Towards ABFT for BLIS GEMM

FLAME Working Note #76

Tyler M. Smith* Robert A. van de Geijn* Mikhail Smelyanskiy† Enrique S. Quintana-Ortí‡

Originally published June 13, 2015
Revised November 5, 2015

Abstract

Fault tolerance is a necessary ingredient if Exascale supercomputers are to operate within a reasonable power budget. Increased complexity in software is usually hidden in libraries, such as the Basic Linear Algebra Subprograms (BLAS), thus alleviating part of the effort required to attain high performance for a significant number of scientific applications and architectures. Can implementations of the BLAS similarly support algorithm-based fault tolerance (ABFT) without sacrificing the performance upon which applications count? We provide insight into this question by focusing on the most important BLAS operation, the matrix-matrix multiplication (GEMM). We demonstrate that ABFT can be incorporated into the BLAS-like Instantiation Software (BLIS) framework's implementation of this operation, degrading performance by only 10-15% on current multicore architectures like the Intel Xeon E5-2580 processor with 16 cores and cutting edge many-core architectures like the Intel Xeon Phi processor with 60 cores.

1 Introduction

Algorithm-based fault tolerance (ABFT) is an application-specific approach that takes advantage of specialized properties of the application to embed error detection and correction within the underlying algorithm. For example, matrix operations can take advantage of ABFT by verifying a pre-proved checksum relationship at the end of the calculation [1]. These ideas have been further extended to tolerate fail-stop failures in distributed environments without checkpointing [2].

Matrix multiplication of two dense matrices (GEMM) plays a key role in scientific and engineering applications, as many complex codes are built on top of linear algebra libraries (e.g., LAPACK [3], `libflame` [4], ScaLAPACK [5], Elemental [6], to name a few) that internally cast a large fraction of their computations in terms of GEMM. Moreover, techniques designed for GEMM can often be easily generalized to other matrix operations. Developing a high-performance fault-tolerant GEMM is therefore a crucial first step towards creating efficient fault-tolerant linear algebra libraries and, in consequence, more reliable scientific and engineering applications.

In [7], on-line error recovery was integrated within matrix multiplication by detecting errors after individual blocked outer-product computations of the result. The work in [8] expanded upon the original ABFT

*Department of Computer Science and Institute for Computational Engineering and Sciences, The University of Texas at Austin `{tms,rvdg,field}@cs.utexas.edu`.

†Parallel Computing Lab, Intel Corporation, `mikhail.smelyanskiy@intel.com`.

‡Universidad Jaime I `quintana@uji.es`

paper [1] and provided the key insights that underly the current work: To implement a high-performance GEMM one must amortize the movement of $O(n^2)$ data (the matrices) over $O(n^3)$ computation. Checking the integrity of the data similarly requires $O(n^2)$ operations. Concretely, our work demonstrated that, for the existing architectures and the best known algorithm for GEMM at that time, high performance could be maintained. The current paper reexamines the results from [8] for the significantly more advanced modern multicore and many-core architectures that did not exist in 2001. While the algorithm for implementing GEMM from [9] that underlied the work in [8] was then state-of-the-art, shortly afterwards Goto introduced the techniques that are at the core of most recent high-performance implementations, including those incorporated in GotoBLAS [10], OpenBLAS [11], Intel's MKL [12], AMD's ACML [13], and IBM's ESSL [14]. Since this approach exposes many more levels of blocking, a careful study of where ABFT can be added to these implementations is in order. In addition, the BLAS-like Library Instantiation Software (BLIS) [15, 16, 17] framework now provides a convenient infrastruction with which to evaluate and analyze insights.

In short, the current work provides a thorough examination of how to incorporate ABFT into practical implementations of GEMM for modern multicore and many-core architectures, supporting applications that can execute on Exascale architectures by providing fault-tolerance at the node level and demonstrating high performance. For this purpose, we revisit GEMM with the goal of providing a software layer that can tolerate multiple errors while retaining high performance. We consider *silent data corruption* [18, 19] only, which does not abort the program execution but may yield an incorrect result of the computation if not properly addressed. Furthermore, we assume that these errors manifest themselves in incorrect results of floating point computation. As such they can originate anywhere in the processor datapath; e.g., in the register file, floating point units (FPUs), reorder buffer, front-end of the pipeline, or cache controller. In this initial study, we finally assume that errors in the L1 cache memory or below are solved via conventional error-correcting code (ECC) memory.

In the remainder of the paper, we will consider the "extended" form of GEMM, $C \mathrel{+}= AB$, where $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$, though in some cases we will simplify it to the more "basic" case$C := AB$. Hereafter, we will define the problem dimension using the triple (m, n, k).

2 BLIS and Potential Errors

In this section, we briefly review how BLIS implements GEMM and succinctly discuss the effects of errors into the BLIS implementation of this operation. While this paper is self-contained, it is recommended that the reader be familiar with [15] and [17].

BLIS implements GEMM as three external loops involving two packing routines around a macro-kernel that computes the suboperation $C_c \mathrel{+}= A_cB_c$, of size (m_c, n_c, k_c); see Fig. 1 and the loops 3–5 there. Note that A_c, B_c correspond to actual buffers involved in data copies, while $C_c \equiv C(\mathcal{I}_c, \mathcal{J}_c)$ is just a notation artifact, introduced to ease the presentation of the algorithm.

Internally, the macro-kernel consists of two additional loops around a micro-kernel that computes

$$C_c(\mathcal{I}_r, \mathcal{J}_r) \mathrel{+}= A_c(\mathcal{I}_r, 0 : k_c - 1)\ B_c(0 : k_c - 1, \mathcal{J}_r),$$

of size (m_r, n_r, k_c); see again Fig. 1 and the loops 1 and 2 there. The BLIS micro-kernel is typically implemented using assembly code or with vector intrinsics, as a loop around a rank–1 (i.e., outer product) update; see loop 0 in Fig. 1. The remaining five loops are implemented in C.[1]

The performance of the BLIS implementation strongly depends on that of the micro-kernel plus the selected blocking parameters m_c, n_c, k_c, m_r and n_r. An appropriate choice of these values: *i)* yields a near-perfect overlap of communication (data fetching into the FPUs) with computation; *ii)* loads B_c into

```
Loop 5  for $j_c = 0, \ldots, n-1$ in steps of $n_c$, $\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4    for $p_c = 0, \ldots, k-1$ in steps of $k_c$, $\mathcal{P}_c = p_c : p_c + k_c - 1$
            $B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow B_c$                                  // Pack into $B_c$
Loop 3      for $i_c = 0, \ldots, m-1$ in steps of $m_c$, $\mathcal{I}_c = i_c : i_c + m_c - 1$
              $A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow A_c$                                // Pack into $A_c$
Loop 2        for $j_r = 0, \ldots, n_c-1$ in steps of $n_r$, $\mathcal{J}_r = j_r : j_r + n_r - 1$   // Macro-kernel
Loop 1          for $i_r = 0, \ldots, m_c-1$ in steps of $m_r$, $\mathcal{I}_r = i_r : i_r + m_r - 1$
Loop 0            for $k_r = 0, \ldots, k_c - 1$                                     // Micro-kernel
                        $C_c(\mathcal{I}_r, \mathcal{J}_r)$ += $A_c(\mathcal{I}_r, k_r)$ $B_c(k_r, \mathcal{J}_r)$
                  endfor
                endfor
              endfor
            endfor
          endfor
        endfor
```

Figure 1: High performance implementation of GEMM in BLIS.

the L3 cache (if there is one) when this block is packed; *iii)* loads B_c into the L1 cache in micro-panels of n_r columns (say B_r) from inside the micro-kernel; *iv)* loads A_c into the L2 cache when packed; *v)* and, from the micro-kernel, loads/stores C from/to the main memory into/from the file register, and streams B_r/A_c from the L1/L2 cache into the FPUs; see [15]. In practice, the BLIS implementations have been demonstrated to deliver performance that rivals those of proprietary libraries such as Intel MKL [12], AMD AMCL [13], and IBM ESSL [14], as well open source libraries like ATLAS [20] and OpenBLAS [11], on a wide variety of modern computer architectures [16, 17]. Since all modern implementations of GEMM are based on the same general approach pioneered by the GotoBLAS [10], the insights in this paper can be applied to any of these.

A key observation in [8] is that, for high-performance implementations of GEMM, the cost moving data is $\mathcal{O}(mn + mk + kn)$, the cost of checking the correctness is $\mathcal{O}(mn + mk + kn)$, and both overheads can be potentially amortized over $\mathcal{O}(mnk)$ computation. Therefore, there is the possibility of incorporating error checking into the data movements, much of which is explicitly exposed as packing into contiguous memory. The question now becomes how to achieve this.

Categorizing errors. Let us analyze silent data corruption that may occur inside of BLIS and its effects. From an abstract point of view, we will consider corruption that may occur either as floating-point data resides in the unprotected register files or during computation with it.

The key to any high performance implementation of GEMM is a careful reuse of data, in order to amortize the cost of data movements across the memory hierarchy over a great deal of computation. The danger of this reuse within the context of resiliency is that, through reuse, errors can propagate. Concretely, elements of A/A_c or B/B_c that are corrupted while in the register files may be reused many times, potentially corrupting many elements of the result C. For example, consider a single iteration of loop 0 in Fig. 1, where the k_r-th column/row of $A_c(\mathcal{I}_r, 0 : k_r - 1)/B_c(0 : k_r - 1, \mathcal{J}_r)$ are copied into the file register and then multiplied to update a micro-block of C_c. Each element of the k_r-th column of $A_c(\mathcal{I}_r, 0 : k_r - 1)$ contributes to n_r elements of this micro-block of C_c, and thus if a single element of this column is corrupted during the copy operation, up to n_r elements of C will be affected.

Because of this, we will consider both errors that occur during computation of C, and errors that occur when moving matrices A and B through the memory hierarchy.

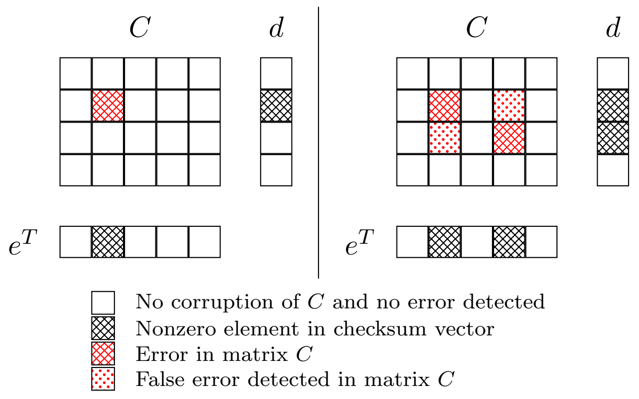
C
d
C
d
e^T
e^T
No corruption of C and no error detected
Nonzero element in checksum vector
Error in matrix C
False error detected in matrix C

Figure 2: This diagram illustrates how to detect where errors may occur, and hence which elements of matrix C must be recomputed, using checksum vectors. Left: There is one error in C and it is detected. Right: Two errors are in C, leading to 2 false postives.

3 Detecting and Correcting

We open this section with a brief review of the two-sided checksum-based method for GEMM introduced in [1], and the technique formulated in [21, 8] to detect soft errors and distinguish them from those intrinsic to the use of finite precision arithmetic.

Assume for the moment that we are interested in computing the basic product $C := AB$. Consider next the augmented matrices

$$A^* = \left(\begin{array}{c} A \\ \hline v^T A \end{array} \right), B^* = \left(\begin{array}{c|c} B & Bw \end{array} \right), C^* = \left(\begin{array}{c|c} C & Cw \\ \hline v^T C & v^T C w \end{array} \right),$$

where v^T and w are respectively row and column vectors with m and n random components. In the absence of errors and in exact arithmetic, the result then satisfies $C^* = A^* B^*$, and a simple mechanism to detect an error is to verify whether

$$\begin{array}{r} \|d\|_\infty = \|Cw - A(Bw)\|_\infty \quad > 0 \quad \text{or} \\ \|e^T\|_\infty = \|v^T C - (v^T A)B\|_\infty > 0. \end{array} \tag{1}$$

Here, d and e^T are referred to as the *left and right checksum vectors* respectively, yielding a two-sided error detection method. Note that in case w is orthogonal to one of the rows of B or v^T is orthogonal to one of the columns of A, a one-sided error detection method is not guaranteed to detect an error if A or B is corrupted. However we consider this unlikely to occur in practice, and we assume that this will not occur.

In a real scenario, round-off error occurs and the previous criteria are modified in to declare an error if

$$\begin{array}{r} \|d\|_\infty > \tau \|A\|_\infty \|B\|_\infty \;\; \text{or} \\ \|e^T\|_\infty > \tau \|A\|_\infty \|B\|_\infty, \end{array} \tag{2}$$

where $\tau = k\,u$, and u denotes the unit round-off of the machine [21, 8], and k is the inner dimension of the matrix multiplication.

The location of the errors may be determined by inspecting the entries of d and e^T Concretely, if the j-th entry of vector d satisfies $|d_j| > \tau \|A\|_\infty \|B\|_\infty = \rho$, there is a significant error somewhere in the j^{th} column of C. Similarly if $|e_i^T| > \rho$, there is an error somewhere in the i^{th} row of C. Then, each error will occur at some combination of i and j coordinates, but there may be false positives, as illustrated in Figure 2.

Handling $C \mathrel{+}= AB$. When performing the "basic" operation $C := AB$, the checksum detection scheme introduced in the previous section is sufficient. In this case, those parts of C that may be corrupted can be simply recomputed. However in case of the more complicated operation $C \mathrel{+}= AB$, two additional problems arise. First the checksum approach simply does not apply to this operation; it only applies to the basic operation. Secondly, if a corrupted (intermediate) result is added to some element of the matrix C, then it may be impossible to recover that element of C. We present two methods to deal with these issues.

First, one can perform the operation $\hat{C} := AB$, using checksum vectors to test the correctness of this operation. If no error was detected, $\hat{C}$ is next added to the original matrix C. A second possibility is to checkpoint (i.e., copy) C as $\check{C} := C$. Next, perform $T := AB$ and $C \mathrel{+}= T$ with errors detected during the computation of $T := AB$. If errors occur, C can be rolled back by performing $C := \check{C}$, and restarting the operation.

Comparison with conventional ABFT. Traditional approaches for ABFT often involve estimating the difference between a corrupted result from an operation and the correctly computed result, and then subtracting that difference from the corrupted result to yield the correct result. We chose a simpler checkpoint and restart approach for two reasons. First, we have concerns that subtracting the estimated error from the computed result may give rise to numerical stability issues, mainly due to catastrophic cancellation. Second, in such schemes, it is necessary to determine exactly both what the errors are and where they occurred. Such schemes may either fall apart or require extra checksums to account for multiple errors occurring in the same row or column of the result, for instance when an error occurs in A_c or B_c and propagates respectively corrupting several elements in a row or column of C. In contrast, our approach can tolerate multiple errors in the same row or column of C. At worst, these lead to false positives, and require the recomputation of uncorrupted elements of C.

4 Practical Solutions

Balancing the costs of fault tolerance. Computing d and e^T in (1) in the two-sided checksum-based method requires a total of six matrix-vector products (GEMV), for a cost of $4(mn+mk+kn)$ floating-point arithmetic operations (flops), plus the matrix norms involving A and B to verify (2), for an additional $mk + kn$ flops. (Hereafter, we neglect lower order terms in the costs.) Therefore, the overhead of this two-sided error detection mechanism, relative to the total problem cost, is given by:

$$\mathcal{O}_d(m, n, k) = \frac{4mn + 5mk + 5kn}{\mathcal{O}_c(m, n, k)} = \frac{4mn + 5mk + 5kn}{2mnk},$$

where the denominator $\mathcal{O}_c(m, n, k)$ also equals the cost of an error correction mechanism which recovers from an error by simply recomputing the complete product. Provided m, n, k are large, the overhead of checking whether an error occurred is thus negligible compared with the $2mnk$ flops of GEMM. Notice that the true cost of the checksums is due more to data movements than the flops involved, so we will take advantage of the data movements that naturally happen in GEMM to reduce these costs.

This general approach has to be performed "off-line", that is, once the matrix multiplication is completed. Thus, in case an error is detected, the operation has to be completely repeated. Furthermore, the extended multiplication $C \mathrel{+}= AB$ requires the original contents of C to be kept (see section 3), in case an error is detected and it becomes necessary to restore the data. For large problems, this extra workspace can be prohibitive.

Inside loop	Loop index	Required workspace	$\mathcal{O}_d$ and $\mathcal{O}_c$ depend on
5	j_c	$m \times n_c$	(m, n_c, k)
4	p_c	$m \times n_c$	(m, n_c, k_c)
3	i_c	$m_c \times n_c$	(m_c, n_c, k_c)
2	j_r	$m_c \times n_r$	(m_c, n_r, k_c)
1	i_r	$m_r \times n_r$	(m_r, n_r, k_c)
0	k_r	$m_r \times n_r$	$(m_r, n_r, 1)$

Figure 3: Analysis of the costs of the fault tolerance mechanism depending on the layer where/ loop inside which it is applied.

Fortunately, the multi-layered organization of the BLIS GEMM allows the application of the checksum technique at different levels (i.e., within different loops), trading off workspace and error correction cost for error detection overhead. In particular, inside each loop, the original BLIS algorithm updates a certain part of C using certain blocks of A and B, say $C_p \mathrel{+}= A_pB_p$ of dimension (m_p, n_p, k_p). Therefore, we can compute $\hat{C}_p := A_pB_p$ using the workspace $\hat{C}_p$, and update C_p if no error is present. Alternatively, we can perform the following operations: (1) $\check{C}_p := C_p$, (2) $T := A_pB_p$, and (3) $C_p \mathrel{+}= T$, so that we can always roll C_p back to $\check{C}_p$ if an error is present. These techniques require $m_p \times n_p$ elements as additional workspace, for $\hat{C}_p$ or $\check{C}_p$. It may appear that it requires workspace for T as well, but we will later discuss how to fuse these three operations such that T is not explicitly formed. Both alternatives also incur an extra cost for error recovery (correction) of $\mathcal{O}_c(m_p, n_p, k_p)$ flops and present a relative overhead incurred for the error detection mechanism of $\mathcal{O}_d(m_p, n_p, k_p)$. Fig. 3 details these costs when applied to the indicated loop of the BLIS GEMM.

We emphasize that supporting the extended operation $Cp \mathrel{+}= A_pB_p$ is absolutely vital when accommodating fault tolerance at some layer within a rank-k update, even if the unpartitioned operation corresponds to the "basic" operation $C := AB$, as the second and subsequent rank-k updates will always add to C.

Focusing on the Third Loop. Whether the overhead costs in Fig. 3 are acceptable or not strongly depends on the problem dimensions as well as the concrete values of the five BLIS blocking parameters. Consider for instance the Intel Xeon E5-2680 target processor (the Intel Xeon Phi Knight's corner has a similar story): $(m_c, n_c, k_c) = (96, 4096, 256)$ and $(m_r, n_r) = (8, 4)$ This particular value of n_c likely turns the checksum approach at the two outermost Loops (Loops 5 and 4) too expensive from the point of view of workspace, as they both require storage for $m \times n_c = m \times 4,096$ numbers. Furthermore, the small values of m_r and n_r make the detection of errors inside the innermost three Loops (Loops 2, 1, and 0) relatively expensive due to the high overhead for detection they would incur. Therefore, the only reasonable choice for this particular architecture is to apply the error recovery mechanism inside the macro-kernel (Loop 3), that is, when $C_c \mathrel{+}= A_cB_c$ is computed. This choice balances a moderate workspace ($96 \times 4,096$ numbers), with reasonable error correction cost, and a low relative overhead for error detection: $\mathcal{O}_d(m_c, n_c, k_c) = (4m_cn_c + 5m_ck_c + 5k_cn_c)/(2m_cn_ck_c) \leq 5/m_c = 5/192 \approx 2.6\%$. Note that the memory operations (memops) are more costly than the flops when performing the checksums. As we detail next, this cost can be reduced if, e.g., we reuse the matrix norms for A_c, B_c or the results of certain GEMV products.

Figure 4 shows a fault-tolerant version of BLIS GEMM, with the changes with respect to the original algorithm highlighted in colors: red for the computation of the right checksum vector; blue for the computation of the left checksum vector; and green for those parts involved in error recovery. We next review

```
Loop 4  for $p_c = 0,\ldots,k-1$ in steps of $k_c$, $\mathcal{P}_c = p_c : p_c + k_c - 1$
Loop 5    for $j_c = 0,\ldots,n-1$ in steps of $n_c$, $\mathcal{J}_c = j_c : j_c + n_c - 1$
            $B(\mathcal{P}_c,\mathcal{J}_c) \rightarrow B_c$                      // Pack into $B_c$ plus simultaneous
            $d_b := -B_c \cdot w$                                  //     right checksum
Loop 3      for $i_c = 0,\ldots,m-1$ in steps of $m_c$, $\mathcal{I}_c = i_c : i_c + m_c - 1$
              $A(\mathcal{I}_c,\mathcal{P}_c) \rightarrow A_c$                    // Pack into $A_c$ plus simultaneous
              $d := A_c \cdot d_b$ $(= A_c \cdot B_c \cdot d_b)$                //     right checksum
------------------------------------------------------------------------------------------
Loop 2        for $j_r = 0,\ldots,n_c-1$ in steps of $n_r$, $\mathcal{J}_r = j_r : j_r + n_r - 1$   // Macro-kernel $\check{C}_c = A_c \cdot B_c$
Loop 1          for $i_r = 0,\ldots,m_c-1$ in steps of $m_r$, $\mathcal{I}_r = i_r : i_r + m_r - 1$
                  $T = A_c(\mathcal{I}_r, 0 : k_c-1) \cdot B_c(0 : k_c-1, \mathcal{J}_r)$     // $T$ resides in registers
                  $d(\mathcal{I}_r) \mathrel{+}= T \cdot w(\mathcal{J}_r)$                  // Update right checksum: $d = T \cdot w - A_c \cdot B_c \cdot w$
                  $e^T(\mathcal{J}_r) := v^T(\mathcal{I}_r) \cdot T$                  // Update left checksum: $e^T = v^T \cdot T - v^T \cdot A_c \cdot B_c$
                  $\check{C}_c(\mathcal{I}_r,\mathcal{J}_r) := C_c(\mathcal{I}_r,\mathcal{J}_r)$              // Perform checkpoint: $\check{C}_c := C_c$
                  $C_c(\mathcal{I}_r,\mathcal{J}_r) \mathrel{+}= T$                      // Update C
                endfor
              endfor
------------------------------------------------------------------------------------------
              if $(\|d\|_\infty > \tau \|A\|_\infty \|B\|_\infty)$
                $e_a^T := v^T \cdot A_c$                                //     left checksum
                $e_{ab}^T(\mathcal{J}_r) := e_a^T \cdot B_c(0 : k_c-1, \mathcal{J}_r)$
                Detect error locations using checksum vectors $d$ and ($e^T$ - $e_{ab}^T$), and record these locations
                Roll back corrupted elements of $C$ using $\check{C}$
                If one entire column of $C$ has been corrupted, repack $B_c$
            endfor
          endfor
          For every corrupted micro-kernel of $C$, recompute $C(\mathcal{I}_r,\mathcal{J}_r) \mathrel{+}= A(\mathcal{I}_r, 0 : k_c-1) \cdot B(0 : k_c-1 : \mathcal{J}_r)$
        endfor
```

Figure 4: Our high performance implementation of GEMM in BLIS with integrated fault tolerance after applying the optimizations of lazy left checksums, lazy recomputation, checkpointing C, and repacking B.

each one of this parts in detail.

Right checksum. We start by noting that d_b and d are small column vectors, of dimension k_c and m_c respectively. The two GEMV products involved in this checksum, $d_b := -B_c w$ and $d := A_c d_b$, can be performed when the corresponding blocks, B_c and A_c, are packed and, implicitly, loaded into the L3 and L2 caches, respectively. In BLIS, packing is a memory-bound operation and, therefore, we can expect that adding a GEMV to it has no real effect on its execution time. In the GEMV with B_c, the cost of this operation is furthermore amortized over the iterations of Loop 3 (i.e., several executions of the macro-kernel), amortizing the cost of this larger GEMV over more computation. Similar arguments hold for the computation of $\|B_c\|_\infty$ and $\|A_c\|_\infty$.

Each execution of the micro-kernel computes a contribution $C_c(\mathcal{I}_r,\mathcal{J}_r)$ to be accumulated into the corresponding micro-block of C_c if the global result passes the checksum test. We exploit that, after the execution of the micro-kernel, $C_c(\mathcal{I}_r,\mathcal{J}_r)$ is available in the processor registers to compute the GEMV $d(\mathcal{I}_r) \mathrel{+}= C_c(\mathcal{I}_r,\mathcal{J}_r) w(\mathcal{J}_r)$ as part of the computation $d := C_c w$ $(-A_c B_c w)$. Unlike the packing operations, this GEMV is not embedded into a memory-bound operation and its cost is explicitly exposed. However, we expect its impact to be negligible, since it requires $2m_c n_c$ flops compared with the $2m_c n_c k_c$ flops of the macro-kernel, with k_c in the range of 256.

Left checksum. In this case, e_a^T and e^T are both small row vectors of dimension m_c. The computation of this checksum follows the same ideas just exposed for its right counterpart, but in this case the vector w is first multiplied by A_c and then by B_c. Because of this, the fault-tolerance overhead is greater. While for the right checksum case, the GEMV operation can be used across multiple iterations of Loop 3, now both GEMV operations must be performed inside of the body of Loop 3.[1] Thus, the relative overhead

[1]Notice the benefit of how BLIS is structured: in the GotoBLAS and OpenBLAS implementations of GEMM, the macro-kernel is typically assembly coded, making it difficult to add the kind of changes we are now discussing.

introduced by the GEMV involving B_c is no longer amortized over several macro-kernels, and is roughly given by $2k_cn_c/(2m_cn_ck_c) = 1/m_c = 1/96 \approx 1.04\%$. The operation $e_a^TB_c$ cannot be performed during a packing operation, increasing its cost further.

Because the left checksum is more expensive than the right checksum, it is preferable to avoid computing the former. Since it is possible to detect errors using only the right checksum, when an error is detected with this mechanism, the left checksum can be obtained in order to locate where the error occured. During normal operation, no errors are expected to occur, and the fault-tolerant GEMM will thus be more efficient. Proceeding in this manner, we shift part of the cost from error detection to to error correction. We call this approach the **lazy left checksum**.

Preventing false negatives. Performing these checksums while packing can be dangerous, since if A or B is corrupted while it is in some unprotected layer of memory, that same corrupted A or B can be both copied into A_c or B_c and used to compute the left or right checksum. This could result in a false negative. A way to prevent this from happening is to read A or B from the fastest protected memory layer twice: Once for packing, and once for computing the checksum. This has a larger performance impact the fewer levels of memory that are protected.

Handling C += AB. We will now discuss the costs and tradeoffs of the two options for handling the extended GEMM operation C += AB when injecting fault tolerance at the third loop.

The first option is to check for errors before accumulating $\hat{C}_c$ into the final block of C in order to prevent corruption of the result. There are a couple of problems with this approach. (1) Since $\hat{C}_c$ is a larger $m_c \times n_c$ buffer we can expect its contents to lie low in the memory hierarchy. The update $C(\mathcal{I}_c, \mathcal{J}_c)$ += $\hat{C}_c$ is therefore a memory-bound operation. (2) If this operation is performed all at once, it may disturb the contents of the caches. Notice the following: $\hat{C}_c$ is $m_c \times n_c$. B_c is $n_c \times k_c$, m_c is typically on the same order of magnitude of k_c, and thus B_c and $\hat{C}_c$ are of similar size. If there is an L3 cache, n_c and k_c are chosen such that B_c occupies a large fraction of it. Because of this, the operation $C(\mathcal{I}_c, \mathcal{J}_c)$ += $\hat{C}_c$ is likely to bump large portions of B_c out of the L3 cache. The takeaway is that in the GotoBLAS approach, B_c is designed to reside in the L3 cache and then reused across many macro-kernels. However if this option is used, B_c will have to be re-read from main memory for each macro-kernel that uses it.

The other option is to checkpoint C_c into $\check{C}_c$, perform $T := A_cB_c$, and then C_c += T. These three operations can be fused during the BLIS micro-kernel in order to reduce memory movements. The micro-kernel computation, $\hat{C}_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r) := A_c(\tilde{\mathcal{I}}_r, 0 : k_c - 1)B_c(0 : k_c - 1, \tilde{\mathcal{J}}_r)$ is generally implemented as a block dot-product $T := A_c(\tilde{\mathcal{I}}_r, 0 : k_c - 1)B_c(0 : k_c - 1, \tilde{\mathcal{J}}_r)$ followed by the update $C(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$ += T, where the $m_r \times n_r$ matrix T resides in registers only. Thus, at the same time that $C(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$ is brought into registers to update T, it can also be used to checkpoint $\check{C}(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$. Thus the only extra memory movements needed for this technique are an extra $m_c \times n_c$ elements of C_c that must be written to $\check{C}_c$. Since each element of $\check{C}_c$ is only used once during each macro-kernel, it most likely lies high in the memory hierarchy, and possibly in main memory. Thus this approach saves $m_c \times n_c$ elements that do not need to be read from main memory per macro-kernel as compared to the first approch. The disadvantage is that recovering from an error now requires the corrupted elements of C to be rolled back. However, as errors are expected to be relatively rare, the second aproach may be preferred, as it is cheaper when detecting errors.

Error correction. Detecting errors at the macro-kernel level, via the checksum vectors, while computing the product with the granularity of a micro-kernel opens the door to a selective error recovery method. For example, a single error in a single entry of $d(\tilde{\mathcal{I}}_r)$ combined with an error in a single entry of $e^T(\tilde{\mathcal{J}}_r)$ points in the direction of a problem during the micro-kernel computation $C_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r) := A_c(\tilde{\mathcal{I}}_r, 0 : k_c - 1)B_c(0 : k_c - 1, \tilde{\mathcal{J}}_r)$. We can therefore recompute only that specific result, if we consider that to be an indivisible unit of computation (which it is the way BLIS is structured). This implies that error detection roughly comes with the overhead $\mathcal{O}_d(m_c, n_c, k_c)$ (as we explained in this section, it is actually lower if we reuse some of the operations), but the cost of error correction is reduced from $\mathcal{O}_c(m_r, n_r, k_c)$ to $\mathcal{O}_c(m_c, n_c, k_c)$.

The presence of one (or multiple) corrupted item(s) in $d(\tilde{\mathcal{I}}_r)$, but one (or multiple) corrupted item(s) in two blocks $e^T(\tilde{\mathcal{J}}_r^1)$, $e^T(\tilde{\mathcal{J}}_r^2)$ will require the recomputation of two micro-kernels. If the errors occur in two blocks $d(\tilde{\mathcal{I}}_r^1)$ and $d(\tilde{\mathcal{I}}_r^2)$ and two blocks $e^T(\tilde{\mathcal{J}}_r^1)$, $e^T(\tilde{\mathcal{J}}_r^2)$, then four micro-kernels have to be recomputed, and so forth

On the other hand, nothing prevents us from being more aggressive in the error correction strategy in case of single (or few) errors per micro-kernel. An error in a register holding a value for $C_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$ will appear as single corrupted elements in both $d(\tilde{\mathcal{I}}_r)$ and $e^T(\tilde{\mathcal{J}}_r)$. A single error in a register holding a value of $A_c(\tilde{\mathcal{I}}_r, 0 : k_c - 1)$ will corrupt a full row[2] of $\hat{C}_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$ and all entries of $d(\tilde{\mathcal{I}}_r)$. Analogously, a single error in a register holding a value of $B_c(0 : k_c - 1, \tilde{\mathcal{J}}_r)$ will corrupt an entire column of $C_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$ and all of $e^T(\tilde{\mathcal{J}}_r)$. In all these three cases we can recompute only the corrupted entries of $C_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$, as dictated by the specific corrupted entries of $d(\tilde{\mathcal{I}}_r)$ and $e^T(\tilde{\mathcal{J}}_r)$.

Repacking A or B. It is important that errors in either A_c or B_c do not propagate corrupting more elements of the result than necessary. Thus, we must detect which elements of A_c or B_c may be corrupted and repack them. An alternate solution is to use the original buffers containing A or B during recomputation.

When detecting and correcting errors at the macro-kernel level, by the time that errors are detected and corrected, A_c is no longer used and the buffer containing the packed block of A_c has been recycled to store the next block of A. This suggests that C_c be recomputed by using the original matrix A rather than repacking A. On the other hand B_c is reused many times after errors in it are detected, so repacking corrupted elements of B_c is beneficial to prevent errors from also occuring in future iterations of Loop 3.

While one cannot determine for certain if an element of A_c or B_c is corrupted, if (for instance) an entire column of C_c is corrupted, in such scenario it is most likely that an element of B_c was corrupted, rather than each element of C_c having been corrupted independently.

5 Multithreaded Parallelism

Much like the structure of the BLIS implementation allows one to systematically reason about where to insert a mechanism for fault tolerance, it also allows one to systematically reason about how to add thread-level parallelism, as discussed in [17]. In that paper, it is also demonstrated that, for many-core architectures, it is beneficial to parallelize multiple loops, including at least one loop that iterates along the m dimension of the problem and one that iterates along its n dimension. We now discuss how to merge these ideas with the proposed mechanisms for fault-tolerance. This creates constraints on where parallelism can be introduced because the presence of checksum GEMV operations introduces dependencies between iterations of some loops.

Loops to parallelize. Let us revisit Fig. 1. Parallelizing Loop 4 is not desirable [17], since then partial results must be computed, stored, and summed. For this reason, hereafter we focus on the other loops, taking into account that dependencies may appear if iterations of a loop share a checksum vector:

Loop 5. For Loop 5, all the fault tolerance happens within it: each iteration of the loop calls fault tolerant operations. Thus, this loop can be trivially parallelized.

[2]This is true if the error occurs immediately after that particular entry of $A_c(\tilde{\mathcal{I}}_r, 0 : k_c - 1)$ is loaded into the register, but before it is used, and persists as long as that data item is in the register. If the error occurs when it value already loaded in the register and used in part of the update of $C_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$, then the corrupted data will only appear in a few elements of the same row of the latter. A similar comment holds for the errors in the registers holding $B_c(0 : k_c - 1, \tilde{\mathcal{J}}_r)$ and the columns of $\hat{C}_c(\tilde{\mathcal{I}}_r, \tilde{\mathcal{J}}_r)$.

Loop 3. If Loop 3 is parallelized, then all the threads compute the checksum $d_b := -B_c w$ during the operation that packs B_c. This packing is performed in parallel by the BLIS framework. Therefore, it is important that the fused pack and GEMV operation are parallelized along rows; if the operation is instead parallelized along columns, each thread will update all of the entries of d_b, requiring either a reduction or mutex synchronization. Now, each iteration of Loop 3 uses different parts of A, different parts of C, and performs its own checksums using parts of A and C. Furthermore each iteration performs its own independent recomputation of C if an error has occurred. This means that no other race conditions are introducing during the update of checksum vectors, except that, if Loop 3 is parallelized, each thread must have its own independent checksum vectors d and e^T.
Loop 2. If Loop 2 is parallelized, then all threads collaborate in the concurrent packing of B_c, as discussed for Loop 3, as well as the operations that pack the $m_c \times k_c$ block of A_c and perform the checksum operations $d := A_c d_b$ and $e_a^T := -v^T A_c$. Notice that there are two dimensions along which the loops iterate: m_c and k_c. Each iteration of the loop along the m_c dimension updates the entire vector e_a^T, and each iteration of the loop along the k_c dimension updates the entire vector d. Thus, there are no independent dimensions over which to parallelize this operation, and either a reduction or a mutex is required for updating either e_a^T or d, depending on which loop is parallelized. Because of this, one ancillary benefit of computing the left checksum only if an error is detected by the right checksum is that the synchronization when updating e_a^T can be avoided. If only the right checksum is computed during the packing of A_c, then the loop that iterates over m_c can be parallelized.

Next, notice that each iteration of Loop 2 updates the entire vector d inside of the micro-kernel. Thus, the iterations of this loop are still dependent, and either a mutex or a reduction must be employed when updating d. There are no other potential race conditions when updating checksum vectors, as each iteration of Loop 2 updates different parts of e^T inside of the micro-kernel. Furthermore, each iteration of Loop 2 updates a different part of C. When parallelizing Loop 2, notice that each thread parallelizes the recomputation of C if an error has occurred.
Loop 1. If Loop 1 is parallelized, all thread will participate in the concurrent packing of B_c and associated operations, as discussed in the bullet point on Loop 3. Also, all thread will collaborate to pack of A_c and associated operations and recompute C, as discussed in the bullet on Loop 2.

In addition, each iteration of this loop updates the entire checksum vector e^T, introducing potential race conditions that must be avoided, but this is not an issue if the left checksums are performed lazily.
Summary. Loop 5 can always be trivially parallelized, and Loop 3 can be parallelized as long as care is taken when packing B_c. When parallelizing Loop 2, care must be taken to avoid race conditions when updating d inside of the micro-kernel, and care must be taken to avoid race conditions when packing A_c. When parallelizing Loop 1, care must be taken to avoid race conditions when updating e^T inside of the micro-kernel, but this is not an issue when the lazy left checksum optimization is employed.

Load imbalance caused by error correction. Delays and load imbalance among threads may occur if one or more threads encounter errors and those threads recompute while other threads have to wait. If the recomputation is done as soon as errors are detected this happens within Loop 3 of our algorithm, at the end of each iteration. If the lazy left checksum is computed, this operation will also take place within Loop 3, just before the recomputation is performed. If loops 3, 4, or 5 are parallelized and, for example, a single error occurs, then one thread will perform the recomputation, while the remaining threads will have to wait for it to finish. On the other hand, if loops 1 or 2 are parallelized, even if only a single thread encounters an error, all the threads parallelizing those loops will participate in the recomputation.

It is desirable to parallelize loops 3 and 5, and to do so without introducing load imbalance upon error recovery overhead. To this end, we introduce the *lazy recomputation* of the parts of C that were calculated incorrectly. The goal of the lazy recomputation is to "push" the recomputation to the outer loops, instead of performing it inside Loop 3. Thus the goal is to delay performing the recomputation of parts of C += AB, so that more threads can participate in the recomputation. The advantage of this approach is better load balancing. In contrast to lazy recomputation, we will refer to the scheme that performs recomputation as soon as errors are detected as the *greedy recomputation*.

There are a couple of weaknesses to this lazy recomputation approach. First, the lazy left checksum, the identification of where the errors happened, the repacking of B_c, and the roll-back of the C to $\check{C}$ are still performed immediately upon encountering an error and this is done inside of Loop 3. Thus there is still some load imbalance that may occur, since these operations will not be performed in cooperation by all threads and some load imbalance will still occur. Second, a disadvantage of the lazy recomputation is that putting off the recomputations for later means that they must be performed cold. That is, when the error is first detected, A_c is still in the L2 cache, and B_c is still in the L3 cache (However it is possible that one or both of A_c or B_c has corrupted elements and must be repacked). By performing the lazy recomputation later, A or B will no longer be packed and will no longer be in cache, so the recomputation will be less efficient.

Swapping the two outermost loops. If the lazy recomputation is performed after Loop 3, then it is possible to remove any load unbalance caused by recomputation that may arise when parallelizing Loop 3, but this load imbalance will still occur when Loop 5 is parallelized instead. Thus it is desireable to perform the lazy recomputation outside of Loop 5, however this means that the recomputation will be done across multiple rank-k updates. This introduces a couple of small challenges. First, the recomputations from different rank-k udpates will happen concurrently, leading to potential race conditions if an element of C must be recomputed across multiple rank-k updates. Second, BLIS scales C by β inside of the micro-kernel, so the first rank-k update uses the β passed in by the user invoking GEMM, and subsequent rank-k updates use $\beta = 1$. This can create difficulties because C must first be scaled by β before it is updated, and each rank-k update uses a different β.

There are several solutions to the above problems but we believe the simplest is to swap loops 4 and 5. Proceeding in this manner, we can parallelize the outermost loop over the n dimension, while performing the lazy recomputation after it has finished, all while doing all of this within a single rank-k update to avoid the above issues. This yields an algorithm that slightly deviates from the one used in the GotoBLAS approach [10], but the resulting algorithm is still one that belongs to the family of high performance algorithms that follows a locally optimal cache blocking scheme investigated in [9]. The main difference is that the next panel of B to work on is the one next to the current panel of B rather than the one below. We do not expect this to have an appreciable impact on performance.

6 Experiments

We report results on a dual socket system with two eight core Sandybridge (SNB) processors as well as an Intel Xeon Phi Knight's Corner (KNC) coprocessor.

Implementation. The described mechanisms were added to the BLIS framework, using modified versions of the standard micro-kernels and the blocking parameters reported in [16]. As mentioned, the standard implementations of BLIS are highly competitive.

The following decisions were made regarding the implementation of the detection and correction mechanism: (1) Our implementation handles the extended GEMM operation using the approach that checkpoints C before updating it, as described at the end of Section 4, because that approach requires less bandwidth from main memory when no error occurs; (2) The implementation always uses the lazy left checksum approach because performance results suggest that the cost of error detection outweighs the cost of error

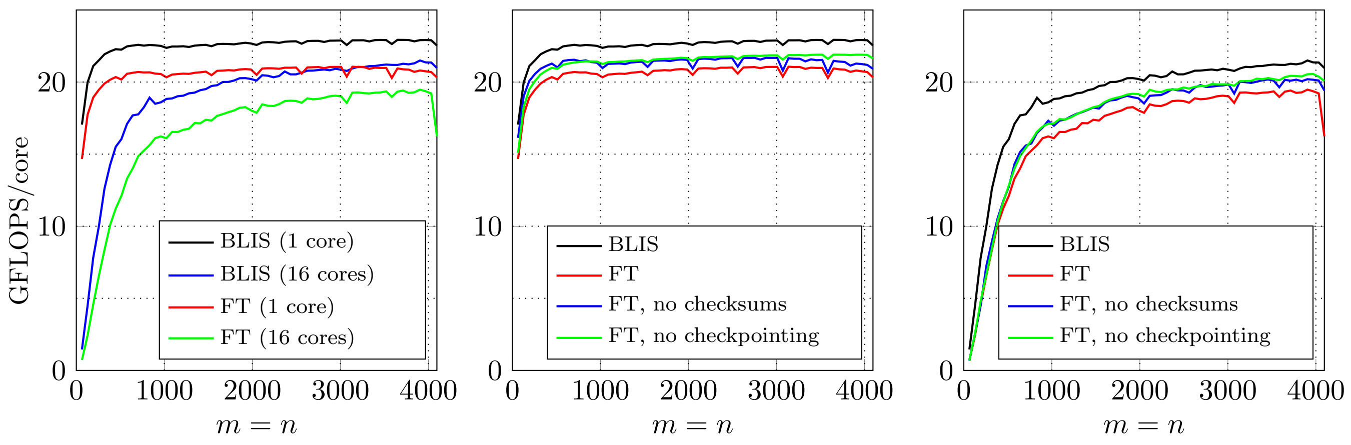

Figure 5: Performance of BLIS vs. prototype implementation of the routine with an embedded fault-tolerant mechanism on an Intel Xeon E5-2680, when no errors are introduced. k is fixed to 256 for all three graphs. **Left:** Cost of detecting errors; **Middle:** Breakdown of costs (1 core); **Right:** Breakdown of costs (16 cores).

correction in most cases; (3) We swap Loops 4 and 5 to place the lazy recomputation outside of the outermost loop over the n dimension for load balancing reasons. With these decisions, the algorithm we use for our experiments is shown in Fig. 4. We note that we did not take the precaution mentioned in Section 4 to prevent false negatives for the Intel Xeon E5-2680, but for the KNC, A and B were read from the assumed to be protected L1 cache twice. This had very little impact on performance.

How to read the graphs. The tops of the graphs represent theoretical peak performance. In some graphs we use problem sizes with fixed k, varying m and n, where k is the algorithmic block size, k_c. Such GEMM cases are often encountered in, for example, LAPACK implementations. Furthermore, our fault tolerant mechanisms occur entirely within each rank-k update, so the costs of the fault-tolerance do not change significantly once k is greater than or equal to $k_c = 256$ (see Fig. 3), making it more interesting to see how the costs of performing the checksums change with varying m and n. The rate of computation in GFLOPS uses the standard $2mnk$ flop count.

Experimental results for a conventional CPU. We now examine the overheads of our fault-tolerant BLIS implementation on SNB. We first examine the costs associated with error detection when no errors occur. In Fig. 5 we report the overhead of the error detection mechanism (labeled with "BLIS-FT") when no errors are introduced, both for a single core and for two SNB processors with eight cores each. These overheads result from the computation of the checksum vectors and the checkpointing of C. This shows that the overhead of fault-tolerance is in the 10% range for both the single and multi-threaded cases, once the problem size is reasonably large. Fig. 5 also breaks those costs down further, showing the overhead if either the checksum vectors are computed or the checkpointing of C is performed. This graph demonstrates that these two costs involved in error detection are roughly equal. Furthermore, this is the case both in the single-threaded case and the multi-threaded case, so performing the checkpointing of C is not causing the GEMM operation to be bandwidth limited, at least with 16 threads.

We will now examine the effects of performance when one or more errors are introduced into various stages of the computation. In Fig. 6 (left), we illustrate the impact on performance when between one and ten errors are artificially introduced into matrix C and then detected and corrected, with a single thread. These errors were injected in such a way that there were no false positives in recomputation. The

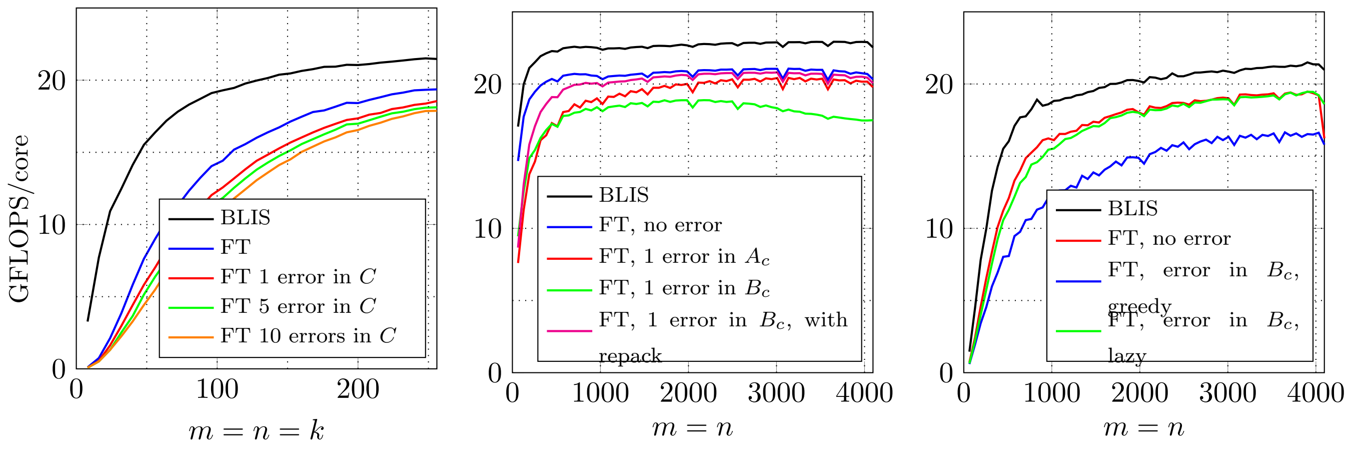

Figure 6: **Left:** Performance of BLIS vs. prototype implementation of the routine with an embedded fault-tolerant mechanism on an SNB, when between 1 and 10 errors are injected into C during computation. **Middle:** Impact on performance when an error is encountered in A_c or B_c. k is set to 256. Repacking B_c reduces the impact of the error on performance because it reduces the number of entries in C that need to be recomputed. **Right:** Impact on performance when correcting a single error encountered when packing B_c. k is set to 256. The error is only encountered by a subset of the threads, and this may result in load imbalance. The lazy recomputation helps alleviate this load imbalance when multiple errors are encountered.

performance impact for recomputation is relatively low even for small matrices. The gap between the curve with no errors and the curve with a single introduced error is greater than the gap between the single error curve and the 5 error curve, and it is still greater than the gap between the single error cuve and the 10 error curve. That this is true, even though the 10 error curve has ten times more work to do during rollback and recomputation, suggests that other costs that are associated with detecting an error can be more costly than recomputation. The largest such cost is the left checksum.

In Fig. 6 (middle), we report the performance impact when a single error is artificially introduced during a packing routine, and then detected and corrected after that error propagates, corrupting potentially hundreds of elements of C. If an error is introduced into A_c, this will corrupt $min(n, n_c)$ elements of C that must be rolled back and recomputed. In this case, an entire row of C will be corrupted, since $n_c = 4096$, the maximum size for n in this experiment. This is illustrated in the curve labeled "FT, 1 error in A_c". Next, if an error is introduced in B_c, this could potentially corrupt an entire column of C. This is illustrated in the curve labeled "FT, 1 error in B_c". Now the amount of recomputation that must occur changes with the problem size. Furthermore, it will cause errors in many macro-kernels, and each time a macro-kernel with error is performed, the left checksum must be performed and the locations of these errors must be determined and recorded since the recomputation is performed lazily. We believe it is these operations that account for the difference in performance between the lines labeled "FT, 1 error in A_c" and "FT, 1 error in B_c", even though at the right-most datapoint in the graph they have the same amount of rollback and recomputation. It is possible to detect if an error has occurred in B_c after a single macro-kernel execution, and then repack B_c if this has happened. In this case, a maximum of $m_c = 96$ elements may be corrupted. This is illustrated in the curve labeled "FT, 1 error in B_c, with repack". A large performance advantage is seen due to this optimization of repacking B_c, as fewer errors must be corrected, despite the extra time spent in repacking B_c. It may be possible to repack only select elements of B_c if an error is detected in it, but in this experiment, we repacked the entire B_c.

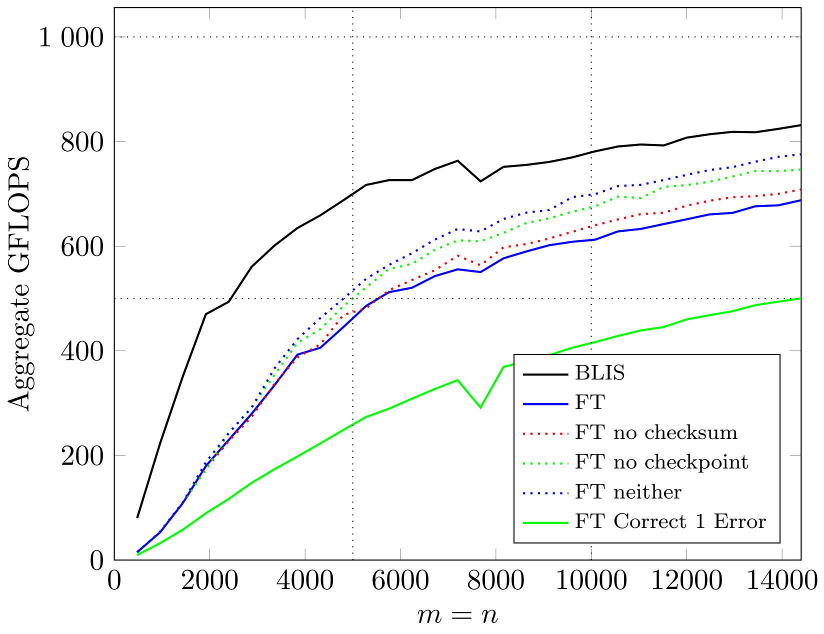

Figure 7: Performance of BLIS and a prototype implementation of the routine with an embedded fault-tolerant mechanism on KNC. Partial fault tolerant implementations that show the breakdown of the costs of the fault-tolerant mechanism are dotted. Performance when one error is introduced in the computation of C is also shown.

In Fig. 6 (right), we report performance when introducing errors into B_c using both lazy and greedy recomputation. Here we do not repack B_C to demonstrate the case where many errors are encountered by one or only a few of the threads. In this experiment, we parallelized Loop 5 with eight threads, and Loop 3 with two threads. Thus, B_c will be shared by each pair of threads parallelizing Loop 3, and corrupt a single column of C. With greedy recomputation, only one pair of threads will perform the recomputation of that corrupted column, while the other threads wait. In contrast, with lazy recomputation all eight pairs of threads will perform that recomputation, yielding much better performance. This demonstrates that lazy recomputation can be very effective in reducing the load imabalance that may occur when few threads encounter many errors.

Experimental results for a many-core coprocessor. We now examine the overheads of our fault-tolerant BLIS implementation on KNC. In Fig. 7, we show the regular BLIS implementation of GEMM, alongside a prototype fault-tolerant implementation. In both implementations, we used 60 cores[3], parallelizing Loop 3 with 15 threads, and Loop 2 with 16 threads. (Each core must use four hyperthreads for high performance.) This graph demonstrates that our error detection mechanisms scale to a many-core architecture. We break down performance further and show that the overhead of performing checksums is relatively minor, and the overhead of performing checkpointing is even smaller. It is interesting that performing checkpointing can have such a low overhead on an architecture with so many threads, despite the extra bandwidth to main memory that this entails.

It was important to use streaming store instructions such as `vmovnrapd` during the checkpointing to attain high performance. The curve "FT neither" is the prototype fault-tolerant implementation of GEMM

[3]Our KNC has 61 cores, however 60 cores are usually employed in practice.

with neither checkpointing nor checksums performed. The fact that this curve is much slower than the normal BLIS implementation suggests that there are inefficiencies in our foult-tolerant prototype (e.g. extra barriers and memory allocations) that can be removed, and thus performance can be closer to that of the regular implementation, especially for smaller problem sizes.

In this graph, we also report the performance exhibited when we introduce errors during the execution of a single micro-kernel. In our experiment, we corrupted an 8×1 row of a single micro-tile of C during computation. Our prototype implementation performs recomputation on the granularity of a micro-kernel, and so a single $8 \times 30 \times 240$ matrix multiplication was performed for the recomputation. This recomputation was performed by a single thread, and it was implemented with an inefficient triply nested loop, so our implementation of it can still be improved. However this was an extremely small amount of computation in the context of a $14400 \times 14400 \times 240$ GEMM. This suggests that while our lazy recomputation solution to load imbalance problems works in the context of a multi-core system where one core encounters many errors, it does not help load imbalance problems in the context of a many-core system where only one error is encountered. This indicates that we should use dynamic parallelism to solve this load imbalance.

7 Conclusion

In this paper, we have analyzed how algorithm-based fault-tolerance can be integrated into a modern framework, BLIS, for the sequential and multi-threaded implementation of matrix-matrix multiplication. A key observation for attaining high performance is that the matrix-vector multiplications that underlies the computations of checksum vectors can be done simultaneous with the packing of data that is inherent in the BLIS framework. This overcomes the need for bringing data in from main memory more often than already necessary for the standard implementation of GEMM. Other innovations include a technique to increase load balancing by performing recomputations lazily, performing checkpointing while C is in registers to reduce the bandwidth requirements, and the technique of lazily performing some error detection operations to reduce the overhead of error detection.

The presented work leaves much work to be done. The BLIS framework implements all BLAS. Importantly, how BLIS implements the other matrix-matrix operations (level-3 BLAS) closely resembles how GEMM is implemented. Thus, the insights we give in this paper likely extend to fault-tolerant implementations of these other closely related operations. Beyond this, the level-3 BLAS are used in higher level matrix operations like the various decompositions. Can the techniques be extended to these? How can fault-tolerance be included in the other parts algorithms for implementing these higher level operations, or applications that utilize BLAS?

We made severe restrictions on where we allow errors to be introduced. Errors can only occur when data is either in registers or while it is moving through the CPU's functional unit, and we assume all data in caches and main memory is safe. Furthermore, we only consider errors that occur during the execution of the micro-kernel or during the packing routines. We did not consider errors that can occur during checkpointing, roll-back, or recomputation. Expanding upon this work may lead to a robust, fault-tolerant BLAS library, which would be an important component of a more comprehensive fault-tolerant solution.

Acknowledgments

This work was supported in part by the National Science Foundation under Award No. ACI-1148125/1340293. Access to the Stampede supercomputer administered by TACC is gratefully acknowledged. Visits by E. S. Quintana-Ortí were supported in part by an ICES J.T. Oden Faculty Fellowship. He was also supported by project CICYT TIN2014-53495-R.

References

[1] Kuang-Hua Huang et al. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6), 1984.

[2] Zizhong Chen et al. Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans. Parallel Distrib. Syst.*, 19(12), 2008.

[3] E. Anderson et al. *LAPACK Users' guide (third ed.)*. SIAM, 1999.

[4] Field G. Van Zee et al. The libflame library for dense matrix computations. *IEEE CiSE*, 11(6), 2009.

[5] L. S. Blackford et al. *ScaLAPACK Users' Guide*. SIAM, 1997.

[6] Jack Poulson et al. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Soft.*, 39(2), 2013.

[7] Panruo Wu et al. Fault tolerant matrix-matrix multiplication: Correcting soft errors on-line. In *ScalA '11*, New York, NY, USA, 2011. ACM.

[8] J.A. Gunnels et al. Fault-tolerant high-performance matrix multiplication: theory and practice. In *DSN 2001*, 2001.

[9] John A. Gunnels et al. A family of high-performance matrix multiplication algorithms. In *ICCS '01*, 2001.

[10] Kazushige Goto et al. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3), 2008.

[11] Xianyi Zhang et al. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *SC12*, 2012.

[12] Intel. Math Kernel Library. `https://software.intel.com/intel-mkl`.

[13] AMD. AMD Core Math Library. `http://developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/`.

[14] IBM. Engineering and Scientific Subroutine Library. `http://www-03.ibm.com/systems/power/software/essl/`.

[15] Field G. Van Zee et al. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Soft.*, 41(3), 2015.

[16] Field G. Van Zee et al. The BLIS framework: Experiments in portability. *ACM Trans. Math. Soft.*, To appear.

[17] Tyler M. Smith et al. Anatomy of high-performance many-threaded matrix multiplication. In *IPDPS'14*, 2014.

[18] Greg Bronevetsky et al. Soft error vulnerability of iterative linear algebra methods. In *ICS'08*, 2008.

[19] S.S. Mukherjee et al. The soft error problem: an architectural perspective. In *HPCA'11*, Feb 2005.

[20] R. Clint Whaley et al. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2), 2001.

[21] Michael Turmon et al. Software-implemented fault detection for high-performance space applications. In *DSN 2000*, 2000.