fv2 requires a longer factorization time than either nasa4704 or aft01. It is possible that factorization performance is strongly dependent on the non-zero pattern of the input matrix.

These results show that our proposed schemes for the supernodal left-looking Cholesky factorization can offer good performance, compared to traditional checksum schemes.

4.5.2 Performance of FT Schemes with Fault Injection

Figure 4.4(a) shows the error coverage of the FT schemes when faults are injected, while figure 4.4(b) shows the runtime of these schemes, compared to the runtime of the base CHOLMOD with no fault tolerance (leftmost column), only for cases where the errors were detected and corrected.

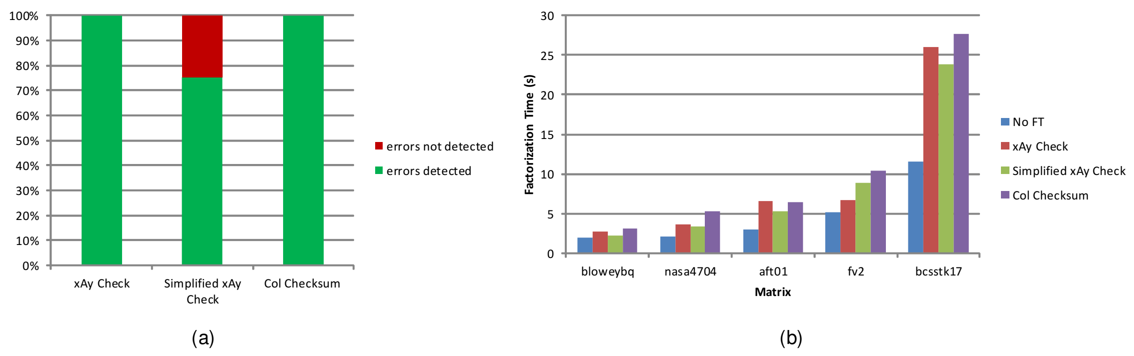


Figure 4.4 Performance of various FT schemes with fault injection

From figure 4.4(a), one can see that both the xAy vs xLL^Ty checking scheme and the column checksum scheme have excellent error detection coverage, with practically all errors getting detected. However, the simplified xAy vs xLL^Ty checking scheme suffers from poorer coverage (about 75%) due to the fact that it does not check all elements that are computed during any supernodal iteration.

As can be seen in figure 4.4(b), the overhead of all three FT schemes go up very steeply when recovery was required. This is mainly due to the fact that we choose to reset back to the initial version of L when an error is detected. The overhead also varies considerably between the various FT schemes, because the overhead depends strongly on which iteration the errors occurred: an error occurring in an earlier iteration will incur less overhead than one occurring during a later iteration. The overhead of the checks themselves, however, is small.

In the following chapter, we will discuss our FT strategies for the LU factorization method. The LU factorization method is more challenging to protect against errors, compared to Cholesky factorization. This is because row pivoting is necessary to stabilize the algorithm, and this can interfere with FT methods.

Chapter 5

Fault Tolerance in LU Factorization

5.1 Overview of LU Factorization

LU factorization is a popular method for factorizing a general matrix, particularly those that are not symmetric. It is typically used as a first step in solving a linear system of equations, as it is often easier to solve such a system of equations if the coefficient matrices are triangular. Other applications of LU factorization include computing the determinant of a matrix and inversion of a matrix. For this work, we implemented our own GPU-based sparse right-looking LU factorization application with dynamic partial pivoting, using the CCS sparse matrix format.

5.1.1 Right-looking LU Factorization

The right-looking LU factorization, also known as Gaussian Elimination, is a well-known method of factorizing general matrices [52]. This algorithm finalizes (modifies and commits) one column and one row of the matrices L and U per iteration, where $A = LU$ and L is a unit diagonal lower triangular matrix and U is an upper triangular matrix. Note that the $n \times n$ input matrix A is typically factorized *in-place*; in other words, the elements L and U overwrite those of A (the unit diagonal of L is implicit and is not present in the final output), such that at the end of factorization, the matrix $L + U - I$ replaces A; this is the case in Algorithm 6 below. During the kth iteration, the kth column of L and U (combined) is calculated, then the remaining $k + 1$ to n columns are updated using the outer product of two vectors, a subset of the kth column and a subset of the kth row. Note that this implies that the elements in the $k + 1$ to n columns are modified, but not committed, as they will be modified in future iterations.

LU factorization is not inherently stable, unlike Cholesky factorization. Partial pivoting (row interchanges) is often necessary to ensure a stable LU factorization, i.e. $PA = LU$ where P is a row permutation matrix. Since P is basically the identity matrix I with its non-zero elements permuted, it suffices to use a $1 \times n$ vector p, where the kth element p_k indicates that row k has been swapped with row p_k. One way to choose a pivot during iteration k is to select the element in column k (at and below the diagonal) with the largest magnitude as the pivot.

When the input matrix is in a sparse matrix format, such as the CCS format, it is necessary to first perform symbolic analysis and symbolic factorization on the input matrix prior to numerical factorization. This is required because the number of non-zero values in the factor matrices are usually significantly greater than those of the original input matrix – as mentioned in Chapter 4, this is referred to as *fill-in*. The symbolic analysis and factorization phase typically includes column permutation to minimize fill-in, as well as the determination of the actual locations of the non-zero values of the factor matrices. Here, we use the efficient symbolic factorization algorithm developed by George and Ng [60], and use the AMD algorithm [61] introduced by Amestoy et al, to minimize fill-in.

Algorithm 6 illustrates the sparse right-looking LU factorization with dynamic partial pivoting. The factorization proceeds in two distinct phases: *symbolic factorization* and *numerical factorization*. In symbolic factorization, the non-zero structure of $L + U - I$ is determined. This phase is necessary because the data structures in sparse matrix factorization reserve space only for the non-zero elements. Note that $L + U - I$ will typically have more non-zero elements than A, and this is known as *fill-in*. During numerical factorization, only the non-zero elements of $L + U - I$ are computed and it is carried out in two distinct steps: column update and trailing matrix update.

Algorithm 6 Compute sparse right-looking factorization $PA \in \mathbf{R}^{n\times n} = LU$

```
 1: procedure RIGHTLU($A$)
 2:     do symbolic factorization to find non-zero struc. of L+U-I
 3:     L+U-I replaces A at end of factorization
 4:     for $k = 1 : n-1$ do
 5:         determine $\mu$ with $k \le \mu \le n$ such that $A_{\mu,k}$ is
 6:         the max abs element
 7:         swap $A_{\mu,k:n}$ with $A_{k,k:n}$
 8:         $p_k = \mu$
 9:         if $A_{k,k} \neq 0$ then
10:             $A_{k+1:n,k} = \frac{A_{k+1:n,k}}{A_{k,k}}$ (column update)
11:             $A_{k+1:n,k+1:n} = A_{k+1:n,k+1:n}$
12:             $-A_{k+1:n,k}A_{k,k+1:n}$ (trailing matrix update)
13:         end if
14:     end for
15:     return $A, p$
16: end procedure
```

5.1.2 Left-looking LU Factorization

The left-looking LU factorization is similar to the right-looking LU version. However, unlike right-looking LU factorization, the left-looking version modifies and commits only one column of L and U each iteration (L is a unit diagonal lower triangular matrix). It also does not make any modifications to elements beyond the column that is being calculated in the current iteration. Algorithm 7 shows the sparse left-looking LU factorization algorithm. During the kth iteration, a lower triangular solve is carried out to determine the column vector $U_{1:k-1,k}$. The scalar $U_{k,k}$ is also computed – this involves a dot product. Pivoting of the current column is performed before calculating the vector $L_{k+1:n,k}$, which involves a matrix-vector product.

Figure 5.1 summarizes how the right-looking and left-looking LU factorization methods modify and commit elements of a sample matrix during one particular iteration. The dots indicate elements that are determined to be non-zero during symbolic factorization. The blue region indicates the set of elements that were modified and committed (i.e. will not be further modified) in previous iterations, while the green region highlights the set of elements that are modified and committed in the current iteration. The red region, which is applicable only for the right-looking LU factorization, indicates elements that are modified during the current iteration, but will be further modified in future iterations.

5.1.3 Performance of Right-looking LU vs Left-looking LU

In this sub-section, we discuss how the computational complexity of the two algorithms change when they are executed on different platforms (CPUs vs. GPUs). To simplify the analysis, we assume that the input matrix is dense, and focus mainly on scalar multiplication operations. We also exclude the effect of row pivoting operations.

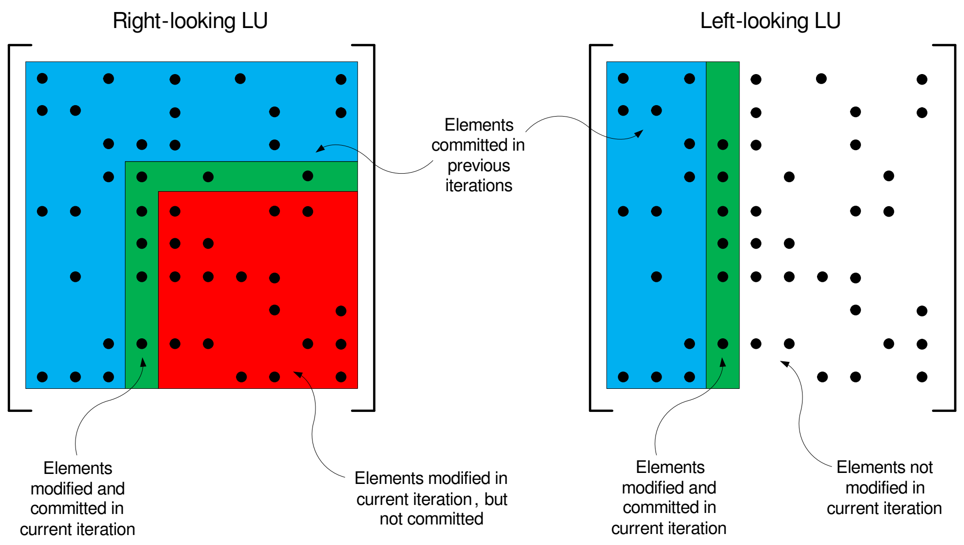

Right-looking LU
Left-looking LU
Elements committed in previous iterations
Elements modified and committed in current iteration
Elements modified in current iteration, but not committed
Elements modified and committed in current iteration
Elements not modified in current iteration

Figure 5.1 Elements modified in right-looking LU vs left-looking LU

5.1.3.1 Computational Complexity of Right-looking LU vs. Left-looking LU on CPUs

As can be seen in algorithm 6, each iteration, there is a column update (this involves $n-k$ divisions, which we do not count), as well as an outer product (trailing matrix update) which require $(n-k)^2$ multiplication operations. The total number of multiplication operations required by the right-looking LU factorization, over the $n-1$ iterations, is:

$$\sum_{k=1}^{n-1}(n-k)^2 = \frac{1}{6}n(2n^2-3n+1).$$

Thus, the total number of operations approaches $O(n^3)$ and when these operations are executed sequentially on a CPU, the right-looking LU factorization algorithm has a computational complexity of $O(n^3)$.

As shown in algorithm 7, each iteration, not counting the row pivoting operations, there is a lower triangular solve for the column vector $U_{1:k-1,k}$, an inner/dot product involved to find the scalar $U_{k,k}$, and finally a matrix-vector product that is used to determine the vector $L_{k+1:n,k}$.

During the kth iteration, the lower triangular solve step requires $\frac{1}{2}k(k-1)$ scalar multiplication operations, the dot product requires $k-1$ multiplications and the matrix-vector product requires $(n-k)(k-1)$ multiplications. Over the n iterations involved, the total number of multiplication operations executed by the left-looking LU factorization is:

$$\sum_{k=1}^{n}(n-\frac{1}{2}k+1)(k-1)=\frac{1}{3}n(n^2-1).$$

Again, the overall number of scalar multiplication operations approaches $O(n^3)$. Thus, like its right-looking counterpart, the left-looking LU factorization algorithm has a computational complexity of $O(n^3)$ when executed on a sequential processor. It is interesting to note that in general, the left-looking LU factorization requires a greater number of multiplication operations than the right-looking LU. This indicates that for large values of n, the right-looking LU factorization method is likely to run faster than the left-looking version on a CPU.

5.1.3.2 Computational Complexity of Right-looking LU vs. Left-looking LU on GPUs

The computational complexity of the two algorithms differ significantly when they are executed on a parallel processor, such as a GPU. Assuming that the GPU has infinite parallelism, then in each iteration, the outer product phase of the right-looking LU factorization can theoretically be done in $O(1)$ time on the GPU. This is because each of the scalar multiplications of the outer product has no data dependency on any other scalar multiplication, and so the multiplication operations involved in the outer product can be done completely in parallel. This implies that in each iteration, effectively only one multiplication operation is performed (in terms of time complexity) and consequently the total effective number of multiplication operations executed by the right-looking LU factorization on a GPU is:

$$\sum_{k=1}^{n-1} 1 = n - 1.$$

This means that overall, the right-looking algorithm has a theoretical complexity of $O(n)$ on a GPU.

In the case of the left-looking LU factorization method, the matrix-vector product in any iteration cannot be done completely in parallel, because even though the scalar elements of the result vector can be computed in parallel (the elements have no data dependencies on each other), each scalar element itself requires a dot product, which has some overhead. A dot product has $O(k)$ complexity if it is computed in the typical serial fashion, but can be made to be more efficient by using parallel reduction on a GPU. This reduces the computational complexity of a dot product operation and the matrix-vector product to $O(log_2\, k)$ during the kth iteration.

Meanwhile, the lower triangular solve step of each iteration has high overhead, even when accounting for the fact that each of the elements of the result vector $U_{1:k-1,k}$ can be computed somewhat in parallel. Unlike the matrix-vector product, however, these elements do have data dependencies on each other, which limits the amount of parallelism – in particular, the scalar elements *cannot* be computed using parallel reduction. In addition, some synchronization mechanisms are

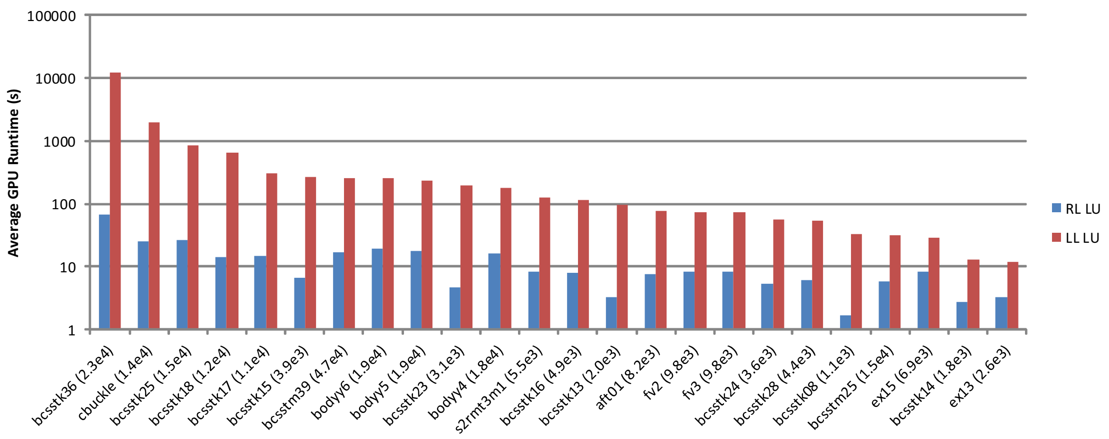


Figure 5.2 Comparison of right-looking LU vs left-looking LU runtime

necessary to maintain coherence. The result is that the lower triangular solve step has $O(k)$ computational complexity in the kth iteration. Consequently, the total effective number of multiplication operations executed by the left-looking LU factorization on a GPU is:

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n-1).$$

This leads to the left-looking LU factorization algorithm having an overall computational complexity of $O(n^2)$ on a GPU, which is significantly worse than that of right-looking LU factorization.

5.1.3.3 Practical Observations on GPUs

We conducted some experiments to compare the runtime of the right-looking LU factorization method and the left-looking method on a GPU platform. Note that our analysis in the previous sub-section is based on dense matrices, while our experiments are conducted using sparse matrices in the CCS format. GPU memory overhead may also account for some of the observed difference in performance between the right-looking and left-looking LU factorization methods.

Figure 5.2 compares the GPU runtime of the right-looking LU factorization method and its left-looking counterpart for a selection of sparse square matrices, with the width of the matrix (n) shown in parenthesis. Note that the vertical axis is in log-scale. We used double-precision floating point numbers. As can be observed in this figure, the runtime of the left-looking LU factorization method is significantly higher than the right-looking LU method, in some cases by as much as two orders of magnitude. The results show that the right-looking LU factorization method does indeed run faster than the left-looking LU method on a GPU platform.

5.1.4 Performance of GPUs on Sparse Matrices

As we mentioned in Chapter 2, there are limitations on the performance of GPUs when the input matrix is in a sparse matrix format, such as the CCS format. In light of these issues, one might wonder whether there is any point in using a GPU implementation for sparse LU factorization, as opposed to simply using a CPU implementation. To address this concern, we ran some experiments to compare the numerical factorization time of our GPU-based sparse right-looking LU factorization implementation to the numerical factorization time of a CPU implementation that uses the sparse left-looking LU factorization method. Our CPU implementation is based on CSparse, an efficient CPU sparse matrix factorization package by Davis [62]. Both implementations use dynamic partial pivoting, double-precision floating point numbers and the CCS sparse matrix format. The system used in the CPU implementation of these experiments is an Intel Core i7 860 CPU (2.80GHz, quad-core) with 8GB RAM. Our GPU implementation uses the NVIDIA GTX 1080 GPU.

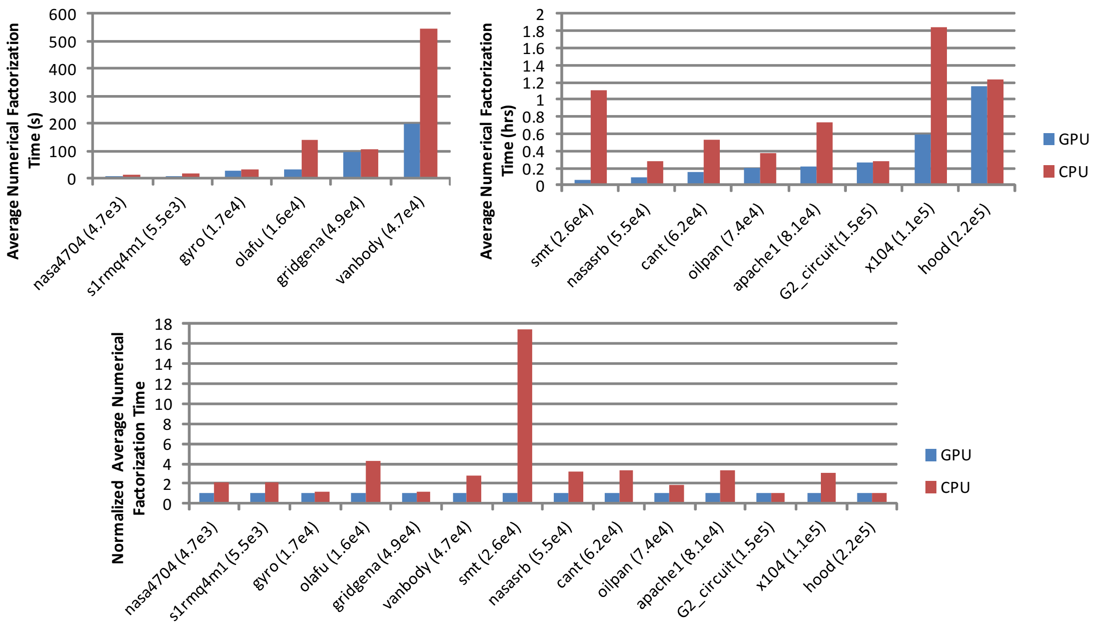


Figure 5.3 Comparison of numerical factorization time for GPU and CPU

Figure 5.3 shows the numerical factorization time of our GPU implementation, compared to the CPU implementation, for a selection of sparse matrices. The upper plots measure the average numerical factorization time for both the GPU and CPU implementations, while the lower plot shows the normalized numerical factorization time (normalized to the average numerical factorization time for the GPU). As can be seen from figure 5.3, for most of the matrices, our GPU implementation outperforms its CPU counterpart by at least a factor of 2, and in some cases by a larger factor. On average, the performance of the GPU implementation is about 3.4 times as fast as the CPU implementation. Based on these results, the GPU implementation has a clear performance advantage over its CPU counterpart. Thus, there is a strong benefit in using a GPU implementation for sparse right-looking LU factorization.

For the remainder of this chapter, we will utilize the right-looking LU factorization algorithm as the base matrix factorization application for our experiments.

5.2 Fault Tolerance Methods

5.2.1 Fault Model

We only consider transient hardware faults that persist for a certain duration, which affect the processing elements of the GPU and lead to SDC. We only consider faults that occur in the SP cores during execution, but allow for the possibility that corrupted results can be written to memory by threads running on faulty cores. Our fault model is similar to those used by other researchers [45, 11, 13]. A SP core of one particular SM is selected for fault injection. Each iteration, there is some probability that a fault will be injected for that iteration, based on a predetermined threshold. Threads performing the basic linear algebra operations in the numerical factorization phase of our application (excluding those that are involved in partial pivoting), that happen to be running on the chosen SP when the fault is injected for that iteration, will have their results corrupted. This is done by flipping a bit in the 64-bit double-precision floating point result of each of those threads. This simulates a fault with an active duration from around $500\mu s$ to a few milliseconds. We inject at most one fault for each experimental run.

5.2.2 Traditional Fault Detection and Recovery Methods

Before discussing our checking and recovery methods, we will first describe traditional checking and recovery methods for the right-looking LU factorization.

5.2.2.1 Duplicate Checking

The duplicate checking method is based on the well-established dual modular redundancy (DMR) strategy of fault tolerance. A major benefit of this checking scheme is that it is universally applicable to any application; it does not rely on the underlying algorithm of the application. In this case, a separate set of kernels are used to compute an extra copy of the unified $L + U - I$ matrix.

All elements of the redundant copy of the $L + U - I$ matrix can then be individually compared against their counterparts in the original $L + U - I$ matrix. In our implementation, the checking of elements is done in parallel on the GPU, and the check is done at the end of each iteration. This checking method also has 100% error coverage; however, it is very expensive and will incur high overhead. On the other hand, the duplicate check is able to detect any errors that occur in the current column, as well as in the trailing matrix, in the same iteration that those errors occur. We chose the threshold so that an error is detected if $|r_{dup} - r_{orig}| > 10^{-5}$, where r_{dup} is a scalar element of the redundant copy of $L + U - I$, and r_{orig} is the corresponding element of the original $L + U - I$ matrix.

It is possible to reduce the overall overhead of this checking scheme by decreasing the frequency of the checks. Another parameter that can be modified to reduce the overhead of checking is number of elements that are checked – limiting the number of elements in the trailing matrix portion that are checked should decrease the overhead. However, it is difficult to select the values of the parameters that will lead to an ideal implementation. For example, checking the results only once every (say) 64 iterations (instead of every iteration) will significantly reduce the error checking overhead, but this will lead to a much greater overhead of error recovery in the event that an error is detected. If the chosen method of recovery is via checkpointing and recomputation (see subsection 5.2.2.3), then in this example 64 iterations will have to be recomputed, which is a significant overhead and is not ideal. Limiting the number of elements that are checked each iteration reduces the error coverage of the check (on a per-iteration basis) which can lead to corrupted checkpoints, making it much more difficult to recover.

In this work, we implement the duplicate checking method such that the check is done at the end of every iteration, and recovery only involves the recomputation of at most one iteration.

We will also consider an *ideal* implementation of this check, in which we ignore the overhead of checking, but still consider the overhead of computing the redundant copy. This is discussed in more detail in Section 5.4.1.

5.2.2.2 Checksum Method

Wu et al [63], [29] proposed using row and column checksums to protect the factor matrices L and U, as well as the trailing matrix portion. While this method was originally evaluated on a distributed system consisting of many CPU nodes, we have made a best-effort implementation of their FT scheme and adapted it for our GPU-based sparse right-looking LU factorization program.

Each iteration, the column sum and row sum are computed for the column k and row k respectively, as well as for all rows and columns of the trailing matrix. These are then compared against their respective checksum elements to check for errors. It is important to note that the checksum invariant remains consistent for all row and column checksums for the trailing matrix, but remains consistent only for the column checksums of L and for the row checksums of U. In addition, since the diagonal elements of L are implicit, these must be accounted for when comparing the column sum and the column checksum of the current column of L (column k).

We chose the threshold such that that an error is detected if $|ColChecksum - ColSum| > 10^{-5}$, or if $|RowChecksum - RowSum| > 10^{-5}$.

5.2.2.3 Recovery via Checkpointing and Recomputation

In the case where the standard method of duplicate checking is used, the traditional method of recovery can be implemented: simply take a checkpoint for $L + U - I$ at the beginning of each iteration, before actual numerical computation is carried out. In the event that an error is detected in some iteration, recovery can be effected by first using the checkpoint (which is guaranteed to be uncorrupted) to "recompute" the numerical factorization for that particular iteration, then comparing the resulting matrix from this recomputation with *both* the original and duplicate matrices. Since one effectively has three result matrices to compare, one can determine which of the elements in any matrix are actually affected by the error. As such, the erroneous element(s) can be corrected and numerical factorization can proceed as normal afterwards.

The overhead of explicitly taking checkpoints can be significant, so one can hide the overhead by writing to separate, alternate memory locations for the unified $L + U - I$ matrix in between iterations. During even-numbered iterations, the kernels responsible for numeric factorization write to the alternate memory location, and during odd-numbered iterations, the kernels write to the original memory location. Generally speaking, if an error is detected during some iteration, the memory location that was previously written is used as the checkpoint to restore the correct, uncorrupted elements. This comes at a cost of a slightly more complicated recovery procedure, since the correct, finalized values of elements will actually be located in either one of the two memory locations.

5.2.2.4 Recovery via Re-execution of Current Iteration

This recovery method, proposed by Wu et al [63], is relevant when the aforementioned checksum method is used as the detection strategy. Here, when errors are detected in some iteration, that iteration is re-executed *before* the trailing matrix update is performed. After that, the trailing matrix update is allowed to proceed, and any subsequent errors detected in the trailing matrix are then corrected using the row and column checksum elements of the trailing matrix (without requiring the re-execution of the trailing matrix update step).

In the interest of implementing this recovery method as closely as possible to that described by Wu et al, we restrict our fault model to allow only at most one erroneous element in any one row or column of the trailing matrix, for just this recovery method alone. This allows for the row and column checksums of the trailing matrix to detect and correct such errors without requiring the re-execution of the entire trailing matrix update step of the iteration in which those errors occurred. Algorithm 8 illustrates this recovery procedure, used in conjunction with the checksum method.

5.2.3 Invariant Properties

All LU factorization methods involve factorizing a matrix A such that $A = LU$, ignoring the effects of partial pivoting. This can be used as an invariant property at a high level; however, the invariant does *not* hold in between iterations, i.e. $A \neq L_k U_k$, where L_k and U_k are the currently computed versions of L and U respectively at the end of the kth iteration. Fortunately, a subportion of the matrices L_k and U_k can still be used to check against A. In the case of right-looking LU, since the kth column of L and U is modified and finalized during iteration k (i.e. column k will not be further modified in future iterations), only the elements in column k should be checked for errors. Note that all columns after column k are also modified during the trailing matrix update, but they are not finalized. Hence, we do not check these columns until later iterations.

5.2.4 Proposed Fault Detection Method

We are now in a position to describe our checking method for the right-looking LU factorization. We will discuss the recovery methods in the next subsection.

5.2.4.1 LU Invariant Checking (LUIC)

An invariant property holds for the currently computed column of L and U in any one iteration. We exploit this property and use it as a means for error checking. Let x be a dense row vector and y a dense column vector, each of size n, where n is the width of A. Then $xAy = xLUy$ and by carefully choosing the elements of x and y, one can ensure that only the columns of L that have been computed up to the current iteration are involved in calculating $xLUy$. We choose the elements of the vectors as follows, such that the LUIC method is as efficient as possible: we set $x_i = 1$ for all i, and $y_i = 1$ for $i = k$, otherwise, $y_i = 0$. We ignore the effects of partial pivoting as it has no effect on the value of xAy and $xLUy$. This check is done at the end of each iteration.

Suppose at the end of the kth iteration, we have computed column k of L and U (i.e. column k is the current column). Note that columns 1 to $k-1$ of L and U were computed and committed (finalized) in previous iterations.

Let $v = Uy$. This indicates that v is effectively the kth column of U. Now let $u = xL$. This means that $uv = xLUy$. Since $L_{i,j} = 0$ for all $i < j$,

$$u_i = \sum_{i=j}^{n} x_i L_{i,j}.$$

Note that during the kth iteration, one only needs to compute element u_k and include it with the previously computed values of u, because the elements u_1 to u_{k-1} only involve the columns of L that were computed in previous iterations. Therefore, it is unnecessary to recalculate these elements. In addition, since all elements below the kth element (the diagonal element) of v are zeroes, it is not necessary (nor desirable) to compute the elements u_{k+1} to u_n. Thus, the computation of $u = xL$ does *not* involve the uncommitted columns of L (columns $k+1$ through n). In other words, during iteration k, one only needs to utilize the committed columns 1 through k of L to compute the vector u. The computation of the element u_k is also relatively straightforward, since it is easy to access elements of a column of a matrix stored in the CCS format. Note that we first compute u and v separately, and then compute uv. We minimize the overhead of computing the dot product uv by using parallel reduction on the GPU.

The calculation of xAy is straightforward. During the kth iteration, Ay is basically the kth column of A. Then the computation of xAy only involves summing up the elements of Ay, and can be computed using parallel reduction. This method allows us to check the elements of the current computed column of L and U with minimal overhead. We would like to point out that partial pivoting does *not* have any effect on the value of xAy or $xLUy$, thus it is not necessary to permute the elements in the relevant columns of A, L and U in order to compute xAy or $xLUy$ – this is good, because permutation operations are expensive.

It is important to note that the right-looking LU method requires $n-1$ iterations and the final column (column n) never becomes the current column; in other words, this column is computed using only trailing matrix updates. Thus, it is necessary to do an additional check for this column after factorization is complete.

Figure 5.4 illustrates the set of elements that are checked by the duplicate checking and the LUIC method during one particular iteration of the right-looking LU factorization. The blue regions outline the committed elements that are *not* checked by both the duplicate checking and the LUIC method during the current iteration; checking these elements is not necessary because these elements have already been checked in a previous iteration. The red region indicates elements that have been committed (up to and including the current iteration) but are *not* checked by the LUIC method during the current iteration. These elements will be checked in subsequent iterations. The darker green regions outline the committed elements that are checked during the current iteration (by both the duplicate checking and the LUIC method) while the lighter green region indicates non-committed elements that are checked (by the duplicate checking method only) during that same iteration. Finally the orange region shows the non-committed elements that are not checked by the LUIC method in the current iteration.

The error coverage of this method is close to 100%. This means that any error affecting the elements of the current column should result in a discrepancy between the values of xAy and $xLUy$. We would like to point out, however, that unlike the duplicate check, the LUIC method *cannot* detect errors in the trailing matrix until subsequent iterations. Fortunately, in such an event, when an error occurs in some of the columns updated during the trailing matrix update, and not in any of the elements in the current column, then when those columns become the *current* column in a later iteration, the LUIC method will be able to detect any errors in that column.

It is also worth mentioning that the method of checkpointing and recomputation is *not* a suitable recovery method to use in conjunction with the LUIC method, because errors can occur in the trailing matrix. These errors will not be detected by the LUIC method until subsequent iterations – by then the checkpoint will be corrupted and recovery of these errors is no longer possible by reloading that checkpoint.

We chose our threshold such that an error is detected if $|xLUy - xAy|/|xAy| > 10^{-3}$ for $|xAy| \geq 1$, or if $|xLUy - xAy| > 10^{-3}$ for $|xAy| < 1$.

5.2.5 Determination of Threshold Values

We empirically determined the above threshold values for the LUIC method. Our procedure is as follows: For each of our 56 input matrices, we ran our sparse right-looking LU factorization application that uses the LUIC method, without any fault injection. We then print the values of $xLUy$ and xAy at the end of every iteration during these runs. The aim here is to identify suitable threshold values to avoid getting runs with false positives during the checking process. To achieve this, we noted the values of $|xLUy - xAy|/|xAy|$ (for cases where $|xAy| \geq 1$) and $|xLUy - xAy|$

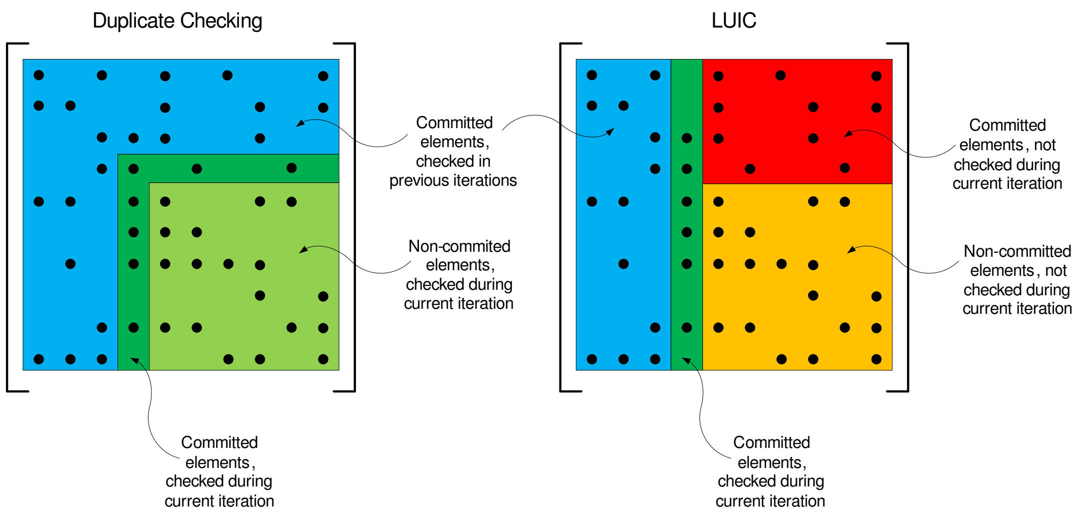

Duplicate Checking
LUIC
Committed elements, checked in previous iterations
Non-commited elements, checked during current iteration
Committed elements, checked during current iteration
Committed elements, not checked during current iteration
Non-committed elements, not checked during current iteration
Committed elements, checked during current iteration

Figure 5.4 Elements checked by duplicate checking vs LUIC, on right-looking LU factorization

(for cases where $|xAy| < 1$) that had the greatest magnitude, out of all the reported values for all matrices. We then set our thresholds to these values, so that no false positives will occur during an error-free run. This does mean that for certain matrices, the threshold can be made tighter still, without resulting in runs with false positives. We used a similar procedure to determine suitable threshold values for the other checking methods.

5.2.6 Proposed Fault Recovery Methods

We will now introduce two ways to recover the correct value of the elements in a column, once a fault is detected in some iteration. These are discussed below.

5.2.6.1 Recovery via Left-looking LU

The procedure for computing a column of L and U via the formulas for left-looking LU can be exploited to enable the reliable recovery of a single column (i.e. the current column k) of L and U, without requiring the use of checkpoints. However, it does require that the previously computed $k-1$ columns of L and U are free from errors in order to recover column k properly.

Suppose during the kth iteration, errors are detected in the currently computed column k. To recover the correct values of this column, one can simply use the following procedure:

1. Solve $L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k}$ for column vector $U_{1:k-1,k}$
2. Compute scalar $U_{k,k} = A_{k,k} - L_{k,1:k-1}U_{1:k-1,k}$
3. Compute column vector $L_{k+1:n,k} = \frac{A_{k+1:n,k} - L_{k+1:n,1:k-1}U_{1:k-1,k}}{U_{k,k}}$

Algorithm 9 illustrates the left-looking LU recovery procedure used in conjunction with the LUIC method (xAy vs $xLUy$ checking). Note that in order to properly recover the correct values, the rows of columns 1 to $k-2$ will first need to be retroactively permuted using the 2nd through $(k-1)$th elements in the permutation vector p, before applying the above formulae (i.e. elements 2 to $k-1$ of p are applied to the 1st column, elements 3 to $k-1$ of p are applied to the 2nd column, and so on). We assume that p has no errors in its computed elements. The left-Looking LU recovery process is particularly sensitive to the number of columns that need correction, since the recovery of *each* column has $O(n)$ complexity on a GPU.

5.2.6.2 Recovery via Right-looking LU

One can also use the algorithm for right-looking LU factorization itself to correct errors that occur during factorization. Suppose an error occurs during iteration k. Assuming that the LUIC method is used to detect errors, then we have to conservatively assume that all elements in columns

k through n are potentially erroneous. The procedure to recover the correct values of the elements in these columns is as follows:

1. Set the elements in columns k to n of $L+U-I$ to the corresponding elements in the original matrix A
2. For $j = 1$ to $k-1$, perform the row pivoting (using element j of vector p) and trailing matrix update steps of iteration j on columns k through n
3. Redo iteration k, i.e. perform the row pivoting, column update and trailing matrix update steps (this step is not necessary if errors are detected in the last column n)

This method, like the left-looking LU recovery procedure, does not require the use of checkpoints. This point is important when the LUIC method is used as the error detection scheme, because it is possible that errors occur only in the trailing matrix portion. Such errors will not be detected by that checking method until subsequent iterations, which means that the checkpoint itself would have been corrupted by the errors. Thus, step 2 above is necessary to reconstruct those elements that are part of the trailing matrix.

Compared to the left-looking LU recovery procedure, the right-looking LU recovery procedure is less sensitive to the number of columns corrupted by a single fault. Note that in this recovery procedure, the elements that are known to be free of errors (i.e. those in columns 1 to $k-1$) are *not* recomputed. Algorithm 10 illustrates the right-looking LU recovery procedure, again in conjunction with the LUIC method.

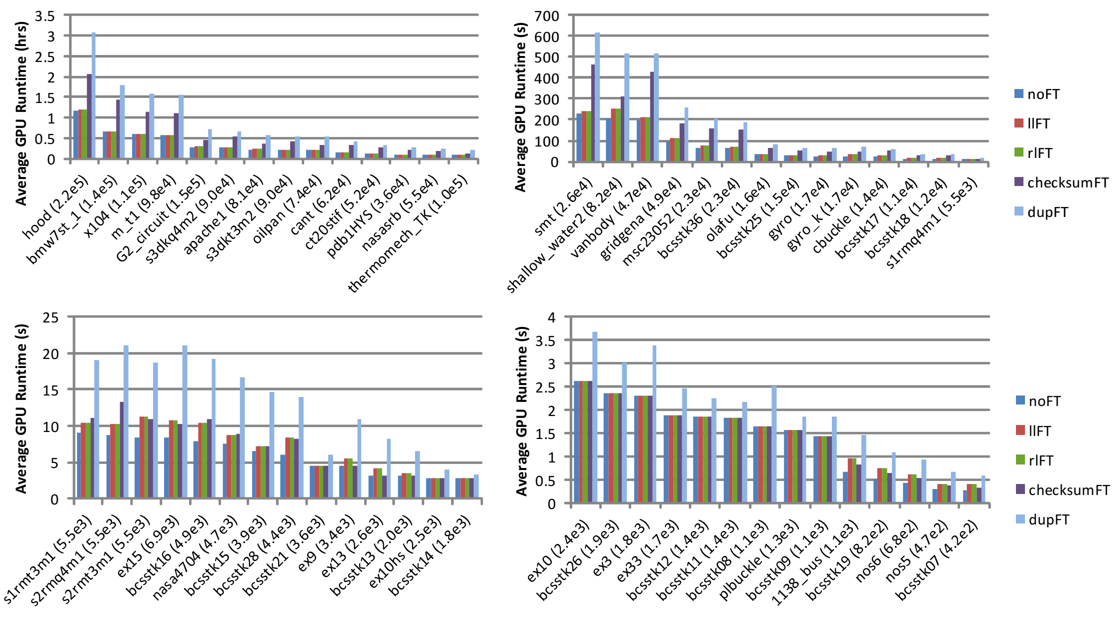


Figure 5.5 Performance of various FT schemes with no fault injection

5.2.7 Properties for Fault Tolerance Schemes

In this section, we list three desirable properties that FT schemes should possess. For the rest of this paper, we will compare four FT schemes, together with the basic LU factorization implementation that does not utilize any FT. The first FT scheme uses the LUIC method and recovers the correct elements of an erroneous column using left-looking LU strategy – we will refer to this scheme as “llFT”. The second FT scheme, “rlFT”, also uses the LUIC method, but utilizes the right-looking LU recovery procedure. The third FT scheme, which we call “dupFT”, utilizes the duplicate checking method, and does recovery via checkpointing and recomputation. We refer to the fourth FT scheme as “checksumFT”, which uses the checksum method and performs recovery via re-execution of the current iteration. We will refer to the scheme that does not use any FT as “noFT”. Note that our FT methods, the llFT and rlFT schemes, differ from the FT method proposed

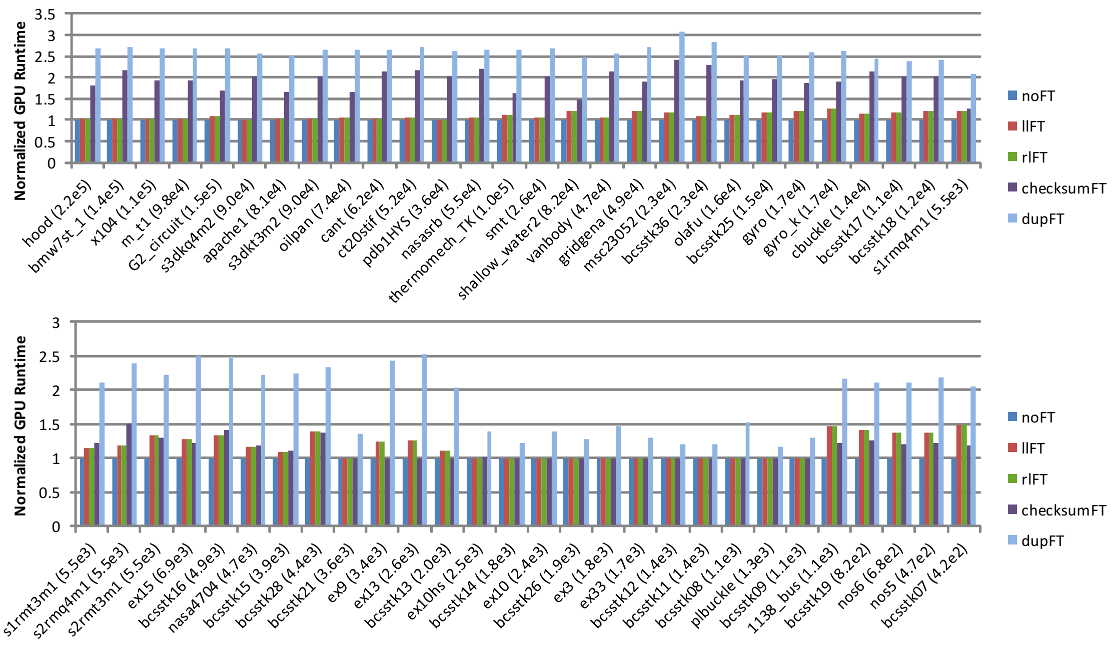


Figure 5.6 Performance of various FT schemes with no fault injection, normalized to noFT scheme

by Du et al [27] in that our FT methods do not use checksums, and do not require checkpoints to recover from errors.

- **Correctness**: The FT scheme checks all elements that are committed during the current iteration (the scheme does not need to check any element that is not computed or committed). All four FT schemes satisfy the correctness requirement.

- **Completeness**: The FT scheme eventually checks all elements that are computed, by the end of factorization of the original input matrix. In this case, all four schemes meet the requirements of the "completeness" property adequately. It is worth pointing out, however, that the LUIC method (used in the llFT and rlFT schemes) can only detect errors that occur in the currently computed column of any iteration – it cannot detect errors if they *only* occur

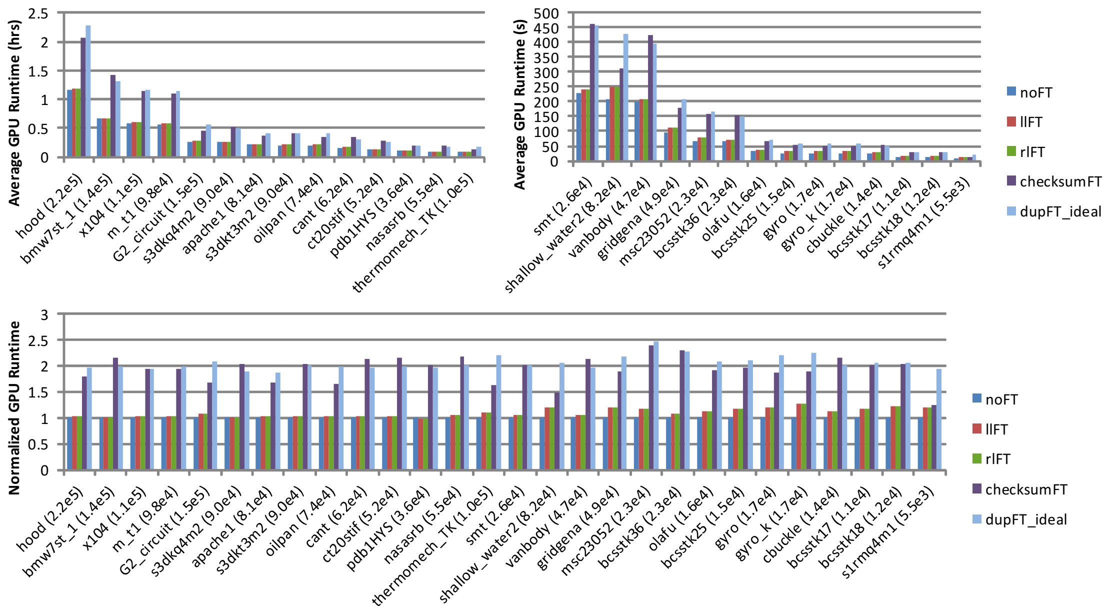


Figure 5.7 Performance of various FT schemes with no fault injection, compared to an ideal dupFT scheme

in the trailing matrix update of that iteration. In contrast, the duplicate checking method (used in the dupFT scheme) and the checksum method (used in the checksumFT scheme) can detect errors that occur in the trailing matrix portion during the iteration in which the errors occur. Fortunately, any such errors will be eventually detected by the LUIC method in subsequent iterations.

- **Latency**: The number of iterations that elapse, from the point an error occurs to the point when the error is actually detected by the FT scheme, if it is detected in the first place. If the errors are confined to the current column that is committed, then all four FT schemes can detect errors by the end of the same iteration in which they occur. However, the LUIC method cannot detect errors in the trailing matrix until subsequent iterations. The duplicate

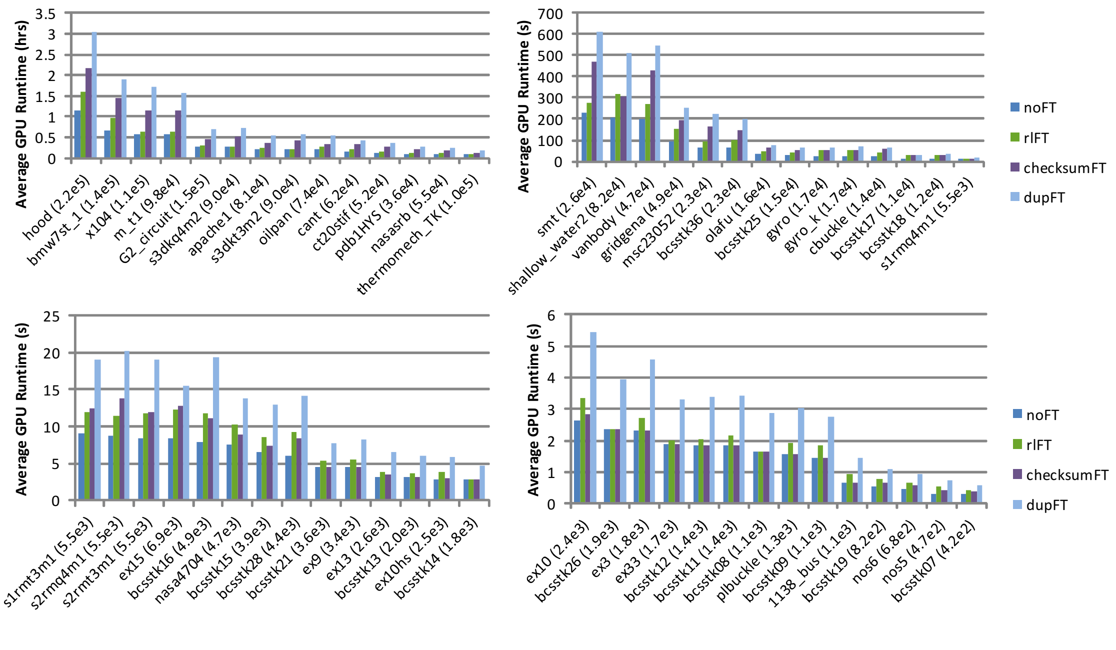


Figure 5.8 Performance of various FT schemes with fault injection

checking method and the checksum method can detect errors that occur in the trailing matrix portion, during the same iteration that those errors occur.

5.3 Experimental Methodology

We ran all experiments using a GPU compute cluster. It contains 14 compute nodes, 12 of which contain GTX 1080 GPUs. Each experiment is run on a single node using one of the four GTX 1080 cards. Table 7.1 shows the specifications of one such node.

Each experiment runs an instance of our right-looking LU application using one of the aforementioned FT schemes, along with the no fault tolerance, i.e. noFT, scheme. We used double-precision floating point numbers. We conducted the experiments in two main categories: performance of FT schemes without fault injection and performance of FT schemes with fault injection.

Table 5.1 Specifications of a compute node

Component	Quantity
Intel Xeon E5-2650 v3 CPU @ 2.30GHz	2
128GB DDR4 ECC RAM	-
NVIDIA GTX 1080	4

We chose 56 sparse, real square matrices from the University of Florida sparse matrix collection [59] as input matrices for our experiments.

For each experimental run that involves fault injection, we inject at most one fault, according to our fault model that is described in Section 5.2.1. We measured the average time needed for factorization of the input matrix, and ran thousands of experimental runs to ensure that our results are statistically significant. We compared the factorization time for any run with the corresponding factorization time with no FT. We treat any run, in which $|xLUy - xAy|/|xAy| > 10^{-3}$ for $|xAy| \geq 1$, or if $|xLUy - xAy| > 10^{-3}$ for $|xAy| < 1$, as erroneous, where $x = [1 \ \ 1 \ ... \ 1]$, $y = [1 \ \ 1 \ ... \ 1]^T$, and L, U are the final factor matrices at the end of LU factorization.

5.4 Performance of Fault Tolerant Schemes

5.4.1 Performance of FT Schemes in the Absence of Faults

Here, the experiments basically measure the typical, expected overhead of the fault detection strategies. Since no faults are injected, the recovery portions of the schemes are never invoked and the additional runtime of the various FT schemes over the noFT scheme should be entirely attributable to the cost of checking.

Figure 5.5 compares the GPU runtime of each FT scheme with respect to the runtime of the our implementations with no FT, in the absence of faults, for the 56 matrices. The width of the matrix is shown in parenthesis. The upper plot shows the raw runtimes for the larger 28 matrices, while the lower plot shows the raw runtimes for the remaining smaller matrices. In a similar fashion, figure 5.6 compares the GPU runtime for each FT scheme, normalized to the runtime of the corresponding noFT scheme of each matrix. As shown in figure 5.5, the LUIC scheme (used in llFT and rlFT) has significantly lower overhead than the duplicate checking scheme (dupFT) – for each matrix, the runtime of the LUIC scheme is always significantly less than the corresponding runtime of the duplicate checking scheme. In general, the LUIC scheme has low overhead; on the other hand, the duplicate checking scheme is much less efficient. As can be observed from figure 5.6, there are a few matrices for which the llFT and rlFT schemes have higher normalized overhead, approaching 1.5 in some cases, but these are restricted to the smaller matrices. A more interesting discussion can be made with respect to the checksum detection method, which is used in the checksumFT scheme. For the smaller matrices, the checksum method is very efficient – its performance is comparable to the LUIC scheme, and even outperforms the LUIC scheme for some matrices. However, for larger matrices, the checking method has significantly greater overhead than the LUIC scheme. This is mainly due to the fact that the overhead of checking the elements of the trailing matrix increases greatly with the size of the matrix. In addition, the checksum method requires row accesses to compute the row sums, which is not efficient for sparse matrices in the CCS format. Despite these limitations, the checksum method is still always faster than the duplicate checking scheme.

It is also useful to compare the llFT, rlFT and checksumFT schemes with the *ideal* implementation of the dupFT scheme, by assuming that it is somehow possible to reduce the overhead of the duplicate checks to zero. Then in this ideal scheme, the additional overhead incurred over the noFT scheme should be completely attributable to the overhead of computing the redundant $L + U - I$ factor matrix. We refer to such a scheme as "dupFT_ideal".

Figure 5.7 compares the GPU runtime of the llFT and rlFT schemes with the dupFT_ideal scheme, for only the larger 28 matrices, with raw runtimes shown in the upper plot and normalized runtimes in the lower one. We obtained the runtime results for dupFT_ideal by subtracting the overhead of the duplicate checks from the corresponding runtime results of dupFT. It can be observed that the llFT and rlFT schemes still perform well, even when compared against the ideal version of the dupFT scheme. The checksumFT scheme now runs *longer* than dupFT_ideal for a few matrices, but in general, the checksumFT scheme still outperforms the ideal version of the dupFT scheme.

5.4.2 Performance of FT Schemes in the Presence of Faults

With the injection of faults, we are now able to measure the expected overhead of checking and recovery. We can also measure the error coverage of each of the FT schemes.

In our fault injection experiments, the llFT, rlFT, checksumFT and dupFT checking schemes all exhibit excellent error detection coverage, with a majority of errors getting detected (over 99%) for all matrices. This satisfies the completeness requirement.

Figure 5.8 shows the raw GPU runtime of these schemes (excluding the llFT scheme), for the 56 matrices in the case where the errors caused by the injected faults were successfully detected and corrected. Similarly, Figure 5.9 shows the normalized runtimes (with respect to the noFT scheme). We chose to exclude the llFT scheme from the latter two figures because its overhead is generally too high compared to the other three schemes and this would not facilitate a good comparison between the other schemes. These are compared to the runtime of noFT, and only for cases where the errors were detected and corrected properly. Note that we do include an additional plot comparing the llFT scheme with the other schemes for 13 selected matrices, with the vertical axis in log-scale – see figure 5.10.

As can be seen in figures 5.8 and 5.9, the overhead of the rlFT and dupFT schemes go up when recovery was required. However the overhead of the rlFT scheme is, for the vast majority of cases, significantly less that the dupFT scheme. In general, the normalized overhead of the rlFT scheme varies from around 1.2 to 1.5. For a very small minority of matrices, the normalized overhead of the rlFT scheme is higher, with values close to 2, but these cases are confined to the smaller matrices. Even in those cases, the performance of the rlFT scheme is still better than the dupFT scheme. The normalized overhead of the checksumFT scheme does not go up appreciably in the presence of errors; this is unsurprising as the recovery overhead of the checksumFT scheme is very low. For a few matrices like shallow_water2, this results in the rlFT scheme actually having a greater overhead compared to the checksumFT scheme (as opposed to our experiments with no faults injected – for shallow_water2, the rlFT scheme had a lower overhead compared to the checksumFT scheme).

The llFT scheme with its left-looking LU recovery strategy reliably recovers the correct values of the elements of any corrupted column, but, as can be observed in figure 5.10, has significantly high overhead – in some cases by as much as two orders of magnitude. These can be explained by the fact that these cases all involve the recovery of *multiple* corrupted columns, often numbering in the thousands, as illustrated by figure 5.11, which corroborates our observation that the left-looking LU recovery procedure is sensitive to the number of columns requiring correction. In reality, one should expect much fewer columns to require correction, and correspondingly the overhead of left-looking LU recovery should be much smaller.

Under an environment with a very high fault rate, the checksumFT scheme will outperform the llFT scheme, and may perhaps be faster than even the rlFT scheme. However, in general, the rlFT scheme possesses the best of both worlds: very low overhead of error checking in the typical scenario where no errors occur, and relatively low recovery overhead in cases where errors do manifest.

In situations where errors tend to last for longer durations, which can lead to the corruption of many elements across a large number of columns, the rlFT scheme offers the best balance of low error checking and correction overhead, with no checkpointing required. Even in cases where only a few elements are corrupted within a limited number of columns, the rlFT scheme should still perform well with relatively low recovery overhead, although in those situations, the llFT scheme (or checksumFT scheme) will likely have a lower overhead of recovery than rlFT.

In the next chapter, we will discuss our FT strategies for the PCG and BiCGSTAB iterative solvers.

Algorithm 7 Compute sparse left-looking factorization $PA \in \mathbf{R}^{n\times n} = LU$

```
1: procedure LeftLU($A$)
2:     do symbolic factorization to find non-zero struc. of L+U-I
3:     L+U-I replaces A at end of factorization
4:     for $k = 1 : n$ do
5:         solve $L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k}$
6:         for $U_{1:k-1,k}$
7:         $U_{k,k} = A_{k,k} - L_{k,1:k-1}U_{1:k-1,k}$
8:         determine $\mu$ with $k < \mu \leq n$ such that $L_{\mu,k}$ is
9:         the max abs element
10:        if $|L_{\mu,k}| > |U_{k,k}|$ then
11:            swap $L_{\mu,k}$ with $U_{k,k}$
12:            $p_k = \mu$
13:        else
14:            $p_k = k$
15:        end if
16:        $L_{k+1:n,k} = \frac{A_{k+1:n,k} - L_{k+1:n,1:k-1}U_{1:k-1,k}}{U_{k,k}}$
17:    end for
18:    return $A, p$
19: end procedure
```

Algorithm 8 checksumFT right-looking LU

```
procedure CHECKSUMFT RIGHTLU($A$)
    do sym. factorization to find non-zero struc. of L+U-I
    L+U-I replaces A at end of factorization
    compute row and col checksums for A
    for $k = 1 : n - 1$ do
        compute elements in current col $k$
        compare col sum and checksum of current col $k$
        compare row sum and checksum of current row $k$
        if errors detected then
            redo this iteration
        end if
        compute elements in trailing matrix
        compare row/col sums and checksums of trail. matrix
        if errors detected then
            recover correct values using appropriate chksums
        end if
    end for
    return $A, p$
end procedure
```

Algorithm 9 llFT right-looking LU

```
1:  procedure LLFT RIGHTLU($A$)
2:      do sym. factorization to find non-zero struc. of L+U-I
3:      L+U-I replaces A at end of factorization
4:      for $k = 1 : n - 1$ do
5:          compute elements in current col $k$ and trailing matrix
6:          perform LUIC for current col $k$:
7:          $x = [1\ 1\ ...\ 1]$ ($x_i$ = 1 for all $i$)
8:          $y = [0\ ...\ 0\ 1\ 0\ ...\ 0]^T$ ($y_k$ = 1)
9:          compute and compare $xAy$ with $xLUy$
10:         if errors detected then
11:             recover current column $k$ via left-looking LU:
12:             solve $L_{1:k-1,1:k-1}U_{1:k-1,k}$
13:             $= A_{1:k-1,k}$ for $U_{1:k-1,k}$
14:             $U_{k,k} = A_{k,k} - L_{k,1:k-1}U_{1:k-1,k}$
15:             $L_{k+1:n,k} = \frac{A_{k+1:n,k} - L_{k+1:n,1:k-1}U_{1:k-1,k}}{U_{k,k}}$
16:         end if
17:     end for
18:     perform LUIC for last col $n$
19:     if errors detected then
20:         recover column $n$ via left-looking LU
21:     end if
22:     return $A, p$
23: end procedure
```

Algorithm 10 rlFT right-looking LU

```
procedure RLFT RIGHTLU($A$)
    do sym. factorization to find non-zero struc. of L+U-I
    L+U-I replaces A at end of factorization
    for $k = 1 : n-1$ do
        compute elements in current col $k$ and trailing matrix
        perform LUIC for current col $k$:
        $x = [1\ 1\ ...\ 1]$ ($x_i$ = 1 for all $i$)
        $y = [0\ ...\ 0\ 1\ 0\ ...\ 0]^T$ ($y_k$ = 1)
        compute and compare $xAy$ with $xLUy$
        if errors detected then
            recover columns $k$ to $n$ via right-looking LU:
            set elements in cols $k$ to $n$ to corresponding
            elements of original matrix $A$
            for $j = 1 : k-1$ do
                redo iteration $j$ row pivoting and trailing
                matrix update for cols $k$ to $n$
            end for
            redo iteration $k$ row pivoting
            redo iteration $k$ column update for col $k$
            redo iteration $k$ trailing matrix update
            for cols $k+1$ to $n$
        end if
    end for
    perform LUIC for last col $n$
    if errors detected then
        recover column $n$ via right-looking LU:
        set elements of col $n$ to those of original matrix $A$
        for $j = 1 : n-1$ do
            redo iteration $j$ row pivoting and trailing matrix
            update for col $n$
        end for
    end if
    return $A, p$
end procedure
```

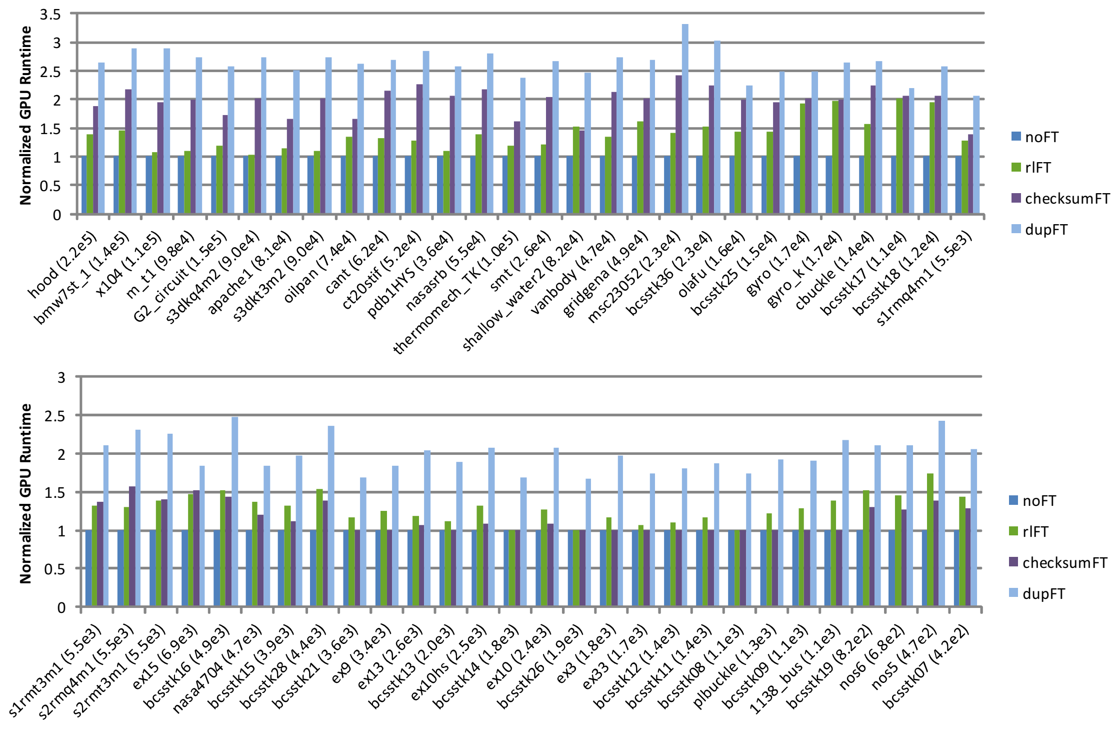


Figure 5.9 Performance of various FT schemes with fault injection, normalized to noFT scheme

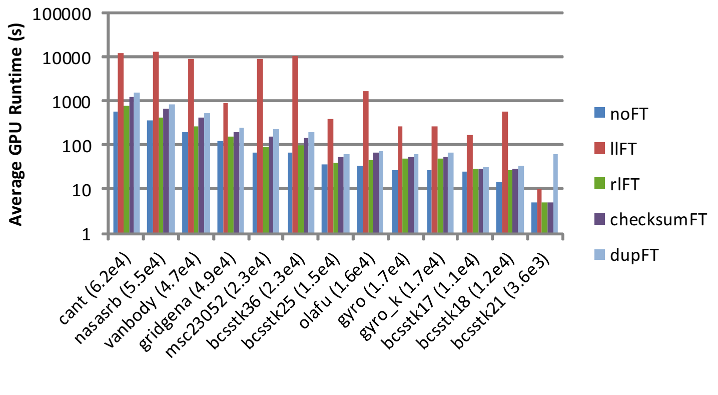


Figure 5.10 Performance of various FT schemes with fault injection (including llFT)

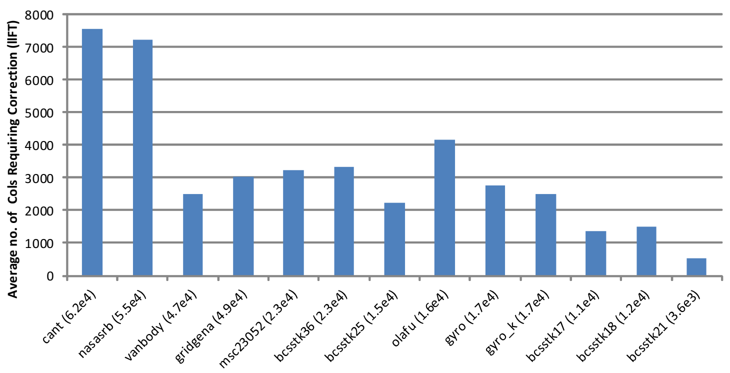


Figure 5.11 Average number of columns requiring correction for the llFT scheme

Chapter 6

Fault Tolerance in Iterative Solvers

One way to solve a system of linear equations is by directly solving the linear system $Ax = b$, which will usually require some form of matrix factorization of the coefficient matrix A. Another way is to use an iterative solver to arrive at a solution indirectly – an initial solution vector x_0 is guessed, then the solver iteratively refines the solution vector x_k. The solver is stopped once the solution vector reaches an acceptable level of accuracy (i.e. converges).

We developed our own GPU-based implementations of the preconditioned conjugate gradient (PCG) and the biconjugate gradient stabilized (BiCGSTAB) iterative solvers. In our implementations, we set the initial guess, x_0, to be a zero vector, and use 32-bit floating point numbers. Our implementations use the CRS sparse matrix format. As mentioned before, these solvers are used to solve very large linear systems of equations. Some common applications of iterative solvers include the solving of discretized elliptic partial differential equations (PDEs), like the Poisson equation [64].

6.1 Overview of Preconditioned Conjugate Gradient (PCG)

The conjugate gradient method, originally proposed by Hestenes et al [65], is a popular solver that is used to find a solution to a linear system $Ax = b$, where the matrix A is symmetric positive definite. However, in some cases where the matrix A is not well conditioned, the conjugate gradient method can take many iterations to converge. PCG addresses this issue by preconditioning the linear system such that the resulting matrix of coefficients becomes either well-conditioned, or it has only a few distinct eigenvalues. This reduces the number of iterations needed to converge to an accurate solution than would otherwise have been required. The algorithm for PCG, reproduced from Golub et al [52], is shown in Algorithm 11.

Algorithm 11 Compute solution x satisfying $Ax = b$

```
procedure PCG($A, b, x$)
    $x_0 =$ initial guess; $k = 0$; $r_0 = b - Ax_0$
    while $||r_k||_2^2 >$ convergence tolerance do
        solve $Mz_k = r_k$ for $z_k$
        $k = k + 1$
        if $k = 1$ then
            $p_1 = z_0$
        else
            $\beta_k = \frac{r_{k-1}^T z_{k-1}}{r_{k-2}^T z_{k-2}}$; $p_k = z_{k-1} + \beta_k p_{k-1}$
        end if
        $\alpha_k = \frac{r_{k-1}^T z_{k-1}}{p_k^T A p_k}$; $x_k = x_{k-1} + \alpha_k p_k$
        $r_k = r_{k-1} + \alpha_k A p_k$
    end while
    return $x_k$
end procedure
```

6.2 Overview of Biconjugate Gradient Stabilized (BiCGSTAB)

The biconjugate gradient method can be used to find a solution to linear systems where the coefficient matrix A is positive definite but not necessarily symmetric. However, it suffers from irregular convergence behavior and may take many iterations to converge to a satisfactory solution.

To mitigate these problems, van der Vorst [66] introduced a variant, called BiCGSTAB, that has faster and smoother convergence. The algorithm for BiCGSTAB is shown in Algorithm 12.

Algorithm 12 Compute solution x satisfying $Ax = b$

```
1: procedure BiCGSTAB($A, b, x$)
2:     $x_0 =$ initial guess; $k = 0$; $r_0 = b - Ax_0$
3:     $\rho_0 = \alpha_0 = \omega_0 = 1$; $v_0 = p_0 = 0$
4:     while $||r_k||_2^2 >$ convergence tolerance do
5:         $k = k + 1$
6:         $\rho_k = r_0^T r_{k-1}$; $\beta_k = \frac{\rho_k}{\rho_{k-1}} \frac{\alpha_{k-1}}{\omega_{k-1}}$
7:         $p_k = r_{k-1} + \beta_k (p_{k-1} - \omega_{k-1} v_{k-1})$
8:         $v_k = Ap_k$; $\alpha_k = \frac{\rho_k}{r_0^T v_k}$
9:         $s_k = r_{k-1} - \alpha_k v_k$; $\omega_k = \frac{(As_k)^T s_k}{(As_k)^T (As_k)}$
10:        $x_k = x_{k-1} + \alpha_k p_k + \omega_k s_k$; $r_k = s_k - \omega_k As_k$
11:    end while
12:    return $x_k$
13: end procedure
```

6.3 Invariant properties for PCG

PCG possesses several properties that can be used as lightweight error checks. First, successive direction vectors, p_k, are conjugate to each other, i.e. $p_i^T Ap_j = 0$ for all $i \neq j$. If one checkpoints the computation of Ap_k in line 12 of Algorithm 11, then this check can be performed in $O(n)$ complexity. Second, the vectors z_{k-1} and r_k are orthogonal, i.e. $z_{k-1}^T r_k = 0$ for all $k \geq 1$. Since this check involves a vector-vector multiplication, its runtime complexity is $O(n)$. Third, one can also show that the vectors p_k and r_k are orthogonal. This check also has $O(n)$ runtime complexity. One can use either one or more of these checks. In our implementation, we only use the invariant $z_{k-1}^T r_k = 0$, and the proof is shown below.

Lemma 1. *Let z_{k-1} and r_k denote the current computed versions of z and r respectively at the end of the kth iteration. Then for all $k \geq 1$, $z_{k-1}^T r_k = 0$.*

Proof. From Golub et al [52], we have

$$r_j^T M^{-1} r_i = 0$$

for all $i \neq j$. Then

$$[M^{-1} r_i]^T r_j = 0$$

is also true. Therefore, we have

$$\begin{aligned} z_{k-1}^T r_k &= [M^{-1} r_{k-1}]^T r_k \\ &= 0 \end{aligned}$$

since $k - 1 \neq k$. □

6.4 Invariant properties for BiCGSTAB

The BiCGSTAB solver does *not* compute two series of direction vectors that are biconjugate with each other. As such, there is no biconjugate property (i.e. $p_i^T A p_j \neq 0$ for some $i \neq j$) to exploit in this solver. Also, the p_k vectors are not necessarily conjugate to each other. However, we show below that s_k and r_0 are orthogonal to each other: $s_k^T r_0 = 0$ for all $k \geq 1$. This orthogonality can be used as a lightweight check, and its runtime complexity is $O(n)$. The proof is as follows.

Lemma 2. *Let s_k denote the current computed version of s at the end of the kth iteration. Then for all $k \geq 1$, $s_k^T r_0 = 0$.*

Proof. From van der Vorst [66],

$$[Q_k(A)P_j(A)r_0]^T r_0 = 0$$

for all $k < j$, where $Q_k(A) = (I - \omega_k A)(I - \omega_{k-1} A)...(I - \omega_1 A)$ and $P_j(A)$ is some polynomial in A. Now,

$$\begin{aligned} r_k &= Q_k(A)P_k(A)r_0 \\ &= (I - \omega_k A)Q_{k-1}(A)P_k(A)r_0 \\ &= Q_{k-1}(A)P_k(A)r_0 - \omega_k A Q_{k-1}(A)P_k(A)r_0 \end{aligned}$$

Comparing the above equation with

$$r_k = s_k - \omega_k A s_k$$

it can be seen that

$$s_k = Q_{k-1}(A)P_k(A)r_0$$

Therefore,

$$\begin{aligned} s_k^T r_0 &= [Q_{k-1}(A)P_k(A)r_0]^T r_0 \\ &= 0 \end{aligned}$$

since $k - 1 < k$.

□

6.5 Fault Detection Methods

In this subsection, we describe several FT methods for PCG and BiCGSTAB. Some of these methods will not apply to a particular iterative solver because the solver lacks the properties required by that FT method.

6.5.1 Dual Modular Redundancy

The dual modular redundancy (DMR) method is a well-known technique for fault detection. We duplicate the computations by having the basic linear algebra kernels call twice the number of threads – the extra threads are used to perform the redundant calculations. At the end of each iteration, each element of the original vector x_k is compared with its redundant counterpart. If any pair of elements differ by more than a specified threshold, an error is assumed. When this happens, we compare the residual norms (i.e. $||b - Ax_k||_2$) for both the original and redundant computations, then choose the norm that is smaller. We then keep the vector x_k that corresponds to that smaller norm and overwrite the *other* version of x_k with the chosen x_k. Fault recovery (as described in Section IV, Part D) is then initialized using the chosen x_k. The overhead of this method is expected to be high – basically one can expect the runtime of this method to take about twice as long as the runtime for the non fault tolerant iterative solver.

6.5.2 Reset Method

The reset method is a passive FT scheme. Instead of checking for errors at the end of each iteration, the reset method simply restarts the iterative solver every R iterations, regardless of whether errors are present or not. During each reset or restart, the current computed version of

x_k is kept, the new initial residual vector $r_0 = b - Ax_k$ is calculated and k is reset to 0. The overhead of this method depends on the value of R – if R is small, then it may take much longer to converge, whereas with large value of R, the fault tolerance is expected to be compromised. In my implementation, we set $R = 10$. This value provides faster convergence in general for the set of data used in our study.

6.5.3 Residual Checking

This method uses a simple residual check at the end of each iteration – basically, we check whether $||r_k + Ax_k - b||_\infty$ is greater than a threshold at the end of each iteration. If so, an error is detected and the iterative solver is restarted from the beginning. The overhead of this check depends on the sparsity of A. If A is a non-sparse matrix, then the runtime complexity is $O(n^2)$. The value of the threshold should be carefully chosen. It may have to depend on the elements of A and b. In our implementation, we chose a threshold of $10^{-7}||b||_2$.

6.5.4 Conjugate Invariant Checking

Chen [33] introduced an error checking strategy for PCG, where the conjugate property of the direction vectors p_k is tested, and also used a residual check to detect whether there were any errors in the computation of x_k (since checking the conjugate property alone will not detect errors in x_k). In Chen's implementation, the error detection bounds are set to $(p_k^T Ap_{k-1})/||p_k||_2.||Ap_{k-1}||_2 > 10^{-10}$ and $||r_k + Ax_k - b||_2/||A||_1.||b||_2 > 10^{-10}$. However, when we tried to use these bounds, the false positive rate was too high. Therefore, in our implementation, we relax both bounds to 10^{-4}, and perform the conjugate and residual checks during each iteration, and if an error is detected, the iterative solver is restarted from the beginning, preserving the current value of x_k. The overhead for each check (per iteration) is $O(n)$, if Ap_{k-1} can be checkpointed within PCG. Note that this strategy is not applicable to BiCGSTAB.

6.5.5 Orthogonal Invariant Checking

In this fault tolerance method, we utilize the orthogonal relationships mentioned in Section IV, Part A. These checks are done at the end of each iteration. We check whether the normalized dot product during the kth iteration exceeds a certain bound. In other words, for PCG, we check whether $(z_{k-1}^T r_k)/||z_{k-1}||_2$ exceeds a specified bound. For BiCGSTAB, we check whether $(s_k^T r_0)/||r_0||_2$ exceeds a specified bound. One can also take advantage of the observation that successive dot product values generally decrease monotonically as the iterative solver progresses (though sometimes the values can fluctuate). As such, at the end of each iteration, we have an additional check where we store the previous normalized dot product value and then check whether the current normalized dot product value is much larger than its previous value – for PCG, we check whether $(z_{k-1}^T r_k||z_{k-2}||_2)/(z_{k-2}^T r_{k-1}||z_{k-1}||_2) > 100$; for BiCGSTAB, we check whether $(s_k^T r_0)/(s_{k-1}^T r_0) > 50$. If a check detects an error (i.e., the vectors are not orthogonal), the iterative solver is restarted from the beginning.

However, it is possible for the result x_k to be corrupted without violating the orthogonal property of the aforementioned pairs of vectors used in our checks. We therefore re-execute the update of x_k (line 11 of Algorithm 11 and line 10 of Algorithm 12 respectively) and assume that an error has occurred if the maximum norm of the difference between the original and the redundant x_k vectors exceeds a specified threshold. In our implementation, the threshold is 10^{-3}. When such an error is detected, the calculation for x_k is recomputed (writing the values of x_k in the original memory location). Note that if this happens, and the orthogonal check for that iteration *passes*, there is no need to restart the iterative solver. The overhead of doing the checks per iteration is $O(n)$.

6.6 Fault Recovery Method

All of our fault tolerance methods utilize the same fault recovery method in general: When an error is detected, we restart the iterative solver, but keep the current computed version of x_k (treating it as x_0) and calculate the new initial residual vector $r_0 = b - Ax_k$.

6.7 Experimental Methodology

We implemented our own CUDA programs for the iterative solvers. We used single-precision floating point numbers, with the input matrix A in the CRS format. In all experiments, we chose the initial guess $x_0 = 0$, and use the 2D discrete Poisson equation as our input linear system, with various random b vectors (some elements of b are set to 0, and the rest are uniformly distributed between -1×10^{10} and 1×10^{10}). A is an $n \times n = m^2 \times m^2$ block tridiagonal matrix and D is a $m \times m$ matrix, as shown below. I is the $m \times m$ identity matrix. We set the convergence tolerance to $10^{-6}||b||_2^2$.

$$A = \begin{bmatrix} D & -I & 0 & \cdots & 0 \\ -I & D & -I & \cdots & 0 \\ 0 & -I & D & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & D \end{bmatrix}, \quad D = \begin{bmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & \cdots & 0 \\ 0 & -1 & 4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 4 \end{bmatrix}$$

We perform experiments in three main categories: behavior of iterative solver with fault injection, performance of fault tolerance schemes with no fault injection and performance of fault tolerance schemes with fault injection. For those experiments that perform fault injection, we inject faults into the kernels that perform basic linear algebra operations in the iterative solvers, i.e., those kernels that find the dot product of two vectors, scale a vector by a constant, add two vectors, or perform matrix-vector multiplication. We set the active duration of a fault to $500\mu s$ and the fault rate to $1s$. We normalize the time taken for any run with the average time taken for the solver with no fault tolerance scheme. We treat any run with a result x that has $||b - Ax||_2/||b||_2 > 10^{-3}$ as erroneous. For experiments investigating the behavior of an iterative solver with fault injection, we set n to 65536 and execute about a few hundred runs for each bit (32 in total) that will be flipped in the event of a fault being activated. For experiments evaluating the performance of fault tolerance method with no fault injection, we use $n = 4096, 16384, 65536$ and 262144, and perform runs for each fault tolerance scheme at each value of n. Finally, for experiments investigating the performance of fault tolerance schemes with fault injection, we set n to 65536 and for each fault tolerance method, we perform a few hundred runs with fault injection. For each run, we randomly choose a bit to flip, uniformly distributed from bit number 31 to 16, in the event a fault is activated. In each fault injection experiment, we pick one random SP of a randomly chosen SM.

6.8 Results

6.8.1 Behavior of Iterative Solvers with Fault Injection

Figure 6.1 shows the behavior of the base iterative solvers in the presence of faults. One will notice that errors in the lower order bits of the mantissa cause no problems for both solvers – each solver converges to the correct solution in the majority of such cases. This can be expected of iterative solvers in general. However, when errors occur in the sign and exponent bits, as well as in the more significant bits of the mantissa, one can expect the solver to converge to an incorrect solution or even fail to converge. Overall, the results are not surprising, considering that in the case of the more significant bits of the exponent, a bit flip effectively changes the magnitude of the result by a large amount. Hence, iterative solvers, despite their inherent resilience, still require some fault tolerance.

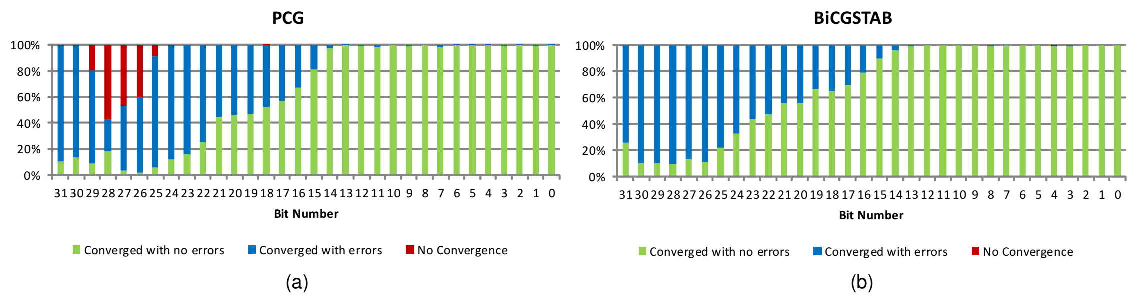
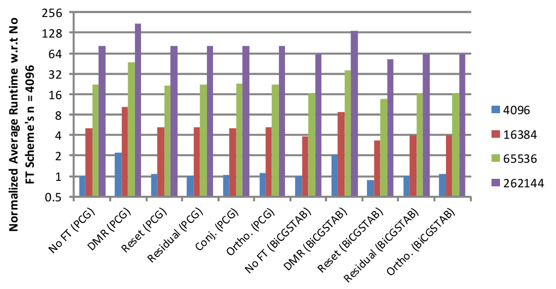


Figure 6.1 Behavior of iterative solvers in the presence of faults.

6.8.2 Performance of fault tolerance Schemes in the Absence of Faults

Figure 6.2 shows the normalized runtimes of the various error checking methods for both solvers. Figure 6.2(a) shows the average runtime for each method, all normalized with respect to the no fault tolerance scheme (No FT) for $n = 4096$. Here, we would like to point out that the average runtime for the No FT scheme at $n = 4096$ is 0.041s for PCG and 0.051s for BiCGSTAB. Figure 6.2(b) is similar, but shows the average runtime for each method normalized with respect to that *same* method's average runtime for $n = 4096$. Both of these graphs illustrate the growth of the various fault tolerance schemes as n increases. From Figure 6.2(b), one can see that the computational time for each method grows at a rate faster than $O(n)$ but less than $O(n^2)$. In addition,



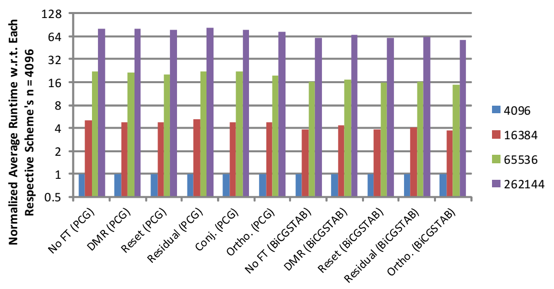


Figure 6.2 Growth of FT methods with no faults injected for various n (average runtime for No FT, $n = 4096$, is 0.048s).

from Figure 6.2(a), one can observe that the growth of computational time for the Orthogonal, Residual and Conjugate methods are very close to the growth for the baseline solver with no fault tolerance for both solvers, which indicates that the overhead of error checking for these methods is low compared to the complexity of the solver itself. Likewise, DMR has a similar growth, but its overhead is always around twice as much as that of the baseline, for each n. In the case of Reset, the growth of computational time is very close to the growth for the baseline solver with no fault tolerance for both PCG and BiCGSTAB; however, our choice of value for R causes Reset to actually converge *faster* than No FT for BiCGSTAB. It appears that the reset strategy accelerates the convergence of the solution, requiring less iterations needed for convergence compared to No FT. While this might seem to indicate that Reset has low overhead, we would like to point out that in general, the performance of Reset is sensitive with respect to R and there is no simple way to determine a good value of R. This limits the usefulness of Reset.

Figure 6.3 shows the average of the runtimes normalized with respect to the runtimes for the base iterative solvers with no fault tolerance (No FT), at the same value of n. For example, the

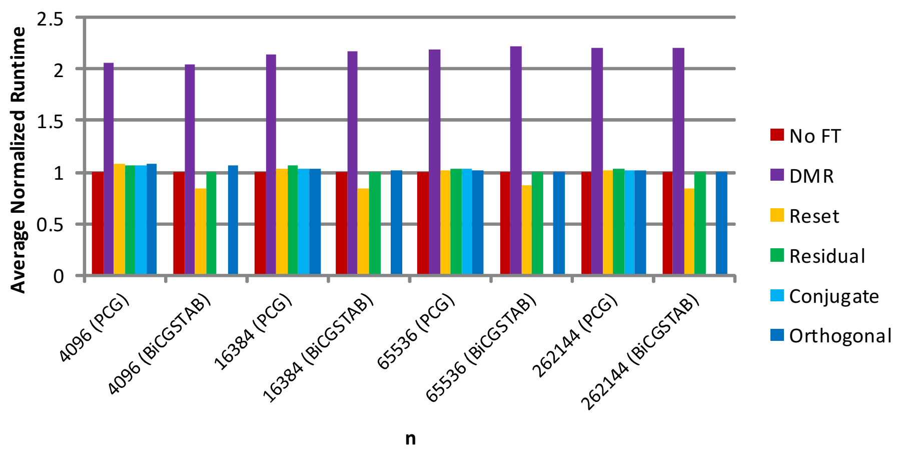


Figure 6.3 Overhead of error detection with no faults injected.

runtime of Orthogonal for $n = 16384$ is normalized with respect to the runtime of No FT for $n = 16384$. The purpose of this graph is to compare the overhead of the various FT methods (without recovery) at each value of n. One can see that the Orthogonal, Residual, Reset and Conjugate checking schemes all have low overhead, as seen from the fact that their overhead does not increase appreciably as n increases, for both the PCG and BiCGSTAB solvers. The overhead of the DMR scheme has an average runtime that is roughly twice as long as that of the respective base iterative solver, but the overhead remains relatively stable across n for both solvers.

6.8.3 Performance of fault tolerance Schemes with Fault Injection

Figure 6.4(a) shows the various outcomes of the fault tolerance schemes when faults are injected for $n = 65536$, while figure 6.4(b) shows the runtime of these schemes, normalized to the runtime of the base iterative solver with no fault tolerance, only for cases where the errors were detected and corrected, with $n = 65536$ (for Reset, we use the runtime for the cases where the error was not detected, but the solution is correct).

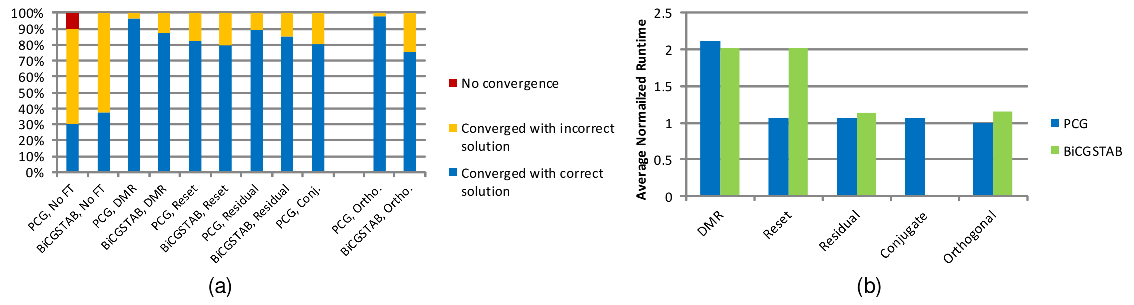


Figure 6.4 Outcomes and Performance of FT methods with faults injected, with $n = 65536$ (Conjugate method only for PCG)

From figure 6.4(a), one can see that the Orthogonal, Conjugate, Reset and Residue schemes have good error coverage, with most outcomes having a correct final result. However, for these schemes, there are a small number of cases where the result is incorrect. The DMR scheme performs well for both solvers, with many cases where the solution is correct; however, this comes at a cost of high overhead. In addition, the DMR, Reset, Orthogonal and Residual schemes all have better error coverage for PCG compared to BiCGSTAB, with Orthogonal performing particularly well for PCG. The error coverage of the fault tolerance schemes are all significantly better than the No FT scheme, which has a large number of cases with an incorrect solution, and even a number of cases in PCG where the solution failed to converge.

As seen in figure 6.4(b), the overhead of the Orthogonal, Conjugate and Residue schemes go up slightly when recovery was required. Conversely, the overhead of DMR remains high – more than double the runtime of the respective base solver. In the case of Reset, the overhead goes up slightly for PCG while the overhead goes up significantly to about 2 times the base runtime for BiCGSTAB.

In Chapter 7, we will consider the Lanczos eigensolver, which has several practical uses, as well as FT strategies for this linear algebraic application.

Chapter 7

Fault Tolerance in the Lanczos Eigensolver

7.1 Overview of Lanczos

A scalar λ and a non-zero column vector x are considered to be an eigenvalue and an eigenvector, respectively, of the matrix A if they satisfy the following relation:

$$Ax = \lambda x$$

The Lanczos eigensolver [67] is a popular iterative method for approximating a few maximal eigenvalues of a real symmetric matrix, particularly if the matrix is large and sparse. It is commonly used when computing the singular value decomposition (SVD) of a large, sparse symmetric matrix. Applications for the Lanczos eigensolver include latent semantic indexing, which requires the SVD of a large and sparse document-term matrix to find the terms that are the most relevant to some particular documents, as well as numerical weather prediction, where it is used to estimate the most quickly (linearly) growing perturbations to the central weather prediction model.

The Lanczos method offers several advantages. Firstly, it does not generate *fill-in* to the original sparse matrix, unlike the Householder tridiagonalization method [52]. This is critical because fill-in destroys the sparsity of the original matrix. Secondly, it is highly parallel and has only $O(n)$

points of synchronization (these are at the points where the values of α and β are computed). This makes it suitable for execution on GPUs.

7.1.1 Basic Theory

It is important to note that the Lanczos method does not directly calculate the eigenvalues of an $n \times n$ matrix A. Rather, it computes an $n \times n$ tridiagonal matrix T. A set of orthonormal vectors v_1, v_2, ..., v_n are also computed. These are known as *Lanczos vectors*, and they comprise the columns of the $n \times n$ matrix V. By construction, T is orthogonally similar to A, such that $AV - VT = 0$. This implies that the eigenvalues of T are also the eigenvalues of A. Moreover, if (λ, y) is an eigenpair of T, then (λ, x) is an eigenpair of A, where $x = Vy$. The proof is as follows.

Lemma 3. *Let T be a matrix that is orthogonally similar to a symmetric matrix A, such that $AV - VT = 0$. If (λ, y) is an eigenpair of T, then (λ, x) is an eigenpair of A, where $x = Vy$.*

Proof. Since T is orthogonally similar to A, $AV - VT = 0$ and consequently, $T = V^T AV$. Suppose (λ, y) is an eigenpair of T, i.e. $Ty = \lambda y$, with $x = Vy$. Then we have

$$\begin{aligned} Ty &= V^T AVy \\ &= V^T Ax \end{aligned}$$

So $V^T Ax = \lambda y$, and we have

$$\begin{aligned} Ax &= \lambda Vy \\ &= \lambda x \end{aligned}$$

Therefore, if (λ, y) is an eigenpair of T, then (λ, x) is an eigenpair of A.

□

Note that the above proof assumes that we are able to compute all n Lanczos vectors in exact arithmetic, such that V is an orthogonal matrix.

7.1.2 Practical Implementation of Lanczos

A practical implementation of Lanczos does not compute all n Lanczos vectors. Rather, it orthogonally projects A onto a k-dimensional Krylov subspace, where k is much smaller than n, to obtain a $k \times k$ tridiagonal matrix T_k.

$$T_k = \begin{bmatrix} \alpha_1 & \beta_2 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \beta_3 & \cdots & 0 \\ 0 & \beta_3 & \alpha_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \alpha_k \end{bmatrix}$$

The scalar k is typically chosen to be an integer multiple of the number of eigenvalues desired. The Lanczos vectors v_1, v_2, ..., v_k which form an orthonormal basis of the Krylov subspace are computed, which form the columns of the $n \times k$ matrix V_k. Because the Lanczos vectors are orthonormal, the non-zero elements of T_k, that is, α_j and β_j (for $j = 1, 2, ..., k$), are easy to compute and are calculated iteratively during the execution of the Lanczos method, as shown in algorithm 13. The following relationship exists between A and T_k:

$$AV_k - V_kT_k = \beta_{k+1}v_{k+1}e_k^T$$

Here, e_k is a $k \times 1$ column vector whose elements are all zeroes except for the kth element, which is 1. It is useful to point out that T_k represents the orthogonal projection of A onto the Krylov subspace. This is true, because $V_k^T v_{k+1} = 0$ by construction, and therefore by pre-multiplying the above equation by V_k^T, we obtain $T_k = V_k^T A V_k$. In other words, T_k is orthogonally similar to A and so the eigenvalues and eigenvectors of T_k can be used to approximate those of A. This algorithm includes the local orthogonalization procedure, which is explained in more detail later.

Algorithm 13 Compute $k \times k$ tridiagonal matrix T_k from symmetric matrix $A \in \mathbf{R}^{n \times n}$

```
 1: procedure LANCZOS($A$)
 2:     choose a unit-norm vector $v_1$
 3:     for $j = 1 : k$ do
 4:         $u_{j+1} = Av_j$
 5:         if $j > 1$ then
 6:             $u_{j+1} = u_{j+1} - (v_{j-1}^T u_{j+1})v_{j-1}$ (local orthogonalization)
 7:         end if
 8:         $\alpha_j = v_j^T u_{j+1}$ (local orthogonalization)
 9:         $u_{j+1} = u_{j+1} - \alpha_j v_j$ (local orthogonalization)
10:         $\beta_{j+1} = \|u_{j+1}\|_2$
11:         if $\beta_{j+1} = 0$ then
12:             stop
13:         end if
14:         $v_{j+1} = \frac{u_{j+1}}{\beta_{j+1}}$
15:     end for
16:     return $T_k, V_k$
17: end procedure
```

Once T_k is obtained, the eigenvalues and eigenvectors of T_k are then directly calculated using the implicit QR method [52], and additional computation is done to determine whether these eigenpairs are good approximations of those of A. This is performed on the CPU, and is not shown in algorithm 13. Because T_k represents the orthogonal projection of A onto the Krylov subspace, the eigenpairs of T_k can be taken as good approximations of the eigenpairs of A. Suppose (λ_i, y_i) is an eigenpair of T_k. Then the scalar λ_i and the vector $x_i = V_k y_i$, are approximations to an eigenpair of A.

It is expected that only a few of the k eigenpairs of T_k are good approximations, due to finite-precision arithmetic. The eigensolver determines which of these eigenpairs are indeed good approximations of those of A by computing the residual norm for the eigenpair, which is relatively easy to compute and is done on the CPU:

$$\begin{aligned}\|Ax_i - \lambda_i x_i\|_2 &= \|AV_k y_i - \lambda_i V_k y_i\|_2 \\ &= \|(AV_k - V_k T_k) y_i\|_2 \\ &= \beta_{k+1} |e_k^T y_i|\end{aligned}$$

7.1.3 Orthogonalization Methods for Lanczos

In exact arithmetic, the Lanczos vectors are always mutually orthonormal, and the extremal eigenvalues of T_k turn out to be good approximations to the extremal eigenvalues of A. However, the Lanczos method, when implemented in finite-precision arithmetic, behaves differently from what would be expected in algorithm 13. In addition to other anomalous behavior, the eigenvalues tend to converge to incorrect values, leading to many spurious approximations to the eigenvalues of A. This undesirable behavior appears at the same time when the Lanczos vectors begin to lose mutual orthogonality due to rounding errors caused by finite-precision arithmetic. This issue can be rectified by explicitly reorthogonalizing the Lanczos vectors using an orthogonalization method. These methods can take up a significant portion of the total computation time.

There are several orthogonalization methods available, which require a tradeoff between robustness and performance. Here, we will briefly review two particular methods, the full orthogonalization and the local orthogonalization methods.

7.1.3.1 Full Orthogonalization

The most robust method is full orthogonalization. In this method, during each iteration of the Lanczos method, the newly-computed vector u_{j+1} is orthogonalized with respect to all of the previously computed Lanczos vectors. This maintains the orthogonality of the Lanczos vectors to full machine precision. Unfortunately, this incurs a high overhead, which increases as more iterations of the Lanczos method is carried out.

7.1.3.2 Local Orthogonalization

A more efficient method for addressing the loss of orthogonality is local orthogonalization. In this method, during each iteration of the Lanczos method, the newly-computed vector u_{j+1} is orthogonalized only with respect to the vectors v_j and v_{j-1}, using the modified Gram-Schmidt method. This is a less robust method, but it has significantly lower overhead. An interesting point to note is that the Lanczos method with local orthogonalization is as robust as the Lanczos method with full orthogonalization *until* an eigenvalue is close to convergence. This suggests that explicitly restarting the Lanczos method after an eigenvalue of T_k has converged can be a good strategy. This is briefly described in the next sub-section.

7.1.4 Explicit Restart Strategy for Lanczos

Even with a sound orthogonalization method, the Lanczos method can still struggle to obtain enough eigenvalues that converge to the desired accuracy. Hence, the Lanczos method is typically restarted several times, performing k iterations (in general) each time with a successively "better" initial vector. The initial vector for the next instance of the Lanczos method is the Lanczos vector associated with the first desired, non-converged eigenvalue.

It is necessary to keep track of eigenpairs that have already converged, and to perform some form of deflation. This requires a *locking* technique, in which Lanczos vectors associated with converged eigenpairs, i.e. eigenpairs found to be good approximations of those of A, are not modified in restarted runs of Lanczos.

7.2 Fault Tolerance Method

7.2.1 Orthogonal Invariant Checking (orthoFT)

We aim to protect the part of our implementation that runs on the GPU (i.e. the Lanczos method itself) with our FT method. We exploit the fact that the during each instance of the Lanczos method, the Lanczos vectors v_1, v_2, ..., v_k are supposed to be mutually orthogonal. Even with explicit orthogonalization of the Lanczos vectors in the Lanczos method, there is a strong possibility that an error might occur during the orthogonalization process, since it takes up a substantial portion of the execution time.

At the end of each iteration j of the Lanczos method, we check whether the newly computed vector, v_{j+1}, is orthogonal to the previous vector, v_j, by taking the dot product of the two vectors. If the program is working normally, then this should be a scalar value close to zero. In other words, we check whether $|v_{j+1}^T v_j|$ exceeds a specified threshold, τ, which is a positive value with a magnitude close to zero. If this is the case, then an error has been detected. We determine the value of τ empirically, which we will describe in more detail later. The overhead of doing the checks per iteration of the Lanczos method is $O(n)$.

When the orthogonal check detects an error, we perform recovery by redoing that iteration in which the error was detected, and recomputing all values calculated during that iteration. After recovery, we perform another orthogonal check to ensure that the errors have been corrected. If not, we repeat the recovery procedure, and then do a final orthogonal check. At this point, if errors are still detected, we allow the eigensolver to proceed (with the output of a warning message), in the hope that the eigensolver will still be able to eventually converge to a few good eigenpairs. Algorithm 15 shows the full eigensolver augmented with the orthogonal checking method. In subsequent sections, we will refer to this method as the "orthoFT" method, and the baseline version with no fault tolerance as "noFT".

Note that this check would also work for an implementation that uses the full orthogonalization method – in this case, each time the current Lanczos vector is orthogonalized with respect to one of the previous vectors, one can perform the orthogonal check to check for errors.

7.3 Implementation Details

Our Lanczos eigensolver program uses SLEPc [68], which is a comprehensive library for solving eigenvalue problems. It contains several eigensolvers, mainly for sparse matrices, including the Lanczos, Arnoldi [69] and locally optimal block preconditioned conjugate gradient (LOBPCG) [70] methods. The SLEPc library inherits a lot of functionality from PETSc [71], which is an older library that contains many routines and functions [72]. These include basic sparse linear algebraic operations (for example sparse matrix-vector multiplication), as well as higher-level routines (such as direct linear solvers). Our implementation uses the CRS sparse matrix format (in PETSc, this is referred to as the *seqaijcusparse* format). We direct the interested reader to the informative technical report by Hernandez et al [73] for more details on the Lanczos eigensolver.

In our implementation of the Lanczos eigensolver, we request for 1 to 10 eigenvalues depending on the input matrix, set $k = 30$ and the convergence tolerance to 0.05. We allow for a maximum of n restarts of the Lanczos method, where n is the width of the matrix. We use the local orthogonalization method. We pick the initial unit-norm vector v_1 randomly (this is the default SLEPc option). Algorithm 14 shows the full implementation of the Lanczos eigensolver with restart and local orthogonalization. We would like to point out that in our implementation, the Lanczos method itself is executed on the GPU, while the computation of the eigenpairs of T_k, the computation of the residual norm estimates and the checking of false eigenvalues are all executed on the CPU. Thus, it is sufficient for us to explicitly protect just the Lanczos method, since that is executed on the GPU.

7.4 Experimental Methodology

We ran all experiments using a GPU compute cluster, the same as the setup we used for our experiments in Chapter 5. It contains 14 compute nodes, 12 of which contain GTX 1080 GPUs. Each experiment is run on a single node using one of the four GTX 1080 cards. Table 7.1 shows the specifications of one such node.

We ran experiments for the following four cases: noFT method with no fault injection, noFT method with fault injection, orthoFT with no fault injection and orthoFT with fault injection. We

Table 7.1 Specifications of a compute node

Component	Quantity
Intel Xeon E5-2650 v3 CPU @ 2.30GHz	2
128GB DDR4 ECC RAM	-
NVIDIA GTX 1080	4

used 20 sparse, real symmetric matrices from the Florida Sparse Matrix Collection [59] and measure the time it takes for the Lanczos method to run on the GPU. We used double-precision floating point numbers.

We empirically determine the value of the orthogonal checking threshold, τ, by running the eigensolver for each matrix, for a specified requested number of eigenvalues, without fault injection. We then identify the value of $|v_{j+1}^T v_j|$ with the greatest magnitude that occurred for that particular run. We set τ to this value. Table 7.2 lists the number of eigenvalues requested and the value of τ for each matrix.

Our fault model is similar to the one used in Chapter 5. We consider transient hardware faults that persist for a certain duration, which affect the processing elements of the GPU and lead to SDC. We only consider faults that occur in the SP cores during execution, but allow for the possibility that corrupted results can be written to memory by threads running on faulty cores. A SP core of one particular SM is selected for fault injection. Each iteration, there is some probability that a fault will be injected for that iteration, based on a predetermined threshold. Threads performing the basic linear algebra operations in the portions of the eigensolver executed on the GPU, that happen to be running on the chosen SP when the fault is injected for that iteration, will have their results corrupted. This is done by flipping a bit in the 64-bit double-precision floating point result of each of those threads. This simulates a fault with an active duration in the order of milliseconds.

For each experimental run that involves fault injection, we inject at most one fault, according to our above fault model. We inject faults only in the portions of the program that are executed on the GPU. Each run, we choose one particular SM and SP core of the GPU to inject faults into. We ran thousands of experimental runs to ensure that the results are statistically significant.

It is possible that activated faults might result in errors that cause the Lanczos eigensolver to converge to values which are valid eigenvalues of A but are *different* from values that would have been obtained had there not been any errors in the first place. For the purposes of our work, we will treat such outcomes, in which different approximated eigenvalues of A are computed by the eigensolver, as having "no errors". However, users may not find such outcomes to be acceptable.

7.5 Results

Figure 7.1 shows the overall runtime of the Lanczos eigensolver, along with with the runtimes for the various components of the eigensolver. It can be observed that the Lanczos portion (which runs on the GPU) takes up a significant portion of the overall runtime, which is more than 50%. Furthermore, the orthogonalization phase dominates the time taken in the Lanczos portion itself. This indicates that it is worthwhile to provide fault tolerance for the portion of the Lanczos eigensolver that runs on the GPU.

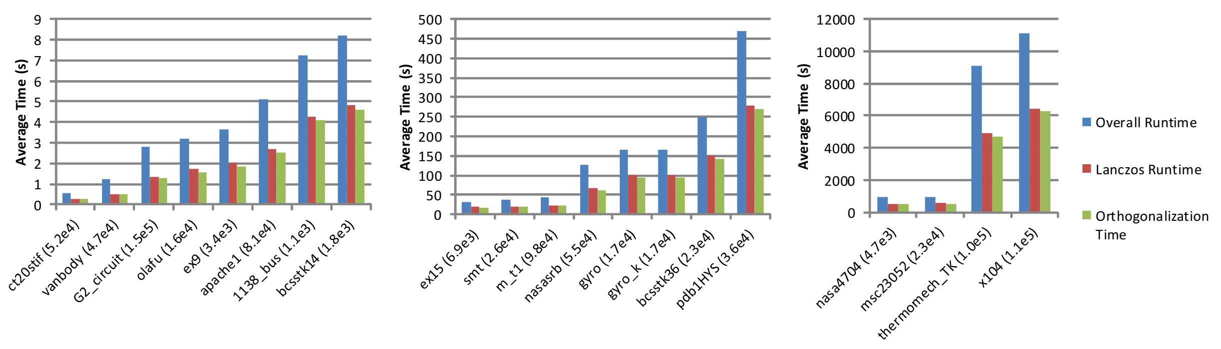


Figure 7.1 Overall runtime of Lanczos eigensolver with runtime of various components (with no fault injection)

Figure 7.2 compares the Lanczos runtime (this is basically the GPU time) for the noFT method versus the orthoFT method that uses the orthogonal invariant checking strategy, with no fault injection. This measures the overhead of error detection. As can be seen from figure 7.2, the detection overhead of the orthoFT method is very low, as the Lanczos runtime for the orthoFT method is barely higher than that of the noFT method.

Figure 7.3 again compares the Lanczos runtime for the noFT method versus the orthoFT method, this time with fault injection *only* for the orthoFT scheme only (i.e. the runtime values for noFT are the same as those of the experiments with no fault injection). This measures the overhead of error detection together with error recovery. As can be seen from this figure, the detection and recovery overhead of the orthoFT method is still low. The error coverage of the orthoFT scheme is good, with errors detected in over 90% of the runs. For the vast majority of cases, even in the presence of errors, the eigensolver with the orthoFT scheme was able to find the requested number of eigenvalues, requiring less than n restarts.

It is worth pointing out that the eigensolver (i.e. basically the noFT scheme) already has some built-in fault tolerance, because at the end of each Lanczos phase, when the computation of T_k is

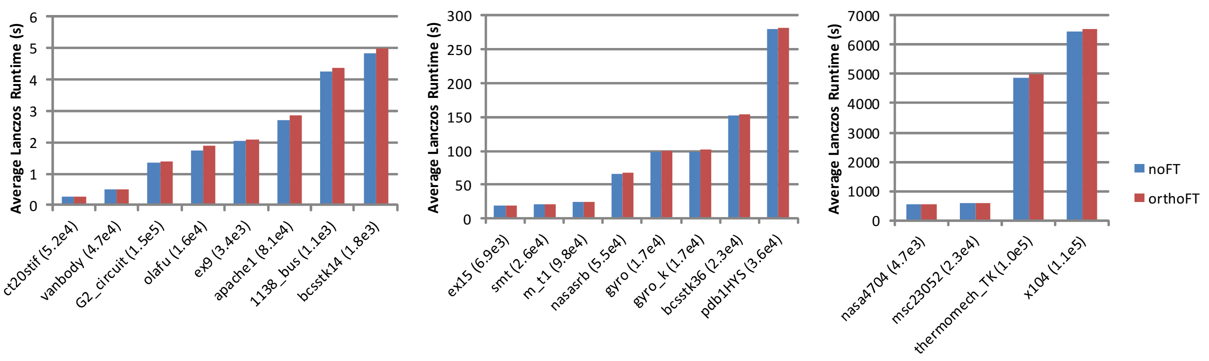


Figure 7.2 Lanczos runtime of FT schemes (with no fault injection)

complete, the eigensolver determines which of eigenpairs of T_k are good approximations of those of A by computing the residual norm for the eigenpair. This also serves as a correctness check for the eigensolver. Thus, one might wonder if there is a need for the orthoFT scheme in the first place. Figure 7.4 serves to illustrate why an explicit FT method, like the orthoFT scheme, is still necessary.

As one can observe from figure 7.4, when faults are injected into the noFT scheme (i.e. the base eigensolver), the runtime of the noFT scheme generally takes substantially longer than the orthoFT scheme (including recovery time). Interestingly, in some rare cases, for example *nasasrb*, the presence of faults actually causes the eigensolver to converge faster. For some matrices, like *smt*, the runtime of noFT is dramatically longer than the runtime for orthoFT. For many of these cases, under fault injection, the noFT scheme had to perform many more explicit restarts of Lanczos in order to obtain the requested number of eigenvalues, or failed to find enough eigenvalues before reaching the limit of n restarts. This demonstrates that our orthoFT scheme is still necessary to ensure error-free operation of the eigensolver.

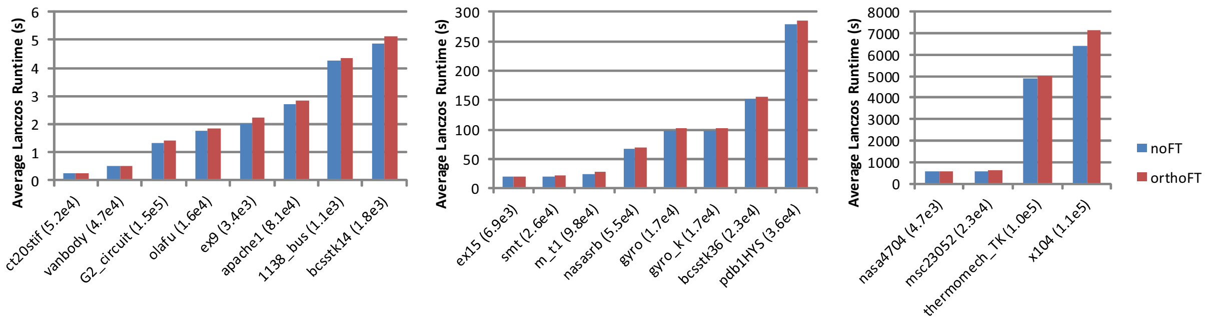


Figure 7.3 Lanczos runtime of FT schemes (with fault injection into orthoFT scheme)

Algorithm 14 Compute s eigenvalues and eigenvectors from symmetric matrix $A \in \mathbf{R}^{n \times n}$

```
1: procedure LanczosEigensolver($A$)
2:     choose a unit-norm vector $v_1$
3:     $V_k = [v_1]$
4:     $m = 0$
5:     while no. of converged eigenpairs $<$ $s$ , and max no. of restarts has not been reached do
6:         perform Lanczos method:
7:         for $j = (m+1) : k$ do
8:             $u_{j+1} = Av_j$
9:             if $j > 1$ then
10:                $u_{j+1} = u_{j+1} - (v_{j-1}^T u_{j+1}) v_{j-1}$ (local orthogonalization)
11:            end if
12:            $\alpha_j = v_j^T u_{j+1}$ (local orthogonalization)
13:            $u_{j+1} = u_{j+1} - \alpha_j v_j$ (local orthogonalization)
14:            $\beta_{j+1} = \|u_{j+1}\|_2$
15:            if $\beta_{j+1} = 0$ then
16:                stop
17:            end if
18:            $v_{j+1} = \frac{u_{j+1}}{\beta_{j+1}}$
19:        end for
20:        compute eigenpairs of $T_k$ , $T_k y_i = \lambda y_i$ using implicit QR method
21:        compute residual norm estimates of eigenpairs, $\gamma_i = \beta_{k+1} |e_k^T y_i|$
22:        check for false eigenvalues
23:        lock converged eigenpairs, update value of $m$
24:        $V_k = V_k Y$
25:    end while
26:    return up to $s$ converged eigenpairs $(\lambda, x)$ of $A$
27: end procedure
```

Algorithm 15 Compute s eigenvalues and eigenvectors from symmetric matrix $A \in \mathbf{R}^{n\times n}$

```
1: procedure ORTHOFTLANCZOSEIGENSOLVER($A$)
2:     choose a unit-norm vector $v_1$
3:     $V_k = [v_1]$
4:     $m = 0$
5:     while no. of converged eigenpairs $<$ $s$ , and max no. of restarts has not been reached do
6:         perform Lanczos method:
7:         for $j = (m+1) : k$ do
8:             $u_{j+1} = Av_j$
9:             if $j > 1$ then
10:                $u_{j+1} = u_{j+1} - (v_{j-1}^T u_{j+1})v_{j-1}$ (local orthogonalization)
11:            end if
12:            $\alpha_j = v_j^T u_{j+1}$ (local orthogonalization)
13:            $u_{j+1} = u_{j+1} - \alpha_j v_j$ (local orthogonalization)
14:            $\beta_{j+1} = \|u_{j+1}\|_2$
15:            if $\beta_{j+1} = 0$ then
16:                stop
17:            end if
18:            $v_{j+1} = \frac{u_{j+1}}{\beta_{j+1}}$
19:            check whether $|v_{j+1}^T v_j| > \tau$ (orthogonal check)
20:            if errors detected (i.e. $v_{j+1}$ is not orthogonal to $v_j$) then
21:                redo iteration $j$
22:            end if
23:        end for
24:        compute eigenpairs of $T_k$ , $T_k y_i = \lambda y_i$ using implicit QR method
25:        compute residual norm estimates of eigenpairs, $\gamma_i = \beta_{k+1}|e_k^T y_i|$
26:        check for false eigenvalues
27:        lock converged eigenpairs, update value of $m$
28:        $V_k = V_k Y$
29:    end while
30:    return up to $s$ converged eigenpairs $(\lambda, x)$ of $A$
31: end procedure
```

Table 7.2 Number of eigenvalues requested and τ for each matrix

Matrix	**Eigenvalues**	τ	**Matrix**	**Eigenvalues**	τ
1138_bus	9	10^{-6}	nasa4704	5	10^{-6}
apache1	10	10^{-6}	nasasrb	1	3.414×10^{-3}
bcsstk14	5	2×10^{-6}	ct20stif	10	5.8×10^{-5}
bcsstk36	3	10^{-6}	olafu	1	3.36×10^{-4}
ex9	1	10^{-6}	pdb1HYS	10	10^{-8}
ex15	1	2×10^{-6}	msc23052	5	10^{-6}
gyro_k	5	10^{-6}	smt	4	10^{-6}
gyro	5	10^{-6}	thermomech_TK	10	10^{-6}
G2_circuit	10	10^{-8}	vanbody	6	2.8×10^{-5}
m_t1	5	2.52×10^{-4}	x104	5	10^{-6}

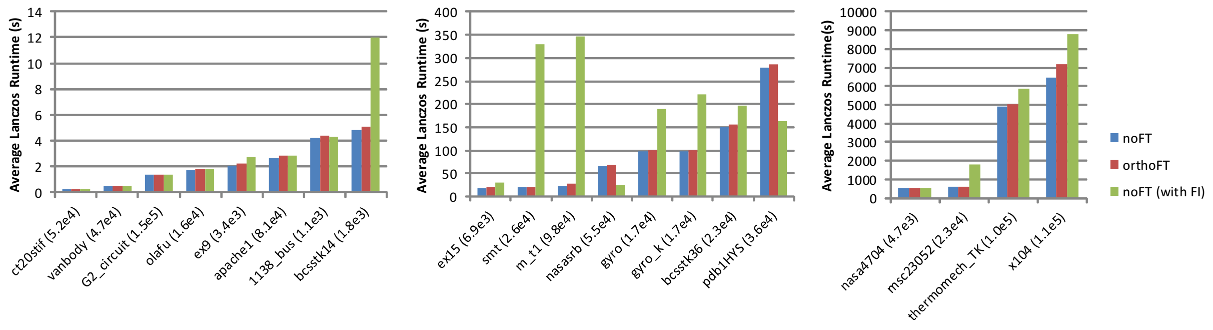


Figure 7.4 Lanczos runtime of FT schemes (with fault injection into noFT and orthoFT schemes)

Chapter 8

Conclusion

8.1 Summary

Linear algebra is important in many computing applications, spanning areas as diverse as artificial intelligence, bioinformatics, computer architecture and sociology.

In recent years, GPUs have become a popular platform for scientific computing applications and are increasingly being used as the main computational units in supercomputers. This trend is expected to continue as the number of computations required by scientific applications reach petascale and even exascale range. As the minimum feature size of transistors decreases, GPUs become more vulnerable to transient faults caused by events such as alpha particle strikes, power fluctuations and electronic noise. One can expect the soft error rate of contemporary GPUs to be high. In addition, the likelihood of a fault increases as more and more GPU computing nodes are used in parallel to meet the increasingly demanding computational requirements of scientific applications, and there are concerns that exascale systems will suffer from very high fault rates. This necessitates the use of FT techniques.

Hardware FT techniques are typically expensive and consume valuable space on a chip. In the case of GPUs, this cost can be unacceptable. Software FT techniques, including ABFT, are an important and necessary approach for providing reliable computation on GPUs without sacrificing valuable chip space. Therefore, scientific computing applications that run on GPUs should be augmented with software-based FT mechanisms.

Several linear algebra software collections are already in existence. However, to the best of our knowledge, there has been no comprehensive fault tolerant linear algebra software collection that can be utilized to provide low-overhead fault tolerance for critical applications that make significant use of certain linear algebra routines.

In this dissertation, we develop low-overhead FT techniques for several commonly-used linear algebraic applications that run on GPUs, by carefully analyzing the algorithms underlying these applications. These techniques exploit the invariant properties of the algorithms used in these applications, and take advantage of the parallel execution model of GPUs to allow for low-overhead error detection.

We introduce and evaluate efficient error checking schemes for three widely-used matrix factorization techniques: block Householder QR factorization, supernodal left-looking Cholesky factorization, and right-looking LU factorization. In addition, we propose low-overhead invariant checking methods for the preconditioned conjugate gradient (PCG) and the biconjugate gradient stabilized (BiCGSTAB) iterative solvers, and introduce an efficient checking method for the Lanczos eigensolver. Lastly, we develop fault injection mechanisms for NVIDIA GPUs that allow for the simulation of transient, non-instantaneous faults.

8.2 Future Work

Our dissertation presents an extensive analysis of several popular linear algebraic applications. However, there are some opportunities for improvement to this work and this section serves to briefly discuss these possibilities.

8.2.1 Additional FT Linear Algebraic Applications

A natural extension to this work is to add more FT linear algebraic applications to our existing collection. Candidates include the Arnoldi eigensolver [69], the LOBPCG eigensolver [70], the symmetric QR algorithm and the singular value decomposition (SVD) algorithm [52].

8.2.2 Formal Characterization of FT Techniques

Roundoff errors are unavoidable in floating point computations because processors are limited by finite-precision arithmetic. This means that it can be challenging to pick suitable threshold values for our invariant checking methods to determine whether an error has occurred. We have already utilized empirical approaches to choose appropriate threshold values for our error checking methods. It might be possible to improve on this process by using a theoretical approach.

We would like to develop a good error model that accurately models the effects of faults. In our theoretical approach, we could investigate the effect of an error within some iteration of an algorithm, by corrupting an intermediate matrix, vector or scalar with δ (which can be a scalar, vector, or even a matrix). We can then follow its effect on the computation of subsequent matrices, vectors or values as the algorithm proceeds, until the error checking procedure is reached. We can subsequently repeat this process by corrupting other matrices, vectors and scalars used in the algorithm. What we would like to achieve is to identify the set of faults (i.e. error magnitude/norm) that is guaranteed to be detected by our checks, given a suitably chosen threshold. Alternatively, we can instead find the set of faults that will *not* be detected by our checks, or the set of faults that may or may not be detected by our checks. This could allow us to pick better threshold values for our error checking schemes.

8.3 Closing Remarks

In this dissertation, we evaluate our FT techniques using mainly sparse matrix data representations, on an NVIDIA GPU platform with the injection of transient, non-instantaneous faults. However, we wish to point out that our FT methods are actually independent of the data representation, the host platform, and the fault injection mechanism. In other words, our FT techniques will work similarly well on (say) versions of our applications that use dense matrix representations and are running on a CPU system that is affected by instantaneous faults.

Another issue that merits further discussion is our focus on protecting the processing elements of the GPU from soft errors, along with our assumption that original data read from memory is error-free. Here, we would like to mention that in most cases, our FT techniques will also detect errors that only occur during the writing of data to memory (for example, after the column update of elements during right-looking LU factorization). However, there are some exceptions. Our FT techniques will not detect errors in the case where the input matrix is somehow corrupted in memory before any actual computation is initiated, as this is equivalent to operating on a different input matrix, which would not violate any invariant properties. Also, in the case of applications involving matrix factorization, if some elements of the original input matrix become corrupted in memory during execution of the application, then our error checking methods for these applications will still be able to detect these errors, but recovery might not be possible.

Linear algebraic applications will always be a significant influence in modern life. With GPUs becoming a popular platform for such applications and the fact that these processors are increasingly vulnerable to transient faults, we are optimistic that the contributions of this dissertation will remain relevant for the foreseeable future.

LIST OF REFERENCES

[1] T. Madej, C. Lanczycki, D. Zhang, P. Thiessen, R. Geer, A. Marchler-Bauer, and S. Bryant, "MMDB and VAST+: Tracking structural similarities between macromolecular complexes," *Nucleic Acids Research*, vol. 42, p. 7, December 2013.

[2] T. Agerwala, "Exascale computing: The challenges and opportunities in the next decade," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, January 2010.

[3] M. A. Heroux, "Software challenges for extreme scale computing: Going from petascale to exascale systems," *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 437–439, 2009.

[4] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the impact of device and pipeline scaling on the soft error rate of processor elements," Tech. Rep. 2002-19, Dept. of Computer Sciences, The University of Texas at Austin, July 2003.

[5] NVIDIA, "NVIDIA GeForce GTX 1080." White Paper, 2016.

[6] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on GPUs in the field," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pp. 519–530, March 2016.

[7] T. A. Davis, W. W. Hager, and I. Duff, "SuiteSparse." `http://faculty.cse.tamu.edu/davis/suitesparse.html`. Accessed: 2018-05-02.

[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[9] "The NVIDIA cuBLAS library." `http://developer.nvidia.com/cublas`. Accessed: 2015-11-08.

[10] "The NVIDIA CUDA sparse matrix library." `http://developer.nvidia.com/cusparse`. Accessed: 2015-11-08.

[11] F. Loh, P. Ramanathan, and K. K. Saluja, "Transient fault resilient QR factorization on GPUs," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*, FTXS '15, pp. 63–70, 2015.

[12] F. Loh, K. K. Saluja, and P. Ramanathan, "Fault tolerant LU factorization on GPUs," Submitted for review.

[13] F. Loh, K. K. Saluja, and P. Ramanathan, "Fault tolerance through invariant checking for iterative solvers," in *Proceedings of the International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*, pp. 481–486, January 2016.

[14] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W. Hwu, "Performance analysis and tuning for general purpose graphics processing units (GPGPU)," *Synthesis Lectures on Computer Architecture*, 2012.

[15] NVIDIA, "NVIDIA's next generation CUDA compute architecture." White Paper, 2009.

[16] D. Sorin, *Fault Tolerant Computer Architecture*, vol. 4. Morgan and Claypool Publishers, January 2009.

[17] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin, "IBM experiments in soft fails in computer electronics," *IBM J. Res. Dev.*, vol. 40, pp. 3–18, January 1996.

[18] D. Gil-Tomas, J. Gracia-Moran, J. Baraza-Calvo, L. Saiz-Adalid, and P. Gil-Vicente, "Analyzing the impact of intermittent faults on microprocessors applying fault injection," *IEEE Design and Test of Computers*, vol. 29, pp. 66–73, December 2013.

[19] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies, Annals of Mathematics Studies*, vol. 34, pp. 43–98, 1956.

[20] N. Vaidya and D. Pradhan, "Fault-tolerant design strategies for high reliability and safety," *IEEE Transactions on Computers*, vol. 42, pp. 1195–1206, October 1993.

[21] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight error detection for GPGPU," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 37–47, December 2012.

[22] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 972–986, October 1998.

[23] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, pp. 518–528, June 1984.

[24] J. Jou and J. A. Abraham, "Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures," *Proceedings of the IEEE*, vol. 74, pp. 732–741, May 1986.

[25] D. Hakkarinen, P. Wu, and Z. Chen, “Fail-stop failure algorithm-based fault tolerance for Cholesky decomposition,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 1323–1335, May 2015.

[26] J. Chen, X. Liang, and Z. Chen, “Online algorithm-based fault tolerance for Cholesky decomposition on heterogeneous systems with GPUs,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016.

[27] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, “Algorithm-based fault tolerance for dense matrix factorizations,” in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 225–234, 2012.

[28] T. Davies and Z. Chen, “Correcting soft errors online in LU factorization,” in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC ’13, pp. 167–178, 2013.

[29] P. Wu and Z. Chen, “FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC ’14, pp. 49–60, 2014.

[30] P. Du, P. Luszczek, S. Tomov, and J. Dongarra, “Soft error resilient QR factorization for hybrid system with GPGPU,” *Journal of Computational Science*, vol. 4, no. 6, pp. 457–464, 2013. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[31] M. Hoemmen and M. Heroux, “Fault-tolerant iterative methods via selective reliability,” *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, vol. 3, p. 9, 2011.

[32] J. Elliott, M. Hoemmen, and F. Mueller, “Evaluating the impact of SDC on the GMRES iterative solver,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1193–1202, May 2014.

[33] Z. Chen, “Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods,” in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 167–176, 2013.

[34] F. Oboril, M. B. Tahoori, V. Heuveline, D. Lukarski, and J.-P. Weiss, “Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers,” in *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 144–153, December 2011.

[35] G. Bronevetsky and B. de Supinski, “Soft error vulnerability of iterative linear algebra methods,” in *Proceedings of the International Conference on Supercomputing*, pp. 155–164, 2008.

[36] J. Sloan, R. Kumar, and G. Bronevetsky, “Algorithmic approaches to low overhead fault detection for sparse linear algebra,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, June 2012.

[37] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, “Characterizing the impact of soft errors on iterative methods in scientific computing,” in *Proceedings of the International Conference on Supercomputing*, (New York, NY, USA), pp. 152–161, ACM, 2011.

[38] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, “Fault tolerant preconditioned conjugate gradient for sparse linear system solution,” in *Proceedings of the International Conference on Supercomputing*, pp. 69–78, 2012.

[39] H.-J. Wunderlich, C. Braun, and S. Halder, “Efficacy and efficiency of algorithm-based fault-tolerance on GPUs,” in *Proceedings of the IEEE International On-Line Testing Symposium (IOLTS)*, pp. 240–243, July 2013.

[40] R. Shah, M. Choi, and B. Jang, “Workload-dependent relative fault sensitivity and error contribution factor of GPU onchip memory structures,” in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 271–278, July 2013.

[41] N. Maruyama, A. Nukada, and S. Matsuoka, “A high-performance fault-tolerant software framework for memory on commodity GPUs,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[42] W. Dweik, M. Abdel-Majeed, and M. Annavaram, “Warped-Shield: Tolerating hard faults in GPGPUs,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 431–442, June 2014.

[43] M. Abdel-Majeed, W. Dweik, H. Jeon, and M. Annavaram, “Warped-RE: Low-cost error detection and correction in GPUs,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 331–342, June 2015.

[44] D. Palframan, N. Kim, and M. Lipasti, “Precision-aware soft error protection for GPUs,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pp. 49–59, February 2014.

[45] C. Braun, S. Halder, and H.-J. Wunderlich, “A-ABFT: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 443–454, 2014.

[46] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, “HAUBERK: Lightweight silent data corruption error detectors for GPGPU,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[47] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, “GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[48] S. Tselonis, V. Dimitsas, and D. Gizopoulos, “The functional and performance tolerance of GPUs to permanent faults in registers,” in *Proceedings of the IEEE International On-Line Testing Symposium (IOLTS)*, pp. 236–239, July 2013.

[49] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, “Impact of GPUs parallelism management on safety-critical and HPC applications reliability,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 455–466, 2014.

[50] N. Shanbhag, R. Abdallah, R. Kumar, and D. Jones, “Stochastic computation,” in *Proceedings of the 47th Design Automation Conference (DAC)*, (New York, NY, USA), pp. 859–864, ACM, 2010.

[51] E. Kim and N. Shanbhag, “Energy-efficient accelerator architecture for stereo image matching using approximate computing and statistical error compensation,” in *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 55–59, December 2014.

[52] G. H. Golub and C. F. van Loan, *Matrix Computations*. Baltimore, MD: Johns Hopkins University Press, 3rd ed., 1996.

[53] C. H. Bischof and C. F. van Loan, *The WY Representation for Products of Householder Matrices*. Ithaca, NY, USA: Cornell University, 1985.

[54] A. Kerr, D. Campbell, and M. Richards, “QR decomposition on GPUs,” in *Proceedings of the 2nd Workshop on General Purpose Processing on GPUs*, GPGPU-2, (NY, USA), pp. 71–78, 2009.

[55] M. Dimitrov, M. Mantor, and H. Zhou, “Understanding software approaches for GPGPU reliability,” in *Proceedings of the 2nd Workshop on General Purpose Processing on GPUs*, GPGPU-2, (NY, USA), pp. 94–104, 2009.

[56] T. A. Davis, S. Rajamanickam, and W. Sid-Lakhdar, “A survey of direct methods for sparse linear systems,” *Acta Numerica*, vol. 25, pp. 383–566, May 2016.

[57] E. G. Ng and B. W. Peyton, “Block sparse Cholesky algorithms on advanced uniprocessor computers,” *SIAM Journal on Scientific Computing*, vol. 14, no. 5, pp. 1034–1056, 1993.

[58] Y. Chen, T. A. Davis, W. Hager, and S. Rajamanickam, “Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate,” *ACM Transactions on Mathematical Software*, vol. 35, pp. 22:1–22:14, October 2008.

[59] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software*, vol. 38, pp. 1:1–1:25, December 2011.

[60] A. George and E. Ng, “Symbolic factorization for sparse Gaussian elimination with partial pivoting,” *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. 877–898, 1987.

[61] P. Amestoy, T. A. Davis, and I. Duff, “Algorithm 837: AMD, an approximate minimum degree ordering algorithm,” *ACM Trans. Math. Softw.*, vol. 30, pp. 381–388, September 2004.

[62] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[63] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, J. Chen, and Z. Chen, “Towards practical algorithm based fault tolerance in dense linear algebra,” in *Proceedings of the 25th International Symposium on High-performance Parallel and Distributed Computing*, HPDC ’16, pp. 31–42, 2016.

[64] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[65] M. Hestenes and E. Stiefel, “Methods of conjugate gradients for solving linear systems,” *Journal of Research of the National Bureau of Standards*, vol. 49, pp. 409–436, December 1952.

[66] H. A. van der Vorst, “Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 13, pp. 631–644, 1992.

[67] C. Lanczos, “An iteration method for the solution of the eigenvalue problem of linear differential and integral operators,” *Journal of Research of the National Bureau of Standards*, vol. 45, no. 4, pp. 255–282, 1950.

[68] V. Hernandez, J. E. Roman, and V. Vidal, “SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems,” *ACM Trans. Math. Software*, vol. 31, no. 3, pp. 351–362, 2005.

[69] W. Arnoldi, “The principle of minimized iterations in the solution of the matrix eigenvalue problem,” *Quarterly of Applied Mathematics*, vol. 9, pp. 17–29, 1951.

[70] A. Knyazev, “Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method,” *SIAM Journal on Scientific Computing*, vol. 23, no. 2, pp. 517–541, 2001.

[71] S. Balay, W. Gropp, L. McInnes, and B. Smith, “Efficient management of parallelism in object oriented numerical software libraries,” in *Modern Software Tools in Scientific Computing* (E. Arge, A. M. Bruaset, and H. P. Langtangen, eds.), pp. 163–202, Birkhäuser Press, 1997.

[72] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, D. May, L. McInnes, K. Rupp, P. Sanan, B.Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc users manual," Tech. Rep. ANL-95/11 - Revision 3.8, Argonne National Laboratory, 2017.

[73] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal, "Lanczos methods in SLEPc," Tech. Rep. STR-5, Universitat Politècnica de València, 2006. Available at http://slepc.upv.es.