Fault Tolerant Cholesky Factorization on GPUs

Felix Loh
University of Wisconsin-Madison
1415 Engineering Drive
Madison, WI 53706
floh@wisc.edu

Kewal K. Saluja
University of Wisconsin-Madison
1415 Engineering Drive
Madison, WI 53706
saluja@ece.wisc.edu

Parameswaran Ramanathan
University of Wisconsin-Madison
1415 Engineering Drive
Madison, WI 53706
parmesh@ece.wisc.edu

***Abstract*—Direct Cholesky-based solvers are typically used to solve large linear systems where the coefficient matrix is symmetric positive definite. These solvers offer faster performance in solving such linear systems, compared to other more general solvers such as LU and QR solvers. In recent days, graphics processing units (GPUs) have become a popular platform for scientific computing applications, and are increasingly being used as major computational units in supercomputers. However, GPUs are susceptible to transient faults caused by events such as alpha particle strikes and power fluctuations. As a result, the possibility of an error increases as more and more GPU computing nodes are used. In this paper, we introduce two efficient fault tolerance schemes for the Cholesky factorization method, and study their performance using a direct Cholesky solver in the presence of faults. We utilize a transient fault injection mechanism for NVIDIA GPUs and compare our schemes with a traditional checksum fault tolerance technique, and show that our proposed schemes have superior performance, good error coverage and low overhead.**

I. Introduction

Solutions of large linear systems (i.e. where a system represented by a linear equation $Ax = b$ is solved for the vector x) are usually obtained by one of two high-level strategies: the direct method and the iterative (indirect) method. Iterative solvers begin with an initial guess to the solution, x_0, and then iteratively make improvements to this solution vector by calculating a series of direction and residual vectors. Direct solvers, by contrast, first factorize the matrix A into a product of matrices with a well-defined non-zero structure (e.g. a lower and an upper triangular matrix), and then solve the factorized linear system by forward or backward substitution. If A is symmetric positive definite, then Cholesky factorization can be used to factorize A into the product LL^T, where L is a lower triangular matrix. This factorization is significantly faster than other matrix factorization methods, such as LU and QR [7].

In recent years, graphics processing units (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 approach exascale range [1]. As the minimum feature size of transistors decreases, GPUs are becoming more vulnerable to transient faults caused by events such as alpha particle strikes and power fluctuations. To make matters worse, the likelihood of an error increases as more GPU computing nodes are used to meet the increasingly demanding computational requirements of scientific applications. There are concerns that exascale systems will suffer from very high fault rates [9]. Therefore, scientific computing applications that run on GPUs must be protected by fault tolerant mechanisms. We focus on transient faults that affect the streaming processors (SPs) of the GPU.

Fault tolerant (FT) techniques for matrix factorization methods, such as Cholesky factorization, have been discussed in the literature. Most of these techniques are based on the pioneering work done by Huang et al [10] in providing fault tolerance for matrix-matrix multiplication through the use of checksums. For example, Hakkarinen et al [8] discuss and evaluate checksum techniques for the outer-product Cholesky and right-looking Cholesky methods on distributed memory systems for dense matrices. However, they do not evaluate the effectiveness of these techniques on GPUs. Chen et al [3] implement and evaluate a checksum FT scheme for Cholesky decomposition on MAGMA, a linear algebra library for heterogeneous CPU-GPU systems that is designed to work with dense matrices. Wu et al [18] propose using row and column checksums for LU factorization, to protect the factor matrices L, U and the trailing matrix portion during factorization.

Researchers have also proposed FT techniques for solvers that involve sparse linear systems. Loh et al [13] propose lightweight invariant checking for the preconditioned conjugate gradient (PCG) and biconjugate gradient stabilized (BiCGSTAB) iterative solvers. Shantharam et al [16] develop a checksum-based technique for protecting sparse matrices used in the PCG iterative solver.

Our work differs from the aforementioned papers. In our paper, we develop efficient FT schemes for the left-looking Cholesky factorization method, as well as its supernodal variant, using a different error checking strategy. The left-looking Cholesky method and its supernodal variant are used in industry and academia, for example, as linear system solvers invoked by the 'backslash' operator in Matlab. We evaluate our proposed FT methods on a popular open-source sparse direct solver, CHOLMOD. Since CHOLMOD only uses the left-looking variant of Cholesky factorization, for the remainder of this paper, we will refer to the "left-looking Cholesky" method as simply the "Cholesky" method, and likewise for its supernodal variant.

In our evaluation, we utilize a variety of sparse input matrices from real applications and use a transient fault injection mechanism for NVIDIA GPUs. We show that our error checking methods have good error coverage and low overhead. The contributions of our paper are the following: 1) We introduce efficient error checking mechanisms for Cholesky factorization, as well as its supernodal variant. 2) We evaluate our FT schemes using a sparse direct solver on a GPU platform, with and without the injection of faults, using matrices from real applications. 3) We use an interrupt-based transient fault injection mechanism for NVIDIA GPUs. Our fault injection method can simulate faults that persist for varying durations.

The rest of this paper is organized as follows: Section II provides an overview of the architecture of NVIDIA GPUs, while Section III provides a brief overview of Cholesky factorization and its supernodal counterpart. Section IV provides justification for the need for fault tolerance in direct solvers. Section V covers various fault tolerant techniques for Cholesky factorization and Section VI provides details of our fault injection mechanism. Section VII explains our experimental setup while Section VIII discusses our results. Section IX concludes the paper.

II. Overview of GPU Architecture

A GPU is a processor that possesses the ability to execute a large number of threads in parallel. It has a large number of processing elements, along with a wide memory bus and fast off-chip memory. The CPU communicates with the GPU through the *PCI-Express* interface [11].

In this paper, we focus on NVIDIA GPUs that are based on the Fermi architecture [15], like the GTX 480. The GTX 480 is capable of performing about 1.3 teraFLOPS and has 15 streaming multiprocessors (SMs). Each SM possesses 32 streaming processors, for a total of 480 SPs available. This GPU also has 16 LD/ST units (for memory operations), 4 special function units, a 32,768 $\times$ 32-bit register file, 64KB of shared memory/L1 cache and 768KB of L2 cache, as well as 1.5GB of off-chip memory.

Contemporary NVIDIA GPU families, including those based on the Fermi architecture, have support for ECC-protected register files, caches and off-chip memory. Each kernel that is executed 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 1 shows one streaming multiprocessor of the GTX 480.

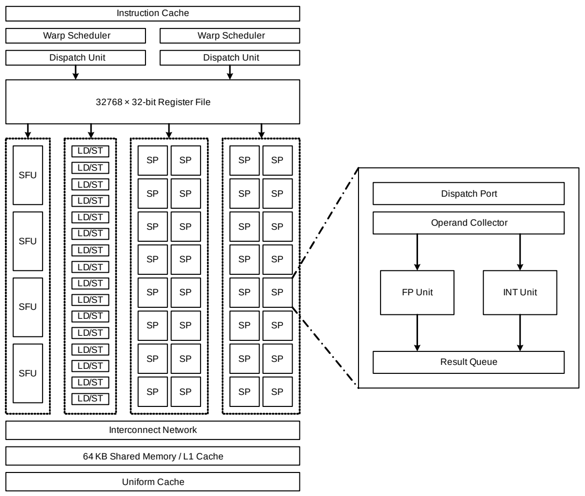


Fig. 1: A single SM of the GTX 480.

III. Overview of Cholesky Factorization

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

A. Cholesky Factorization

Cholesky factorization is a frequently used method of factorizing sparse symmetric positive definite matrices [6]. 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 1 illustrates the sparse 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, only the non-zero elements of L are computed.

Algorithm 1 Compute sparse factorization $A \in \mathbf{R}^{n\times n} = LL^T$

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

B. Supernodal Cholesky Factorization

The supernodal Cholesky method works similarly to the basic sparse 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 [14]. The columns in each supernode have a similar non-zero pattern and this allows the supernodal Cholesky method to maximize throughput by operating on dense blocks of sub-matrices. Figure 2 shows a sample L matrix divided into 5 supernodes.

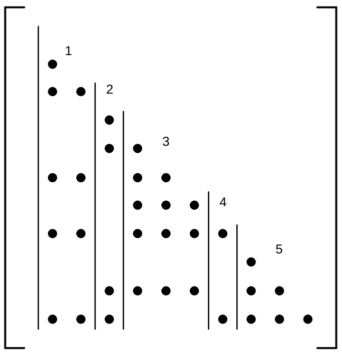


Fig. 2: Supernodes for a sample L matrix.

During each iteration, in a process analogous to the computations performed by the basic sparse Cholesky factorization method, the supernodal Cholesky method first computes the left-looking update of all columns in the current supernode, similar to line 5 of algorithm 1. Then it factorizes the diagonal block of the current supernode using block Cholesky factorization, which is equivalent to line 6 of algorithm 1. Finally, similar to line 7 of algorithm 1, the supernodal 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 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 direct the interested reader to Davis et al [6] for more details on the supernodal Cholesky factorization.

IV. Necessity for Fault Tolerance

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.

A. CHOLMOD

CHOLMOD [4] is a popular open-source application for sparse matrices that is able to perform two major kinds of factorizations, which are 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 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 Cholesky factorization is specified. The GPU is utilized only in the numerical factorization phase.

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.

B. 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 Cholesky method, like CHOLMOD. In these experiments, we ran the basic CHOLMOD program with no fault tolerance, using the supernodal Cholesky factorization with the nd6k matrix from the University of Florida sparse matrix collection [5]. Faults are injected into the GPU by flipping a chosen bit position using our fault injection mechanism. We ran a few hundred executions for each bit position to ensure that the results are statistically significant. 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 3, 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 not nearly as bad, but still unacceptable, 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 silent data corruption (SDC), are not apparent to 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.

V. Fault Tolerance Methods

Before we discuss various FT methods for each solver, we first describe some of their invariant properties.

A. Invariant Properties

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 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 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 Cholesky factorization, where only one supernode is modified and finalized after each iteration.

B. Fault Detection Methods

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

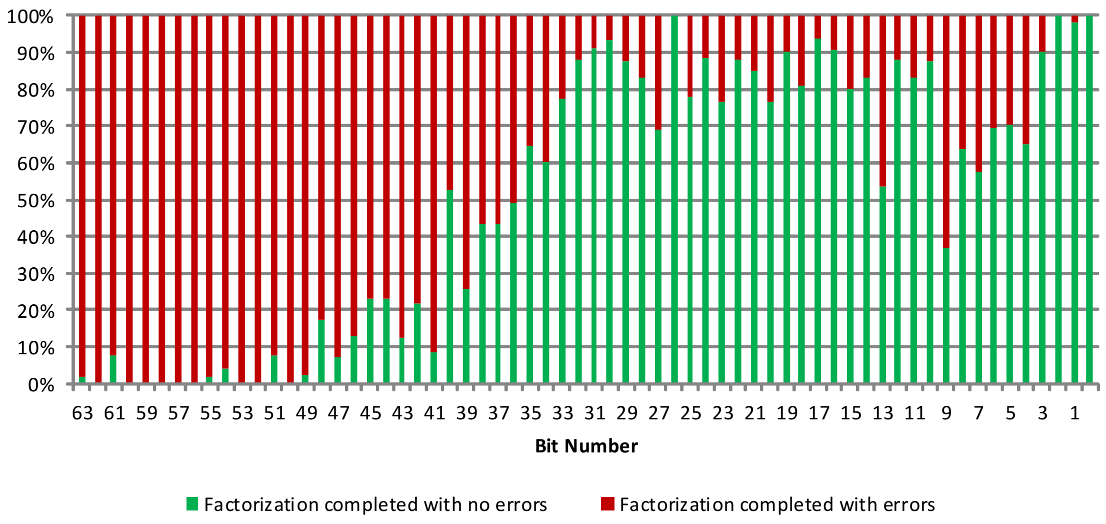


Fig. 3: CHOLMOD factorization outcomes for various bit positions.

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, we can ensure that only the columns of L that have been computed up to the current iteration are involved in calculating xLL^Ty. We pick the elements of the vectors as follows.

Suppose at the end of the kth iteration, we have 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 $c \le i \le c+b$, 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 $j > c+b$, we don't 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; we only need 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 us to check the elements of the recently computed supernode with low overhead.

The error coverage of this method is effectively 100%, and this coverage is similar to the column checksum method discussed later in this section, 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 $|xAy| \ge 1$, or if $|xLL^Ty - xAy| > 10^{-6}$ for $|xAy| < 1$.

At this point, it is worth noting that one could simply check the invariant xAy and xLL^Ty, where $x_i = y_i = 1$ for all i, *once* at the end of the last iteration, after factorization is complete. While this method would have a very low overhead of checking, the detection latency is high and this can substantially increase the cost of recovery and the overall runtime, in the event of errors. Our method offers a good balance between short detection latency and low checking overhead, and this overhead is unlikely to be significantly higher than that of checking the invariant once at the end of factorization.

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 – unlike the xAy vs xLL^Ty checking method, the product xLL^Ty may *not* account for all columns in the current supernode.

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

3) Column Checksum: Wu et al [17] 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 Cholesky factorization. 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 implement a checksum scheme similar to the one proposed by Chen et al [3], using only one column checksum each for A and L, and compare this scheme against our 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 expected to be 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 $|ColChecksum| < 1$.

C. Coverage of Fault Detection Methods

Figure 4(a) illustrates which elements of the sample matrix L are computed and finalized during iteration 3 of the supernodal Cholesky factorization method. Our definition of a "finalized" element is one that was computed in the current iteration of the Cholesky factorization, and is not modified further in subsequent iterations. Figure 4(b) shows the elements of L that are conclusively or definitively checked by the xAy vs xLL^Ty checking method and the column checksum scheme, during the third supernodal iteration. Both of these FT methods are able to check all elements in the 3rd supernode.

Similarly, Figure 4(c) shows the elements of L that are conclusively checked by the simplified xAy vs xLL^Ty checking scheme in the same iteration. Note that in this case, the first two columns of supernode 3 cannot be guaranteed to be checked. This is because the elements to the left of the diagonal element in the 3rd column of supernode 3, i.e. elements in the same row as this diagonal element, might possibly be computed zeroes (the diagonal element is guaranteed to be non-zero). As a result, even though $v = L^Ty$ does take into account elements in the first two columns of the 3rd supernode, the relevant elements of v are zeroed out in the dot product $uv = xLL^Ty$.

D. Fault Recovery Method

There are several ways to recover the correct value of the elements in a supernode, once a fault is detected in some iteration. The most efficient method would be to use the equations for Cholesky factorization, as shown in section III-A, to recover each column in the erroneous supernode, starting from the left-most column in that supernode. Algorithm 2 illustrates this process, and uses the xAy vs xLL^Ty checking method. A major benefit of this approach is that it does not require additional memory to store checkpoints and re-execution of the current iteration is not necessary.

Other methods of recovery include taking checkpoints for L each iteration and re-executing the iteration in which errors were detected, as well as simply restarting the Cholesky factorization upon detection of a fault.

Algorithm 2 A FT procedure for supernodal Cholesky factorization

```
1:  procedure FT SUPERNODAL CHOLESKY($A, L$)
2:      do symbolic factorization to find non-zero struc. of L
3:      for each supernode $s$ of $L$ do
4:          compute elements in supernode $s$
5:          perform $xAy$ vs $xLL^Ty$ checking:
6:          c = index of first column of supernode $s$
7:          b = no. of columns in supernode $s$
8:          $x = [0 ... 0\ 1 ... 1\ 0 ... 0]$ ($x_i$ = 1, $c \leq i \leq c+b$)
9:          $y = [1\ 1 ... 1]^T$ ($y_i$ = 1 for all $i$)
10:         compute and compare $xAy$ with $xLL^Ty$
11:         if errors detected then
12:             recover each column of supernode $s$:
13:             $d = [0\ \ 0\ \ ...\ \ 0]^T$
14:             for $k = c : c+b$ do
15:                 $d_{k:n} = A_{k:n,k} - L_{k:n,1:k-1}L^T_{k,1:k-1}$
16:                 $L_{k,k} = \sqrt{d_k}$
17:                 $L_{k+1:n,k} = \frac{d_{k+1:n}}{L_{k,k}}$
18:                 $d = [0\ \ 0\ \ ...\ \ 0]^T$
19:             end for
20:         end if
21:     end for
22:     return $L$
23: end procedure
```

As we mentioned previously, the supernodal left-looking Cholesky method has a nice property in which all elements of a particular supernode are computed in one iteration and no other elements are updated. However, CHOLMOD does access and modify other metadata structures during factorization, and these data structures would also need to be checkpointed order to successfully recover from an error. For this reason, we instead choose to restart the numerical factorization when an error is detected, so as to avoid the significant overhead of checkpointing all of these data structures every iteration. Our choice of recovery only requires a one-time checkpoint of the initial clean version of L. While the overhead of recovery will be high, we justify this choice by observing that the occurrence of an error is generally rare, and it is expected that recovery will not be necessary most of the time.

E. Properties for Fault Tolerance Schemes

In this section, we list three desirable properties that FT schemes should possess, and briefly discuss how well the three FT methods described in section V-B meet these requirements.

- **Correctness**: The FT method checks all elements that are computed and finalized during the current iteration (the scheme does not need to check any element that is not computed or finalized). All three FT schemes satisfy the correctness requirement.
- **Completeness**: The FT method eventually checks all elements that are computed, by the end of factorization of the input matrix. In this case, only the xAy vs xLL^Ty checking method and the column checksum scheme meet the requirements of the "completeness" property adequately. The simplified xAy vs xLL^Ty checking method cannot guarantee that all computed elements are eventually checked, and so does not have full error coverage.
- **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 method, if it is detected in the first place. Here, all three FT schemes are able to detect errors by the end of the same iteration in which they occur; however, the simplified xAy vs xLL^Ty checking method does not have full error coverage.

VI. Fault Injection Mechanism

In this paper, we only consider transient hardware faults that persist for a certain length of time, which affect the processing elements of the GPU and lead to SDC. We consider faults that occur in the SP cores during execution, and allow for the possibility that corrupted results can be written to memory by threads running on faulty cores. Our fault injection mechanism is similar to that used by other researchers [2], [12], [13]. It uses 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) for fault injection. The fault arrivals are assumed to form a Poisson process with a user-specified rate. All threads that happen to be running on the chosen SP when the fault is injected will have their results corrupted. This is done by flipping a predetermined bit in the 64-bit double-precision floating point result of each of those threads.

VII. Experimental Methodology

We run all experiments using a GPU compute cluster, using a single node and one of its four GTX 480 GPUs. Table I shows the specifications of one such node.

TABLE I: 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 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 [5] that represent a variety of different applications, as shown in Table II. We executed several hundred runs for each matrix, for each category.

TABLE II: 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

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 II. We measure the average time needed for factorization of the input matrix. We compare 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 we 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 $||LL^Tx - b||_2/||b||_2 > 10^{-6}$ as erroneous. As mentioned earlier, we choose to restart the factorization from the beginning, in the event that an error is detected in some iteration.

VIII. Results

A. Performance of FT Schemes in the Absence of Faults

Figure 5 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. This measures the overhead of error checking alone. As shown in figure 5, 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 scheme (but by only a small margin), while the column checksum scheme always has the largest overhead. The overhead of the column checksum scheme varies from 39.9% (nasa4704) to 53.0% (aft01). By contrast, the overhead of our proposed FT method, the xAy vs xLL^Ty checking scheme, ranges from 11.3% (bloweybq) to 17.8% (fv2). Our other proposed FT method, the simplified xAy vs xLL^Ty checking scheme, has similar overhead. Therefore, the xAy vs xLL^Ty checking scheme and the simplified xAy vs xLL^Ty checking scheme both have low overhead; on the other hand, the column checksum scheme is not as efficient.

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

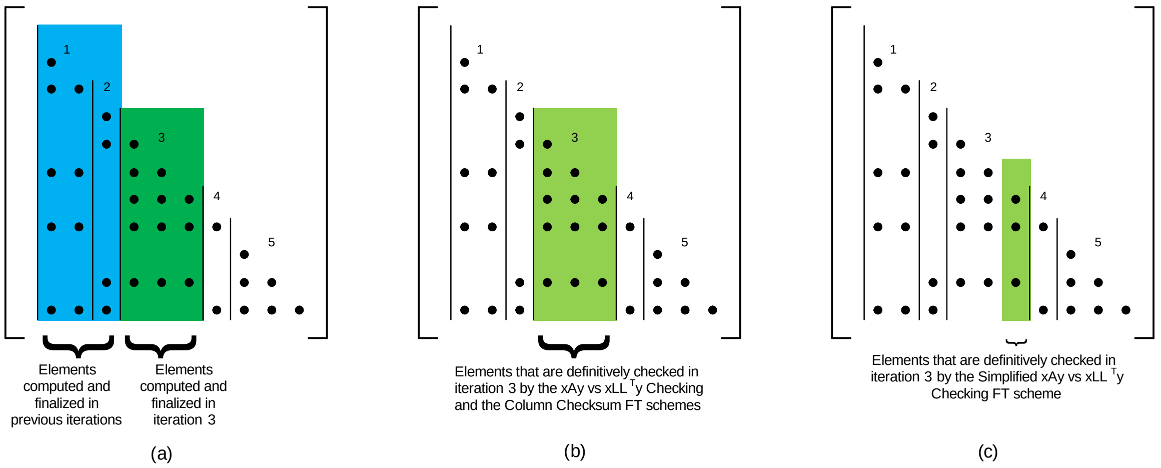

1
2
3
4
5
Elements computed and finalized in previous iterations
Elements computed and finalized in iteration 3
(a)
Elements that are definitively checked in iteration 3 by the xAy vs xLL^{T}y Checking and the Column Checksum FT schemes
(b)
Elements that are definitively checked in iteration 3 by the Simplified xAy vs xLL^{T}y Checking FT scheme
(c)

Fig. 4: Elements of the sample L matrix that are computed and checked in the third iteration

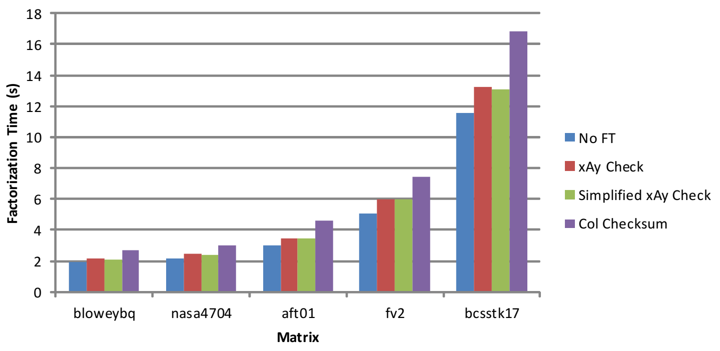


Fig. 5: Performance of various FT schemes with no fault injection

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 Cholesky factorization can offer good performance, compared to traditional checksum schemes.

B. *Performance of FT Schemes with Fault Injection*

Figure 6(a) shows the error coverage of the FT schemes when faults are injected, while figure 6(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. This measures the overhead of error checking and recovery.

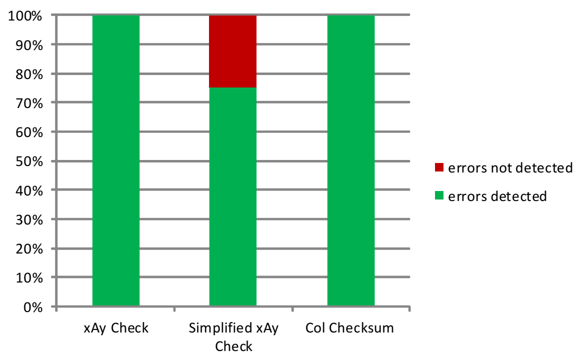
From figure 6(a), one can see that both the xAy vs xLL^Ty checking scheme and the column checksum scheme have excellent error detection coverage, with practically all errors getting detected, thus satisfying the completeness requirement, as we expect. However, the simplified xAy vs xLL^Ty checking scheme suffers from poorer coverage (about 75%) due to the fact that it does not check all elements that are computed during any supernodal iteration. This correlates with our earlier assertion that the simplified xAy vs xLL^Ty checking scheme fails to meet the completeness requirement adequately.

As can be seen in figure 6(b), the overhead of all three FT schemes go up when recovery was required. 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.

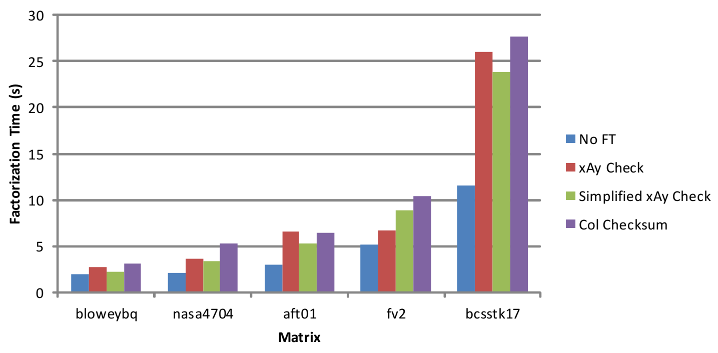
At this point, we would like to reiterate that all three FT schemes satisfy the "correctness" property. However, only the xAy vs xLL^Ty checking method and the column checksum scheme meet the requirements of the "completeness" property. Conversely, the simplified xAy vs xLL^Ty scheme does not address the requirements of the completeness metric to a satisfactory degree, since it can only check some of the elements that were finalized in a particular iteration. All three FT schemes have low latency, in the sense that each of the three schemes can potentially detect errors by the end of the same iteration in which they occur. In terms of efficiency, the xAy vs xLL^Ty checking scheme and the simplified xAy vs xLL^Ty checking scheme both perform well, while the col checksum scheme is considerably less efficient.

IX. Conclusion

In this paper, we implemented and analyzed two FT schemes for Cholesky factorization. We compared these schemes against a traditional checksum-based scheme using a direct Cholesky solver and show that our checking methods have relatively low overhead and good error coverage. In particular, we recommend the xAy vs xLL^Ty checking scheme,



(a)



(b)

Fig. 6: Performance of various FT schemes with fault injection

as it offers the best of both worlds: excellent error coverage and low error checking overhead.

References

[1] 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.

[2] 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)*, pages 443–454, 2014.

[3] 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.

[4] 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*, 35(3):22:1–22:14, October 2008.

[5] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, December 2011.

[6] T. A. Davis, S. Rajamanickam, and W. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25(1):383–566, May 2016.

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

[8] D. Hakkarinen, P. Wu, and Z. Chen. Fail-stop failure algorithm-based fault tolerance for Cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, May 2015.

[9] M. A. Heroux. Software challenges for extreme scale computing: Going from petascale to exascale systems. *The International Journal of High Performance Computing Applications*, 23(4):437–439, 2009.

[10] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[11] 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.

[12] 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, pages 63–70, 2015.

[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)*, pages 481–486, January 2016.

[14] E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14(5):1034–1056, 1993.

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

[16] 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*, pages 69–78, 2012.

[17] 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, pages 49–60, 2014.

[18] 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, pages 31–42, 2016.