FT-BLAS: A Fault Tolerant High Performance BLAS Implementation on x86 CPUs

Yujia Zhai, Elisabeth Giem, Kai Zhao, Jinyang Liu, Jiajun Huang,
Bryan M. Wong, Christian R. Shelton, and Zizhong Chen

Abstract—Basic Linear Algebra Subprograms (BLAS) serve as a foundational library for scientific computing and machine learning. In this paper, we present a new BLAS implementation, FT-BLAS, that provides performance comparable to or faster than state-of-the-art BLAS libraries, while being capable of tolerating soft errors on-the-fly. At the algorithmic level, we propose a hybrid strategy to incorporate fault-tolerant functionality. For memory-bound Level-1 and Level-2 BLAS routines, we duplicate computing instructions and re-use data at the register level to avoid memory overhead when validating the runtime correctness. Here we novelly propose to utilize mask registers on AVX512-enabled processors and SIMD registers on AVX2-enabled processors to store intermediate comparison results. For compute-bound Level-3 BLAS routines, we fuse memory-intensive operations such as checksum encoding and verification into the GEMM assembly kernels to optimize the memory footprint. We also design cache-friendly parallel algorithms for our fault-tolerant library. Through a series of architectural-aware optimizations, we manage to maintain the fault-tolerant overhead at a negligible order ($<$3%). Experimental results obtained on widely-used processors such as Intel Skylake, Intel Cascade Lake, and AMD Zen2 demonstrate that FT-BLAS offers high reliability and high performance – faster than Intel MKL, OpenBLAS, and BLIS by up to 3.50%, 22.14%, and 21.70%, respectively, for both serial and parallel routines spanning all three levels of BLAS we benchmarked, even under hundreds of errors injected per minute.

Index Terms—BLAS, SIMD, Assembly Optimization, Dual Modular Redundancy, Algorithm-Based Fault Tolerance, AVX-512, AVX2, OpenMP, Parallel Algorithm

1 Introduction

Due to common performance-enhancing technologies such as shrinking transistor width, higher circuit density, and lower near-threshold voltage operations, processor chips are more susceptible to transient faults than ever before [1], [2], [3]. Transient faults can alter a signal transfer or corrupt the bits within stored values instead of causing permanent physical damage [4], [5]. As a consequence, reliability has been identified by the U.S. Department of Energy as one of the major challenges for exascale computing [6].

The academic and industry communities have observed significant effects of transient faults since the first transient error and resulting soft data corruption was observed by Intel Corporation in 1978 [7]. Sun Microsystems reported in 2000 that server crashes caused by cosmic ray strikes on unprotected caches were responsible for the outages of random customer sites including America Online, eBay, and others [8]. In 2003, Virginia Tech demolished the newly-built Big Mac cluster of 1100 Apple Power Mac G5 computers into individual components and sold them online because the cluster was not protected by error correcting code (ECC) and fell prey to cosmic ray-induced partial strikes, causing repeated crashes and rendering it unusable [9]. Transient faults can still threaten system reliability even if a cluster is protected by ECC: Oliveira et al. simulated an exascale machine with 190,000 cutting-edge Xeon Phi processors, which could still experience daily transient errors under ECC protection [10]. Not only being reported in simulations, Google experienced transient faults in a real-world production environment that provided incorrect results [11]. To mitigate the negative impact of transient faults on large-scale infrastructure services, Meta also started their internal investigation to address this issue in 2018 [12].

Transient faults can be grouped into two categories according to the outcome. If an affected application crashes when a transient fault occurs, it is a fail-stop error. If the affected application continues but produces incorrect results, it is called a fail-continue error. Fail-stop errors can often be protected by checkpoint/restart mechanisms (C/R) [13], [14], [15], [16] and algorithmic approaches [17], [18], [19]. Fail-continue errors are often more dangerous because they can corrupt application states without any warning from the system and lead to incorrect computing results [20], [21], [22], [23], [24], which can be catastrophic under safety-critical scenarios [25]. In this paper, we restrict our scope to fail-continue errors from computing logic units (e.g., 1+1=3), assuming fail-stop errors are protected by checkpoint/restart and memory errors are protected by ECC. In what follows, we will use *soft errors* to denote such fail-continue errors from computing logic units.

Dual modular redundancy (DMR) is an approach to handle soft errors. Typically assisted by compilers, DMR duplicates computing instructions and inserts check instructions into the original programs [26], [27], [28], [29], [30].

- *Yujia Zhai, Elisabeth Giem, Jinyang Liu, Jiajun Huang, Bryan M. Wong, Christian R. Shelton, and Zizhong Chen are affiliated with the University of California, Riverside, CA 92521. Kai Zhao is affiliated with University of Alabama at Birmingham, Birmingham, AL 35294.* *A shortened version of this paper was presented at ACM International Conference on Supercomputing 2021 and was published in its proceedings. The open-source FT-BLAS can be downloaded at https://github.com/yzhaiustc/ftblas.*

DMR is very general and can be applied to any application, but it introduces a high overhead especially for computing-bound applications because it duplicates all computations. To reduce fault tolerance overhead, algorithm-based fault tolerance (ABFT) schemes have been developed for many applications in recent years. Huang and Abraham proposed the first ABFT scheme for matrix-matrix multiplication [31]. Sloan et al. proposed an algorithmic scheme to protect conjugate gradient algorithms for sparse linear systems [32]. Sao and Vuduc explored a self-stabilizing FT scheme for iterative methods [33]. Di and Cappello proposed an adaptive impact-driven FT approach to correct errors for a series of real-world HPC applications [34]. Chien at al. proposed the Global View Resilience system, a library that enables applications to add resilience efficiently [35]. Many other FT schemes have been developed for widely-used algorithms such as sorting [36], fast Fourier transforms (FFT) [37], [38], [39], iterative solvers [40], [41], [42], and convolutional neural networks [43]. Recently, the interplay among resilience, power, and performance has been studied [44], [45], [46], revealing the strong correlation among these key factors in HPC.

Although numerous efforts have been made to protect scientific applications from soft errors, most routines in the Basic Linear Algebra Subprograms (BLAS) library remain unprotected. The BLAS library is a core linear algebra library fundamental to a broad range of applications, including weather forecasting [47], deep learning [14], [16], and computational chemistry simulations [48]. Because of this pervasive usage, academic institutions and hardware vendors provide a variety of BLAS libraries such as Intel MKL [49], AMD ACML, IBM ESSL, ATLAS [50], BLIS [51], and OpenBLAS [52] to pursue extreme performance on a variety of hardware platforms. BLAS routines are organized into three levels: Level-1 (vector/vector), Level-2 (matrix/vector), and Level-3 (matrix/matrix). [53]. Except for the general matrix-matrix multiplication (GEMM) routine, which has been extensively studied [31], [54], [55], [56], [57], minimal research has concentrated on protecting the rest of the BLAS routines.

For the general matrix-matrix multiplication routine, several fault tolerance schemes have been proposed to tolerate soft errors with low overhead [31], [54], [55], [57]. The schemes in [31] and [55] are much more efficient than DMR. However, these two schemes are offline schemes that cannot correct errors in the middle of a computation in a timely manner. In [54], Wu et al. implemented a fault-tolerant GEMM that corrects soft errors online. However, built on third-party BLAS libraries, this ABFT scheme becomes less efficient when using AVX-512-enabled processors because the current gap between computation and memory transfer speed becomes so large that the added memory-bound ABFT checksum computation is no longer negligible relative to the original computing-bound GEMM routine. In [57], Smith et al. proposed a fused ABFT scheme for BLIS GEMM at the assembly level to reduce the overhead for checksum calculations. An in-memory checkpoint/rollback scheme was used to correct multiple simultaneous errors online. Although this scheme provides wider error coverage, it presents a moderate overhead "in the range of 10%" [57].

When projecting a BLAS implementation to real-world deployment, enabling support for parallel multi-core systems, as well as for a variety of mainstream micro-architectures, such as AVX-512 and AVX2 extensions, can both be crucial. Compared with AVX-512-enabled processors, an AVX2-enabled-only processor exposes halved vectorized registers to a user and, consequently, significantly higher register pressure when designing performance-oriented fault-tolerant algorithms. In addition to providing delicate assembly-level optimizations on computing kernels, one should propose a cache-friendly design for parallel Level-3 BLAS routines [58].

In this paper, we develop FT-BLAS—the first BLAS implementation that not only corrects soft errors online, but also provides at least comparable performance to modern state-of-the-art BLAS libraries such as Intel MKL, OpenBLAS, and BLIS. Our FT-BLAS provides superior performance and maintains negligible overhead on both AVX-512 and AVX-2-enabled x86 processors with multi-thread support. FT-BLAS not only protects the GEMM general matrix-matrix multiplication routine but also protects other Level-1, Level-2, and Level-3 routines. BLAS routines are widely used in many applications from a variety of fields; therefore, improvements to the BLAS library will benefit not only a large number of users but also a broad cross-section of research areas. The main contributions of this paper include:

- We develop a new implementation of BLAS using AVX-512 assembly instructions that achieves comparable or better performance than the latest versions of OpenBLAS, BLIS, and MKL on AVX-512-enabled processors such as Intel Skylake and Cascade Lake.
- We benchmark our hand-tuned BLAS implementation on an Intel Skylake processor and find that it is faster than the open-source OpenBLAS and BLIS libraries by 3.85%-22.19% for DSCAL, DNRM2, DGEMV, DTRSV, and DTRSM, and comparable performance ($\pm 1.0\%$) for the remaining selected routines. Compared to the closed-source Intel MKL, our implementation is faster by 3.33%-8.06% for DGEMM, DSYMM, DTRMM, DTRSM, and DTRSV, with comparable performance in the remaining benchmarks.
- We build FT-BLAS, the first fault-tolerant BLAS library, on our new BLAS implementation by leveraging the hybrid features of BLAS: adopting a DMR strategy for memory-bound Level-1 and Level-2 BLAS routines and ABFT for computing-bound Level-3 BLAS routines. Our fault-tolerant mechanism is capable of not only detecting but also correcting soft errors online during computation. Through a series of low-level optimizations, we manage to achieve a negligible (0.35%-3.10%) overhead.
- We provide multi-thread AVX-512-enabled implementations for BLAS routines (DDOT, DNRM2, DGEMV, DGEMM) and benchmark their parallel performance on an Intel Cascade Lake processor. Experimental results validate that our fault-tolerant designs maintain a negligible overhead (0.16%-3.53%), and the performance with the FT capability remains comparable to or faster than reference libraries.
- We extend FT-BLAS with AVX2-instruction support.

We benchmark four representative routines (DNRM2, DGEMV, DTRSV, and DGEMM) on an AMD R7 3700X processor. Experimental results validate that FT-BLAS maintains its high performance and low overhead on this AVX2-enabled AMD processor.
- We evaluate the performance of FT-BLAS under error injection on Intel Skylake, Intel Cascade Lake, and AMD Zen2 processors. Experimental results demonstrate FT-BLAS maintains a negligible performance overhead under hundreds of errors injected per minute while outperforming state-of-the-art BLAS implementations such as OpenBLAS, BLIS, and Intel MKL by up to 22.14%, 21.70%, and 3.50% respectively—all of which cannot tolerate any errors.

The rest of the paper is organized as follows: We introduce background and related works in Section II and then describe how we achieve higher performance than state-of-the-art BLAS libraries in Section III. Section IV and Section V present the design and optimization of our fault-tolerant schemes. Evaluation results are given in Section VI. We present our conclusions and future work in Section VII.

2 Related Work and Background

2.1 Algorithm-Based Fault Tolerance

Algorithmic approaches to soft error protection for computing-intensive or iterative applications have achieved great success [37], [40], [56], [57], [59], [60], [61], [62], [63], [64], [65], ever since the first algorithmic fault tolerance scheme for matrix/matrix multiplication in 1984 [31]. The basic idea is that for a matrix-matrix multiplication $C = A \cdot B$, we first encode matrices into checksum forms. Denoting e=$[1, 1, \ldots, 1]^T$, we have $A \xrightarrow{encode} A^c := \begin{bmatrix} A \\ e^T A \end{bmatrix}$ and $B \xrightarrow{encode} B^r := \begin{bmatrix} B & Be \end{bmatrix}$. With A^c and B^r encoded, we automatically have:

$$C^f = A^c \cdot B^r = \begin{bmatrix} C & Ce \\ e^T C & \end{bmatrix} = \begin{bmatrix} C & C^r \\ C^c & \end{bmatrix}.$$

The correctness of the multiplication can be verified by checking the matrix C against C^r and C^c. Any disagreements, that is, if the difference exceeds the round-off threshold, indicate errors occurred during the computation. The cost of checksum encoding and verification is $O(n^2)$, negligible compared to the $O(n^3)$ of matrix multiplication algorithms and thus ensures lightweight soft error detection for matrix multiplication. For any arbitrary matrix multiplication algorithm, correctness can be verified at the end of the computation (offline) via the checksum relationship.

The previous ABFT scheme can be extended to outer-product matrix-matrix multiplication and the checksum relationship can be maintained during the middle of the computation:

$$C^f = \sum_s A^c(:, s) \cdot B^r(s, :) = \sum_s \begin{bmatrix} C_s & C_s e \\ e^T C_s & \end{bmatrix},$$

where s is the step size of the outer-product update on matrix C, and C_s represents the result of each step of the outer-product multiplication $A^c(:, s) \cdot B^r(s, :)$. Noting this outer-product extension, Chen et al. proposed correcting errors for GEMM online with a double-checksum scheme [56]. The offline version of the double-checksum scheme can only correct a single error in a full execution, while the online version, which corrects a single error for *each* step of the outer-product update, is able to handle multiple errors for the whole program. A checkpoint-rollback technique can also be added to overcome a many-error scenario. In [57], once errors, regardless of how many, are detected via the checksum relationship, the program restores from a recent checkpoint to correct the error. A typical solution to mitigate the memory cost is to fuse the memory footprint with compute kernels [66], [67], [68]. In this paper, we target a more lightweight error model and correct one error in each verification interval using online ABFT without checkpoint/rollback by adopting the kernel-fusion strategy for the sake of performance.

2.2 Duplication-Based Fault Tolerance

Known as dual modular redundancy (DMR), duplication-based fault tolerance is rooted in compiler-assisted approaches and has been widely studied [26], [27], [28], [29], [30]. Classified by the Sphere of Replication (SoR), that is, the logical domain of redundant execution [69], previous duplication-based fault-tolerant work can be grouped into one of three cases:

- Thread Level Duplication (TLD). This approach duplicates the entire processor and memory system: Everything is loaded twice, computed twice, and two copies are stored [26], [27].
- TLD with ECC assumption (TLD+ECC). In this approach, operands are loaded twice but from the same memory address. All other instructions are still duplicated. [28].
- DMR only for computing errors. Only the computing instructions are duplicated and verified to prevent a faulty result from being written back to memory [29], [30].

Different SoRs target different protection purposes and error models. TLD and TLD+ECC lead to the worst performance and memory overheads, but provide the best fault coverage without requiring any other fault-tolerance support such as checkpoint/restarting. Duplicating only the computing instructions shrinks the SoR to soft errors but almost halves the performance loss compared with TLD. We adopt the third SoR, duplication and verification of computing instructions only, in this work.

Since compiler front ends never intrude into the assembly kernels of performance-oriented BLAS libraries, in the few cases that can be found in the compiler literature relating to soft error resilience in BLAS routines [30], the performance is *never* compared against OpenBLAS or Intel MKL, but only to LAPACK [70], a reference implementation of BLAS with much slower performance on modern processors. In this work, we manually insert FT instructions into self-implemented assembly computing kernels for Level-1 and Level-2 BLAS and then hand-tune them for highest performance.

3 Optimizing Level-1, Level-2, and Level-3 BLAS routines

Before adding FT capabilities to BLAS, we first create a new library that provides *comparable or better* performance to

modern state-of-the-art BLAS libraries. We introduce the target instruction set of our work, as well as a sketch of the overall software organization. We then dive into our detailed optimization strategies for the assembly kernel to illustrate how we push our performance from the current state-of-the-art closer to the limits of the hardware. To simplify the presentation, we only present the results for double-precision routines, while all other data types, such as single-precision or complex floating point numbers, share the same optimization techniques as that of double-precision inputs.

3.1 Optimizing Level-1 BLAS

Level-1 BLAS contains a collection of memory-bound vector/vector dense linear algebra operations, such as vector dot products and vector scaling.

3.1.1 Opportunities to Optimize Level-1 BLAS

Software strategies to optimize serial Level-1 BLAS vector routines are typically no more than exploiting data-level parallelism using vectorized instructions: processing multiple packed data via a single instruction, loop unrolling to benefit pipelining and exploit instruction-level parallelism, and inserting prefetching instructions. In contrast to computing-bound Level-3 BLAS routines, where performance can reach about 90% of the theoretical limit, sequential memory-bound routines usually reach 60%-80% saturation because throughput is not high enough to hide memory latency. This fluctuating saturation range makes the experimental determination of underperforming routines difficult. We therefore survey open-source BLAS library Level-1 routines source code with regard to three key optimization aspects: single-instruction multiple-data (SIMD) instruction set support, loop unrolling, and software prefetching. We include double-precision routines in Table 1 for analytical reference.

As seen in Table 1, all Level-1 OpenBLAS routines have been implemented with support for loop unrolling. We also observe the interesting fact that software prefetching, an optimization strategy as powerful as increasing SIMD width for Level-1 routines, is only adopted in legacy implementations of x86 kernels in OpenBLAS. Based on the results of this optimization survey, we optimize two representative routines: we upgrade DNRM2 with AVX-512 support and enable prefetching for DSCAL. In the evaluation section, we show that the performance of our AVX-512-enabled DNRM2 with software prefetching surpasses OpenBLAS DNRM2 (SSE+prefetching) by 17.89%, while our DSCAL with data prefetch enabled via `prefetcht0` obtains a 3.85% performance improvement over OpenBLAS DSCAL (AVX-512 with no prefetch).

TABLE 1
Survey of Selected OpenBLAS Level-1 Routines

AVX-512/AVX2	DDOT, DSCAL, DAXPY, DROT
AVX or earlier	DNRM2, DCOPY, DROTM, IDAMAX, DSWAP
Loop Unrolling	all routines
Prefetching	DNRM2, DCOPY, DROTM, IDAMAX, DSWAP

3.2 Optimizing Level-2 BLAS

Level-2 BLAS performs various types of memory-bound matrix/vector operations. In contrast to Level-1 BLAS, which never re-uses data, register-level data re-use emerges in Level-2 BLAS. We choose the two most typical routines, DGEMV and DTRSV, as examples to explain the theoretical underpinnings of our Level-2 BLAS optimization strategies.

3.2.1 Optimizing DGEMV

DGEMV, double-precision matrix/vector multiplication, computes $y = \alpha op(A)x + \beta y$, where A is an $m \times n$ matrix and $op(A)$ can be A, A^H, or A^T. The cost of vector scaling βy and $\alpha \cdot (Ax)$ is negligible compared with $A \cdot x$; therefore, it suffices for us to consider $\beta = 1$, and $\alpha = 1$, and restrict our discussion to the case $y = Ax + y$, where A is an $n \times n$ square matrix. The naive implementation can be summarized as $y_i = \sum_j^n A_{ij}x_j + y_i$. Since DGEMV is a memory-bound application, the most efficient optimization strategy is to reduce unnecessary memory transfers. It is clear that the previous naive implementation requires n^2 loads for A and x and n^2 loads + stores for y since both x and y sit inside two layers of for loop. No memory transfer operations can be eliminated on matrix A because each element must be accessed at least one time. We must focus on register-level re-use for vectors x and y to optimize DGEMV. We notice that index variable i in $A(i, j)$ is partially independent of the index j of the j-loop, and we can unroll the i-loop R_i times to exploit loading x_j into registers for re-use. Now each load of x_j is reused R_i times within a single register, so the total load operations for x improves from n^2 to n^2/R_i. In practice, R_i is typically between 2-6, because accessing too many discontinuous memory addresses increases the likelihood of translation lookaside buffer (TLB) and row buffer thrashing. We adopt R_i=4 because the longest SIMD ALU instruction (`VFMA`) latency in this loop is 4 cycles [71].

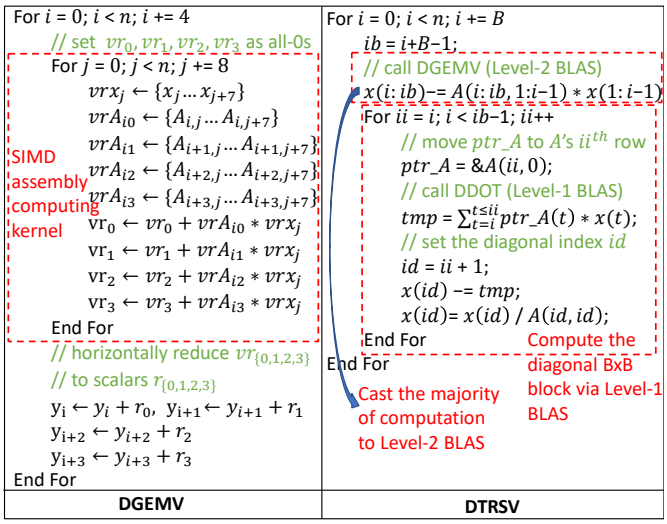


Fig. 1. Optimization schemes of DGEMV and DTRSV.

Unrolling the inner loop (j-loop) improves nothing in terms of load/store numbers but will benefit a SIMD implementation (vectorization). Because both an AVX-512 SIMD register and a cache line of the Skylake microarchitecture

accommodate 8 doubles, we unroll the j-loop 8 times. Before entering the j-loop, four SIMD registers $vr_{\{0,1,2,3\}}$ are initialized to zero. Within the innermost loop body, each x element is still reused R_i times (shown as 4 in Figure 1). We load 8 consecutive x elements into a single SIMD AVX-512 register vrx_j, load the corresponding A elements into SIMD registers vrA_{i*}, and conduct vectorized fused multiplication/addition operations to update vr_*. After exiting the j-loop, vectorized registers vr_* holding temporary results are reduced horizontally to scalar registers, added onto the corresponding y_i, and stored back to memory. Some previous literature [52], [53] suggests blocking for cache level re-use of vector elements. However, this may break the continuous access of the matrix elements, which is the main workload of the DGEMV computation. Hence, we do not adopt a cache blocking strategy in our DGEMV implementation: experimental results validating our DGEMV obtain a 7.13% performance improvement over OpenBLAS.

3.2.2 Optimizing DTRSV

Double-precision triangular matrix/vector solver (DTRSV) solves $x = op(A)^{-1}x$, where A is an $n \times n$ matrix, $op(A)$ can be A, A^H, or A^T, and either the lower or upper triangular part of the matrix is used for computation due to symmetry. We restrict our discussion to $x = A^{-1}x$ using the lower triangular part of A. Since Level-2 BLAS routines are more computationally intensive than Level-1 BLAS routines, we introduce a paneling strategy for DTRSV to cast the majority of the computations — $(n^2 - nB)/2$ elements — to the more computationally-intensive Level-2 BLAS routine DGEMV. The minor $B \times B$ diagonal section is handled with the less computationally-intensive Level-1 BLAS routine DDOT. Given that DGEMV is more efficient, adopting a smaller block size B is preferable since it allows more computations to be handled by DGEMV. Considering the practical implementation of DGEMV, where we unroll the j-loop 4 times for register re-use (shown in Figure 1), the minimal, and also the optimal, block size B should then be 4. In fact, OpenBLAS adopts block size $B = 64$ for DTRSV [72], resulting in more computations handled by the less efficient diagonal routine; this is the major reason our performance exceeds that of OpenBLAS by 11.17%.

It is worth mentioning that our high-performance implementation design for memory-bound routines at best ensures contiguous memory access for matrices and vectors. Therefore, we provide distinct kernels for different matrix layouts (column-major and row-major) of the Level-2 BLAS routines for optimal performance. To simplify the presentation, we always benchmark column-major storage matrices in this paper, and the actual efficiency and performance will not conceptually vary for other layouts.

3.3 Optimizing Level-3 BLAS

3.3.1 Overview of Level-3 BLAS

Level-3 BLAS routines are matrix/matrix operations, such as dense matrix/matrix multiplication and triangular matrix solvers, where extreme cache and register level data reuse can be exploited to push the performance to the peak computation capability. We choose two representative routines, DGEMM and DTRSM to illustrate our implementation and optimization strategies for Level-3 BLAS.

3.3.2 Implementation of DGEMM

We adopt packing and cache-blocking frames similar to OpenBLAS and BLIS. The outermost three layers of the `for` loop are partitioned to allow submatrices of A and B to reside in specific cache layers. The step sizes of these three `for` loops, M_C, N_C, and K_C, define the size and shape of the macro kernel, which are determined by the size of each layer of the cache. A macro kernel updates an $M_C \times N_C$ submatrix of C by iterating over A $(M_R \times K_C)$ multiplying B $(K_C \times N_R)$ in micro kernels. Since our implementation contains no major update on the latest version of OpenBLAS other than selecting different micro-kernel parameters M_R and N_R, nor on the performance ($< \pm 0.5\%$), we do not present a detailed discussion of the DGEMM implementation here but instead refer readers to [51] for more details.

3.3.3 Optimizing DTRSM

DTRSM, a double-precision triangular matrix/matrix solver, solves $B = \alpha \cdot op(A)^{-1}B$ or $B = \alpha B \cdot op(A)^{-1}$, where α is a double-precision scalar, A is an $n \times n$ matrix, $op(A)$ can be A, A^H, or A^T, and either the lower or upper triangular part of the matrix is used for computation due to symmetry. We restrict our discussion to $B = A^{-1}B$ in the presentation of our optimization strategy. We adopt the same cache blocking and packing scheme as DGEMM, but with the packing routine for A and the macro kernel slightly modified. For DTRSM, the packing routine for matrix A not only packs the matrix panels into continuous memory buffers to reduce TLB misses but also stores the reciprocal of the diagonal elements during the packing to avoid expensive division operations in the performance-sensitive computing kernels. When the A_{block} to feed into the macro kernel is on the diagonal, `macro_kernel_trsm` is called to solve $B_{block} := \tilde{A}^{-1} \cdot \tilde{B}$, where $\tilde{A}$ and $\tilde{B}$ are packed matrices. Otherwise, the corresponding B_{block} is updated by calculating B_{block} -= $\tilde{A} \cdot \tilde{B}$, using the highly-optimized GEMM macro kernel. We see that the performance of the overall routine is affected by both macro kernels, and to ensure overall high performance, we must ensure the TRSM kernel is near-optimal as well.

Inside `macro_kernel_trsm`, the B_{block} is calculated by updating a small $M_R \times N_R$ B_{sub} block each time. The B_{sub} block is calculated by B_{sub} -= $\tilde{A}_{curr} \cdot \tilde{B}_{block}$ until A_{curr} reaches the diagonal block. Temporary computing results are held in registers instead of being saved to memory during computation. When A_{curr} is on the diagonal, we solve $B_{sub} := \tilde{A}^{-1}_{curr} \cdot \tilde{B}_{block}$ using an AVX-512-enabled assembly kernel. It should be noted that the packed buffer $\tilde{B}$ needs to be updated during the solve because DTRSM is an in-place update and the corresponding elements of the buffer should be updated during computation. Our highly-optimized TRSM macro kernel grants us a 22.19% overall performance gain on DTRSM over OpenBLAS, where the TRSM macro kernel is an under-optimized prototype.

Different from the memory-bound Level-1 and Level-2 BLAS routines, we do not need to explicitly provide a distinct kernel for different storage layouts for Level-3 BLAS routines. Instead, we only need to provide a different packing routine that stores the needed data in a proper area that maximizes kernel efficiency. In short, the storage layout only changes the packing rather than the assembly

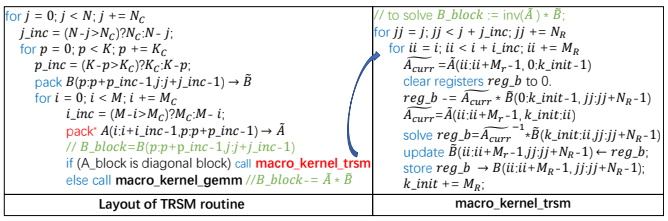


Fig. 2. DTRSM optimization layout.

computing kernels, and also to simplify the presentation, we present the performance of Level-3 BLAS also for all-column-major layouts.

4 Optimizing Fault Tolerant Level-1 and Level-2 BLAS

We first outline our assembly code syntax and duplication scheme. We then show our step-wise assembly optimization of DMR decreases fault tolerance overhead from 50.8% in the scalar version to our 0.35% overhead. After the optimization, the performance of both our FT and non-FT versions surpass both current state-of-the-art BLAS implementations.

4.1 Assembly Syntax and Duplication Scheme

In this paper, all assembly examples follow AT&T syntax; that is, the destination register is in the right-most position. We adopt the most common duplication scheme, DMR [26], [28], [29]. Our chosen sphere of reduction dictates that we duplicate and verify computing instructions instead of memory instructions. More specifically, in our case, most ALU operations are floating point operations. Integer addition/subtraction are used to check whether the loop terminates. We only use two integer registers (%0, %1) throughout our assembly kernels.

4.2 Scalar DMR versus Vectorized DMR

We use DSCAL, one of the most important routines in Level-1 BLAS, to show how even though DMR is slow, it can actually be fast. DSCAL computes $x := \alpha \cdot x$, where x is a vector containing n DPs. DP represents a double-precision data type, so α is also DP.

4.2.1 Scalar scheme

The scalar implementation of DSCAL performs a load (`movsd`), multiplication (`mulsd`), and then a store (`movsd`) operation on scalar elements. The scalar α is invariant within the loop body, so we load it before entering the loop. The array index (stored in register %0) to access array elements is incremented by \$1 before starting the next iteration. Meanwhile, register %1 (initialized by the array length n) is decremented by one to test whether the loop terminates. Once register %1 reaches zero, the EFLAG ZF is set to 1, branch instruction `jnz` will not be taken, and the loop terminates. Because scalar multiplication `mulsd` only supports a two-operand syntax—that is, `mulsd, src, dest` multiplies values from two operands and stores the result in the `dest` register—the value in the `dest` register will be overwritten when the computation finishes. Therefore, we should back up a copy of the loaded value of $x[i]$ into an unused register for use in our duplication to avoid an extra load from memory. After both the original and duplicated computations finish, we check for correctness and set the EFLAGs via `ucomsid`. If two computing results (`xmm1` and `xmm2`) are different, the EFLAG is set as ZF=1, and the branch `jne ERROR_HANDLER` will redirect the control flow to activate a resolving procedure, a self-implemented error handling assembly code. When the correctness of computing is confirmed or an erroneous result is recovered by the error handler, the result $\alpha \cdot x[i]$ is stored in memory.

4.2.2 AVX-512 vectorized scheme

Our AVX-512 vectorized duplication scheme differs from the scalar version in two ways. First, vectorized multiplication supports a three-operand syntax, so source operand registers are still live after computing, and an in-register backup is no longer needed. Second, a comparison between SIMD registers cannot set EFLAGs directly. Therefore, we set EFLAGs indirectly: The comparison result is first stored in an opmask register `k0`, and `k0` is then tested against another pre-initialized opmask register `k1` to set EFLAGs. If two 512-bit SIMD registers with 8 packed DPs are confirmed equal, opmask register `k0`, updated by `vpcmpegd`, will be eight consecutive '1's corresponding to the eight DPs in the comparison. If one (or more) DP(s) from two source operands in comparison are different, the corresponding bit(s) of the opmask register is set to `0`, indicating the erroneous position. We test the comparison result opmask, `k0`, with another opmask, `k1`, pre-initialized to `00000000` via `kortestw`. EFLAG is set to CF=0 first and updated to CF=1 only if the results of OR-ing both source registers (`k0, k1`) are all '`1`'s. Any detected errors will leave CF=0, and the control flow is branched to the error handler by `jnc`.

4.2.3 Performance gain due to vectorization

Our vectorized FT enlarges the verification interval compared to the scalar implementation: The scalar scheme gives a computing/comparison+branch ratio of 1:1, while the vectorized scheme expands this ratio to 8:1, which significantly ameliorates the data hazards introduced by duplication and verification. Our experimental results confirm that vectorization improves the overhead from 50.8% in the scalar scheme to 5.2% in the vectorized version.

4.3 Adding More Standard Optimizations

The peak single-core performance of an Intel processor that supports AVX-512 instructions is 30-120 GFLOPS, whereas the performance of DSCAL is less than 2 GFLOPS. Since CPU utilization is severely bounded by memory throughput, the inserted FT instructions, which do not introduce extra memory queries, should ideally bring a near-zero overhead if computations and memory transfers are perfectly overlapped. This underutilization of CPU performance motivated us to explore optimization strategies to further bring the current 5% overhead to a negligible level.

4.3.1 ***Step 1***: Loop unrolling

Loop unrolling is a basic optimization strategy for loop-based programs. However, it can only reduce a few branch and add/sub integer instructions in practice because CPUs

TABLE 2
DSCAL assembly kernel: scalar and vectorized fault tolerance schemes. %0: the index to access array elements, initialized to 0. %1: the counter to test whether the loop terminates, initialized to n. %2: the address of scalar α. %3: the starting address of vector x.

Original Scalar Instructions	Scalar FT Instructions	Original Vectorized Instructions	Vectorized FT Instructions
movsd (%2), xmm0	movsd (%2), xmm0	vbroadcastsd (%2), zmm0	vbroadcastsd (%2), zmm0
Loop:	Loop:		kxnorw, k1, k1, k1
	movsd (%3, %0, 8), xmm1	Loop:	Loop:
	movsd xmm1, xmm2	vmovupd (%3, %0, 8), zmm1	vmovupd (%3, %0, 8), zmm1
mulsd xmm0, xmm1	**mulsd xmm0, xmm1**	**vmulpd zmm0, zmm1, zmm2**	**vmulpd zmm0, zmm1, zmm2**
	mulsd xmm0, xmm2		vmulpd zmm0, zmm1, zmm3
	ucomisd xmm1, xmm2		vpcmpeqd zmm2, zmm3, k0
	jne ERROR_HANDLER		kortestw k0, k1
movsd xmm1, (%3, %0, 8)	movsd xmm1, (%3, %0, 8)		jnc ERROR_HANDLER
add $1, %0; sub $1, %1	add $1, %0; sub $1, %1	vmovupd zmm2, (%3, %0, 8)	vmovupd zmm2, (%3, %0, 8)
jnz Loop	jnz Loop	add $8, %0; sub $8, %1	add $8, %0; sub $8, %1
		jnz Loop	jnz Loop

automatically predict branches and unroll loops via speculative execution. Possible data hazards caused by speculative execution can be ameliorated by out-of-order execution mechanisms in hardware. Our experiments show that the performance of both our FT and non-FT versions only slightly improves after unrolling the loop 4 times: the overhead decreases from 5.2% to 4.9%.

4.3.2 ***Step 2**: Adding comparison reduction*

Inspired by the previous ten-fold improvement on overhead due to the enlargement of the verification interval, this optimization is naturally focused on the reduction of branch instructions for comparison and diverging to the error handler by leveraging features of the AVX-512 instruction set. Intermediate comparison results are stored in opmask registers and a correct comparison result is stored as "`11111111`" in an opmask register. Therefore, we can propagate the comparison results via `kandw k1, k2, k3`, AND-ing the two intermediate comparison results (`k1`, `k2`), and storing into the third opmask register `k3`. The AND operation ensures that any detected incorrectness marked by "0" in source opmask registers will pollute bit(s) in the destination register during *reduction* and will be kept. Instead of inserting a branch to the error handler for each comparison, only one branch instruction is needed for every 4 comparisons in a loop iteration. This enlargement of the verification interval further decreases the overhead from 4.9% to 2.7%.

4.4 Optimizations Underrepresented in Main Libraries

At this point, we still have not reached optimality. We review possible performance concerns left from the previous step:

- Data hazards. A read-after-write hazard is a true data dependency, and severely impacts this version of the code.
- Structural hazards. Four consecutive store instructions all demand specific AVX-512 units, but there are only two in SkylakeX processors; the instructions stall until the hardware becomes available.

Although out-of-order execution performed by a CPU can avoid unnecessary stalls in the pipeline stage, it consumes hardware resources and those resources are not unlimited. Therefore, we optimize instruction scheduling manually, assuming no hardware optimizations.

4.4.1 *Heuristic software pipelining*

We perform software pipelining to reschedule the instructions *across* the boundary of basic blocks to reduce structural and data hazards. Unfortunately, finding an optimal software pipelining scheme is NP-complete [73]. To simplify the issue, we design the software pipelining heuristically by not considering the actual latency of each type of instruction. To scale eight consecutive elements that can be packed and processed in a AVX-512 SIMD register, we should first load them from memory (L), multiply with the scalar (M1), duplicate multiplication for verification (M2), compare between the original and duplicated results (C), and store back to memory (S) if correct. Stacking these five stages within the loop body causes a severe dependency chain because they all work on the same data stream. To deal with this issue, we first write down the required five stages for a single iteration (L, M1, M2, C, and S) vertically and issue horizontally with a one-cycle latency for two adjacent instruction streams.

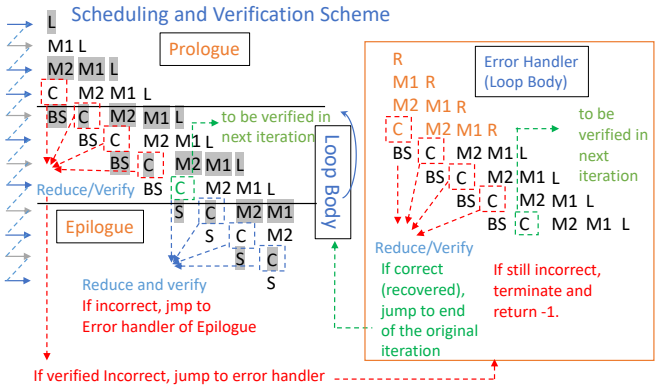


Fig. 3. Software pipelining design. Each letter represents a vectorized instruction. L: Load; M1: Mul; M2: Duplicated Mul; C: Vectorized Comparison; S: Store; BS: Checkpoint original value before scaling into an unused register then Store the computing result back to memory; R: Restore from a checkpoint register.

4.4.2 *Verification reduction and in-register check-pointing*

Since the loop is still unrolled four times, comparison results can be reduced via `kandw` between opmask registers. The next loop iteration will start to execute only if the loop does

not terminate and the correctness of the current iteration is verified. With cross-boundary scheduling, we compute for iterations 2, 3, 4, and 5 but verify iterations 1, 2, 3, and 4. The comparison result of the fifth iteration is only stored and then verified in the next iteration or in the epilogue. Because the memory is updated before the computing results are verified, we checkpoint original elements loaded from memory in an unused register. This operation coalesces the "in-register checkpoint" (B) followed by a store (S) and is denoted by BS when designing our software pipelining. Once an error is detected in the loop body and the recovery procedure is activated, the error handler restarts the computation from a couple of prologue-like instructions where the load is substituted with recovery from the backup registers. The corruption is recovered by a third calculation with duplication so the results must be verified again. If the disagreement still exists, the program is terminated and signals that it is unable to recover. If the recovered computing results reach consensus, the control flow returns back to the end of the corrupted loop iteration and continues as normal.

4.4.3 *Effectiveness of scheduling*

Experimental benchmarks report the latencies of `vmulpd`, `vcmpeqpd`, and `vmovpd` (both store and load) are 4, 3, and 3 cycles (under a cache hit), respectively [71]. After scheduling, operands are consumed after 3 instructions; before our scheduling, these operands were consumed immediately by the following instruction. For structural hazards, according to the Intel official development manual [74], two adjacent vectorized multiplications (M2, M1) can be executed by Port 0 and Port 1, and Port 5 accommodates the following comparison (C) simultaneously. Therefore, three consecutive ALU operations $C, M2$, and $M1$ within the loop body produce no structural hazard concerns. Additionally, Skylake processors can execute two memory operations at the same time so the structural hazard concerns on load and store are also eliminated. Therefore, we confirm that our heuristic scheduling strategy on DSCAL effectively ameliorates the hazards introduced by fault tolerance. We optimize the non-FT version using the same method and compare it with our FT version. Our experimental result demonstrates that software pipelining improves FT overhead from 2.7% to 0.67%.

4.4.4 *Adding software prefetching*

Prefetching data into the cache before it is needed can lower cache miss rates and hide memory latency. Dense linear algebra operations demonstrate high regularity in their memory access patterns, enabling performance improvement via accurate cache prefetching. We can send a prefetch signal *before* data is needed by a *proper prefetch distance*. When the data is actually needed, it has been prefetched into the cache instead of waiting for the approximately 100 ns required to load it from DRAM. Accurate prefetching distance is important. If data is prefetched too early or too late, the cache is polluted and performance can degenerate. Here we select the prefetch distance to be 1024 bits: We prefetch 128 elements in advance into the L1 cache using the `prefetcht0` instruction. Instead of prefetching for all load operations, we only prefetch half of them in the loop body to avoid conflicts with hardware prefetching. Prefetching improves the performance of both our non-FT and FT versions by $\sim 4\%$, and the overhead further decreases from 0.67% to 0.36%. Given the target processor is equipped with DDR4 main memory at 2666 MHz, the theoretical performance of the memory-bound DSCAL routine is 2.666 GHz * 8 Byte * (1 FLOP / 8 Byte) = 2.666 GFLOPS. The best variant of our DSCAL implementation, which is at ~1.55 GFLOPS, already reaches 0.58% of the peak memory bandwidth.

4.5 Enabling parallel support using OpenMP

As discussed in Section 4.4, the redundant computation and verification instructions for the FT functionality lead to extra structural and data dependencies, which is the major reason for the fault-tolerant overhead for memory-bound routines. Therefore, we propose delicate assembly-level optimizations to alleviate the overhead. On a multi-core system, the massive parallelism introduced by enabling multithreading naturally reduces the cost of FT.

Most Level-1 and Level-2 BLAS routines require little communication or synchronization among threads, so one can promptly enable the parallel support for these routines by partitioning input vectors and/or input matrices when mapping workloads to physical cores. According to our evaluation, our DMR-based fault-tolerant BLAS implementations maintain negligible overhead after being threaded on Intel Cascade Lake processors.

4.6 Extending to AVX2-enabled CPUs

In an AVX2-enabled processor, there are 16 256-bit SIMD registers (`ymm0` - `ymm15`), which can store 4 packed DPs, namely 4 lanes. Compared with AVX-512-enabled Intel processors, an AVX2-enabled microarchitecture possesses a shorter SIMD width and fewer SIMD registers. Since the AVX2 instruction set does not support opmask registers, we substitute the `ymm` registers for the opmask registers in AVX512 in select cases — to store intermediate comparison results for reduction.

TABLE 3
Code snippet of the AVX2 fault tolerant code. FT-related instructions are marked in red.

Original Computation	AVX2 Protected Computation
	mov $15, r14d
vfmadd231pd ymm0, ymm1, ymm2	**vfmadd231pd ymm0, ymm1, ymm2**
	vfmadd231pd ymm0, ymm1, ymm3
	vpcmpeqd ymm2, ymm3, ymm4
	vmovskpd ymm4, r10d
	cmp r14d, r10d
	jne ERROR_HANDLER

Table 3 shows the code snippet of computation, duplication, and comparison using AVX2 instructions. We duplicate the original SIMD computation instruction and perform a lane-by-lane comparison between the duplicated and original computational results using `vpcmpeqd`. The comparison result is stored in the SIMD register `ymm4`, which is further extracted and stored in the 32-bit general-purpose register `r10d`. If all the four DPs (lanes) of the two 256-bit SIMD registers are confirmed equal, the `r10d` register will be 4 consecutive '1's. Otherwise, a mismatch at any specific lane

will set its corresponding bit in `r10d` to '0'. We test `r10d` by comparing against `r14d`, which is pre-initialized to '1111'. Any detected errors will redirect the control flow to the error handler by `jne`. When the loop body is unrolled, and there are multiple comparisons in an iteration, we store intermediate comparison results in different general-purpose registers such as `r11d`, `r12d`, and `r13d`. These intermediate results, similar to when adopting opmask registers in AVX-512, can be reduced by `and` to reduce the verification interval as well as the FT overhead.

5 Optimizing Fault Tolerant Level-3 BLAS

Since Level-3 BLAS routines are computing-bound routines, adopting the same DMR strategy as Level-1 and Level-2 BLAS, which doubles the computing instructions, will consequently double the performance overhead. Considering the limited registers in a single core, DMR will also increase the register pressure in the computing kernels, which will further hinder the performance. Therefore, we adopt the classic checksum-based ABFT scheme for our fault-tolerant functionality, introducing $O(n^2)$ computational overhead over the original $O(n^3)$ computation.

5.1 First trial: building online ABFT on a third-party library

Building ABFT on a third-party library is not a new topic [54]. As shown in the left side of Figure 4, we first encode checksums for matrices A, B, and C before starting matrix multiplication. The checksums C^c and C^r are updated asynchronously using a rank-k outer-product update of matrix C with a step size k=K_c. In every completed rank-k update, we verify the checksum relationship by first computing the reference row checksum C^r_{ref} according to the current matrix C and comparing it against C^f. If an error is detected, we continue to compute the reference column checksum C^c_{ref} and compare it against C^c to locate the erroneous row index i_{err} of C. If there is no error detected when comparing the row checksum vectors, we do not need to verify the column checksum vectors.

The total cost of the ABFT overhead consists of the initial checksum encoding, online checksum updating, and reference checksum computing–all of which are matrix-vector multiplications (DGEMV). T_{enc} includes the costs of encoding for four checksums (C^c,C^r,A^r, and B^c). T_{update} includes the cost of updating on two checksums (C^c and C^r). Denoting the time of an $n \times n$ DGEMV as t_{mv}, the total cost of ABFT T_{ovhd} is

$$T_{ovhd} = T_{enc}+T_{update}+\frac{K}{Kc}\cdot(T_{C^r_{ref}}+T_{C^c_{ref}}) = (6+\frac{2K}{Kc})t_{mv}.$$

We further denote the performance of DGEMV and DGEMM as P_{mv} and P_{mm}, both in units of GFLOPS. Consequently, the total execution times of $n \times n$ DGEMM and DGEMV are T_{GEMM}=$2e^{-9}n^3/P_m$ and t_{mv}=$2e^{-9}n^2/P_{mv}$. Therefore, we have

$$\frac{T_{ovhd}}{T_{GEMM}} = \frac{(6+\frac{2K}{Kc})t_{mv}}{2e^{-9}n^3/P_{mm}} = \frac{(6+\frac{2K}{Kc})P_{mm}}{n\cdot P_{mv}}.$$

As shown in the above derivation, the real influence of ABFT is not simply a $O(1/n)$ computationally negligible contribution to the baseline but is dependent on the relative performance between the memory-speed-determined P_{mv} and computing-capability-determined P_{mm} as well. On non-AVX-512-enabled CPUs, P_{mm}/P_{mv} ranges from 5 to 20, while on AVX-512-enabled CPUs, this ratio can be as large as 35, exaggerating the overhead up to 7-fold over old processors. The ABFT overhead reported for an older CPU [54] is around 2%, while the overhead on an AVX-512-enabled processor, measured by our benchmark in Section VI, is 15.27% — much larger than on old processors.

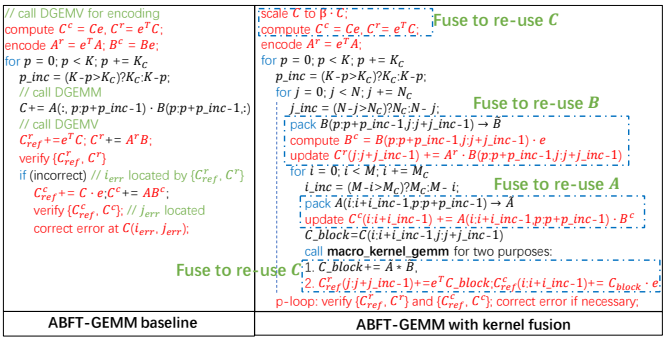


Fig. 4. Outer-product online ABFT DGEMM optimization layout. The ABFT-related operations are marked in red.

5.2 Reducing the memory footprint: fusing ABFT into DGEMM

As discussed in the previous section, the huge gap between memory transfer and floating-point computation is the reason the $O(n^2)$ checksum-related operations can no longer be amortized by the $O(n^3)$ GEMM. We, therefore, design a fused ABFT scheme to minimize the memory footprint of checksum operations. To be more specific, the encoding of C^c and C^r is fused with the matrix scaling routine C=βC. When we load B to pack it to the continuous memory buffer $\tilde{B}$, checksum B^c is computed and checksum C^r is computed simultaneously by reusing B. In this fused packing routine, each B element is reused three times for each load. Similarly, each element of A loaded for packing is reused to update the column checksum C^c. In the macro kernel, which computes C_{block}+=$\tilde{A} \cdot \tilde{B}$, we reuse the computed C elements at the register level to update the reference checksums C^r_{ref} and C^c_{ref} to verify the correctness of the computation. By fusing the ABFT memory footprint into DGEMM, the FT overhead becomes purely computational, decreasing from about 15% to 2.94%. It is worth noting that, in addition to square inputs, our proposed kernel fusion strategy for ABFT GEMM can be beneficial to irregularly shaped GEMMs. With the proper kernel parameters selected, the performance overhead of fault-tolerant kernels can be well maintained for even irregularly shaped inputs. We refer the interested reader to [64] for details.

5.3 Enabling parallel support for ABFT using OpenMP

In addition to providing highly efficient serial implementations, we further enable the multithreading support for DGEMM with and without fault tolerance. As discussed in Section 3.3.2, our DGEMM starts from three `for` loops

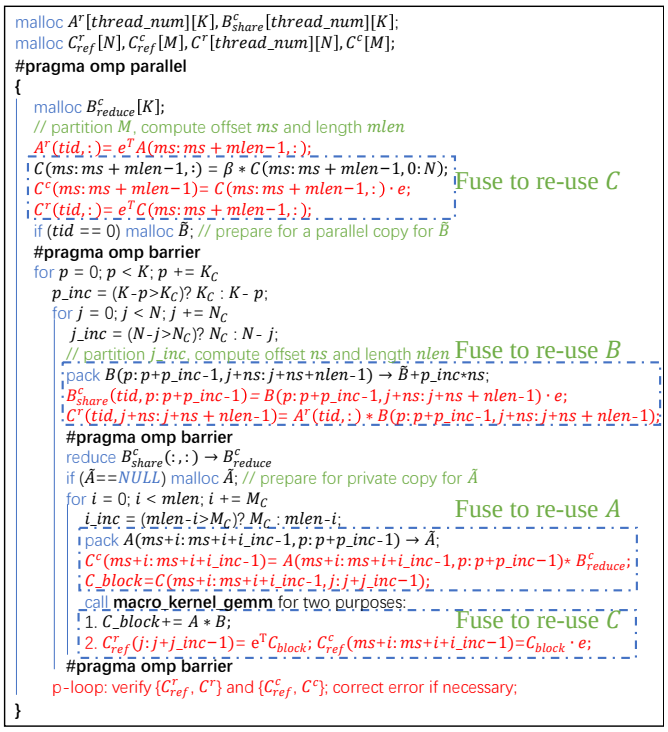


Fig. 5. Parallel ABFT-GEMM with kernel fusion. The ABFT-related operations are marked in red.

allowing submatrices of A ($M_C \times K_C$) and B ($K_C \times N_C$) to reside in L2 and L3 cache, respectively. The cache-blocking parameters M_C, K_C, and N_C are tuned to fit with the physical cache size. In practice, we set $M_C = 192$, $K_C = 384$, and $N_C = 9216$ for AVX-512-enabled DGEMM. Before starting the computation, both submatrices A and B are packed into continuous memory buffers, namely $\tilde{A}$ and $\tilde{B}$, to minimize TLB misses in performance-sensitive computing kernels.

On Intel Skylake and Cascade Lake server CPUs, physical cores share a large unified L3 cache while each physical core holds a smaller private L2 cache. To map this cache hierarchy in a threaded implementation, we allocate a memory buffer shared among all the threads for $\tilde{B}$, and each thread requests a private memory buffer for $\tilde{A}$. The computation workload on the C matrix is partitioned along the M-dimension. Since memory buffers $\tilde{A}$ are thread-private, each thread packs data from matrix A into their own $\tilde{A}$ buffers. When packing matrix C into the shared memory buffer $\tilde{B}$, the memory access workloads are partitioned along the N-dimension and each thread is responsible for packing a chunk of $\tilde{B}$. Just like the serial ABFT GEMM implementation, we conduct checksum encoding for the row checksum vector of A (A^r) and full checksum vectors of C (C^c, C^r). To compute the C checksums, we partition the C matrix along the M-dimension such that each thread computes a slice of the column checksum C^c while maintaining a local copy of its own row checksum vector C^r. Similarly, we partition the A matrix along the M-dimension to compute its row checksums A^r in parallel. The checksum encoding of B^c is fused with the parallel packing operation for B to $\tilde{B}$ and at the same time, we update the reference row checksum of C. Therefore, each B element loaded from the main memory is re-used three times. Since the parallel copy operation partitions B from the N-dimension, an extra stage of reduction operation among threads is required to compute the final column checksum B^c.

6 Experimental Evaluation

To validate the effectiveness of our optimizations, we compare the performance of FT-BLAS with three state-of-the-art BLAS libraries: Intel oneMKL (`2020.2`, abbreviated as MKL in this Section), OpenBLAS (`0.3.13`), and BLIS (`0.8.0`), on a machine with an Intel Gold 5122 Skylake processor at 3.60 GHz, equipped with 96 GB DDR4-2666 RAM. We also compare the performance of FT-BLAS under error injection with references on an Intel Xeon W-2255 Cascade processor. This Cascade Lake machine has a 3.70 GHz base frequency and 32 GB DDR4-2933 RAM. Hardware prefetchers on both machines are enabled according to the Intel BIOS default [75]. In addition to Intel processors, we validate our AVX2 implementations on an AMD Ryzen7 3700X desktop processor and an AMD EPIC 7713 server processor. The AMD desktop processor has a 3.60 GHz base frequency and 32 GB DDR4-2933 RAM, while the AMD EPIC 7713 server processor has a 2.0 GHz base frequency (3.68 GHz boost frequency) equipped with 512 GB main memory. We repeat each measurement twenty times and then report the average performance. For Level-1 BLAS routines, the performance is averaged from array lengths ranging from 5×10^6 to 7×10^6. For Level-2 and Level-3 BLAS routines, the performance is averaged for matrices ranging from 2048^2 to 10240^2. For the multi-threading parallel benchmark, we test the array lengths ranging from 2×10^8 to 3×10^8 and matrices ranging from 512^2 to 20480^2. We compile the code with `icc 19.0` and the `-O3` optimization flag.

6.1 Performance of FT-BLAS without FT Capability

We provide a new BLAS implementation, comparable to or faster than the modern state-of-the-art, before embedding FT capability. We abbreviate this BLAS implementation as *FT-BLAS: Ori* in the figures.

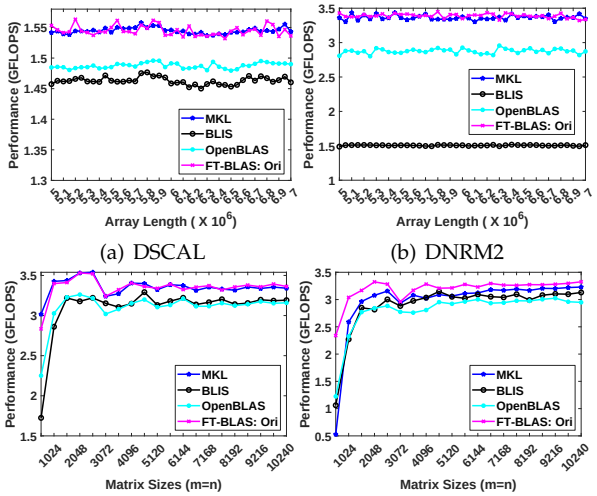
6.1.1 *Optimizing Level-1 BLAS*

For memory-bound Level-1 BLAS, the optimization strategies employed are: 1) exploiting data-level parallelism using the latest SIMD instructions, 2) assisting pipelining by unrolling the loop, and 3) prefetching. As seen in Table 1, OpenBLAS has under-optimized routines, such as DSCAL and DNRM2, with respect to prefetching and AVX-512 support. We add prefetching for DSCAL, obtaining 3.85% and 5.61% speed-up over OpenBLAS and BLIS. DNRM2 is only supported with SSE2 by OpenBLAS, so our AVX-512 implementation provides a 17.89% improvement over OpenBLAS, while reaching a 2.25-fold speedup on BLIS. Our implementations for both routines reach comparable performance to closed-source MKL, as seen in Figure 6.

6.1.2 *Optimizing Level-2 BLAS*

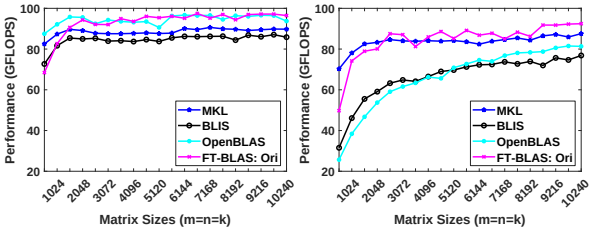
Register-level data re-use enters the picture in the Level-2 BLAS routine optimization. Following the optimization schemes described in Section II, we see in Figure 6 that our

DGEMV obtains a 7.13% speed-up over OpenBLAS. This is enabled by discarding cache blocking on matrix A over concerns about the potential harm of discontinuous memory accesses regarding TLB thrashing and the corresponding performance of hardware prefetchers. Because BLIS adopts the same strategy as OpenBLAS on DGEMV, our DGEMV is 6.16% faster than BLIS, while achieving nearly indistinguishable performance with MKL. For DTRSV, our strategy of minimizing the blocking parameter to cast the maximized computations to the more efficient Level-2 BLAS DGEMV grants us higher performance than all baselines, surpassing MKL, OpenBLAS, and BLIS by 3.76%, 11.17%, and 6.98%, respectively.



(a) DSCAL (b) DNRM2 (c) DGEMV (d) DTRSV

Fig. 6. Comparisons of selected Level-1/2 BLAS routines.



(a) DGEMM (b) DTRSM

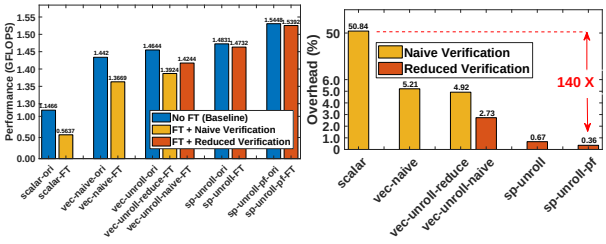
Fig. 7. Comparisons of selected Level-3 BLAS routines.

6.1.3 *Optimizing Level-3 BLAS*

Adopting the traditional cache blocking and packing scheme, our DGEMM performs similarly to OpenBLAS DGEMM. As seen in Figure 7, both of these DGEMM implementations outperform MKL and BLIS by 7.29-11.75%. For the Level-3 BLAS routine DTRSM, we provide a highly-optimized macro kernel to solve for the diagonal block and cast the majority of the computation to the near-optimal DGEMM. Because OpenBLAS and BLIS simply provide an unoptimized scalar implementation for the diagonal solver, our DTRSM outperforms OpenBLAS and BLIS by 22.19% and 24.77% and surpasses MKL by 3.33%.

6.2 Performance of FT-BLAS with Fault Tolerance Capability

Having achieved comparable or better performance than the current state-of-the-art BLAS libraries without fault tolerance, we now add on fault tolerance functionalities. For memory-bound Level-1 and Level-2 BLAS routines, we propose a novel DMR verification scheme based on the AVX-512 instruction set and then further reduce the overhead of fault tolerance to a negligible level via assembly optimization. For compute-bound Level-3 BLAS, we fuse the checksum calculations into the packing routines and assembly kernels to reduce data transfer between registers and memory. The results in this section were obtained with fault-tolerant DMR and ABFT operating, but not under active fault injection—see subsection C for injection experiments.

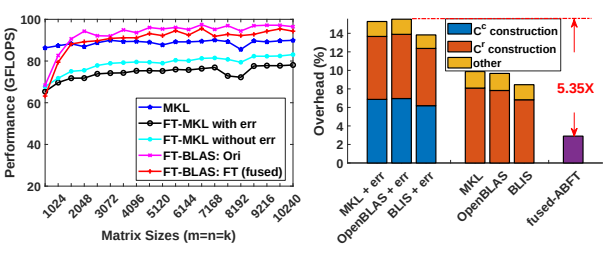


(a) Performance Optimization (b) Overhead Optimization

Fig. 8. Optimizing DSCAL with/without FT.

6.2.1 *Reducing DMR overhead for memory-bound routines*

Figure 8 presents the performance and overhead of DSCAL with step-wise assembly-level optimization. In each step, the assembly optimization described in Sections III and IV is applied to the FT version and its baseline, our non-FT version evaluated above. The performance of the most naive baseline, a scalar implementation, is 1.15 GFLOPs. Duplicating computing instructions and verifying correctness for this baseline halves the performance to 0.56 GFLOPS, bringing a 50.83% overhead. A vectorized implementation based on AVX-512 instructions decreases overhead by a factor of 9.8 compared to the scalar duplication/verification scheme. A vectorized implementation with fault-tolerance capability increases performance to 1.36 GFLOPs, a 2.42-fold increase of the scalar FT version. After this vectorization, simply unrolling the loop gains 1.55% and 1.87% improvement on the non-FT (vec-unroll-ori) and FT (vec-unroll-naive) versions respectively, while the overhead is now 4.9%. It is at this point that our non-FT version reaches OpenBLAS performance. Our novel verification scheme involving opmask registers improves the overhead to 2.7%. We then schedule instructions via heuristic software pipelining, improving the performance of the non-FT (sp-unroll-ori) and FT (sp-unroll-FT) implementations to 1.48 and 1.47 GFLOPs respectively. The overhead improves to 0.67% in this step. We add prefetch instructions as a final step, and the overhead settles at 0.36%.



(a) Performance of FT-DGEMM (b) Linking to different libraries

Fig. 9. Optimizing DGEMM with FT.

6.2.2 Reducing ABFT overhead for compute-bound routines

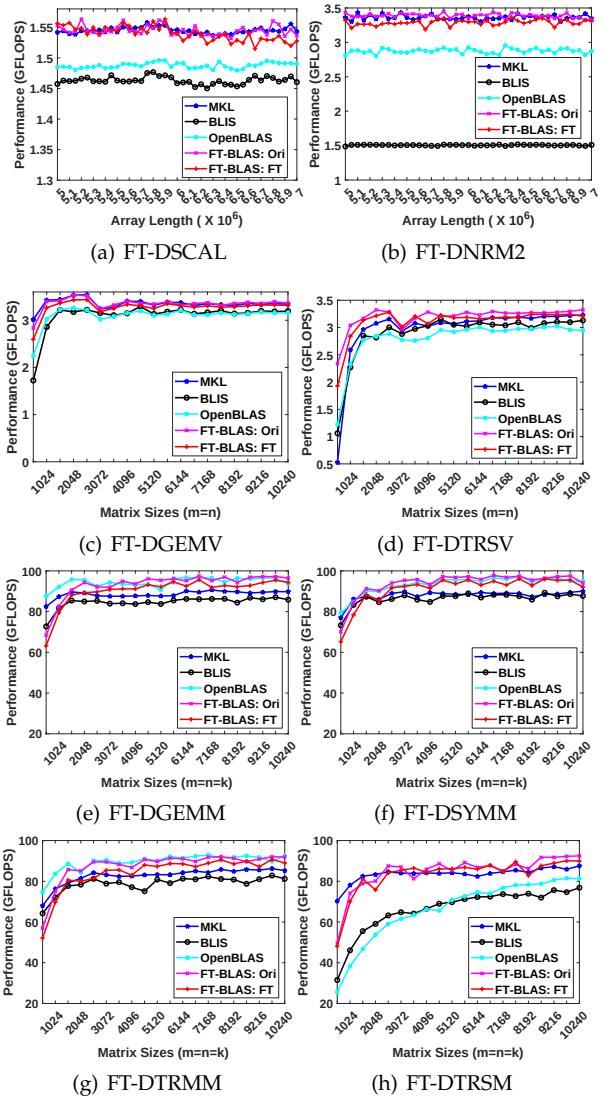
Figure 9(a) presents the performance of two methods of implementing ABFT for GEMM: building upon MKL (FT-MKL) and fusing into the GEMM routine (FT-BLAS: FT fused). FT-MKL under error injection leads to 15% overhead compared with the baseline MKL. When there is no error injected, we no longer compute and verify the checksum C^r so the overhead decreases to 9%. In contrast, the fused implementation (2.9% overhead) of ABFT does not generate an extra cost when encountering errors because its reference checksum computation is fused into the assembly computing kernel and is computed regardless of whether an error is detected. As shown in Figure 9(b), the overhead of building ABFT on a third-party library slightly varies when linking to different libraries, but the trend is clear: reference checksum construction generates the majority of the ABFT overhead, which is eliminated by the fusing strategy. The overhead can be up to a factor of 5.35 that of fusing ABFT into DGEMM. Our overhead is also lower than the 2015 work by Smith et al. [57], where checkpoint/rollback recovery is used to tolerate errors. Their checkpoint/rollback recovery has a wider error coverage, but the overhead is "in the range of 10%" [57].

6.2.3 Generalizing to other routines

Figure 10 compares the performance of FT-BLAS with FT capability (FT-BLAS: FT) against its baseline: our implementation without FT capability (FT-BLAS: Ori) and reference BLAS libraries on eight routines of all three levels of BLAS. The DMR-based FT implementations for the Level-1 and Level-2 BLAS routines (DSCAL, DNRM2, DGEMV, and DTRSV) generate 0.34%-3.10% overhead over the baseline. For the Level-3 BLAS routines, DGEMM, DSYMM, DTRMM, and DTRSM, our strategy to fuse memory-bound ABFT operations with matrix computation generates overhead ranging from 1.62% to 2.94% on average. Our implementation strategy for DSYMM in both FT-BLAS: Ori and FT-BLAS: FT is similar to the DGEMM scheme, with moderate modification to the packing routines. For DTRMM, we use the same strategy with some additional modifications to the computing kernel, similar to the methods in [76]. With these negligible overheads added to an already high-performance baseline, our FT-BLAS with FT capability remains comparable to or faster than the reference libraries.

6.2.4 Benchmarking on an Intel Cascade Lake processor

Figure 11 benchmarks the performance of FT-BLAS with and without FT capability by comparing against reference BLAS libraries on an Intel Cascade Lake Xeon W-2255 processor. Similar to the results on Skylake, our baseline BLAS implementations (FT-BLAS: Ori) present comparable or better performance compared with MKL, OpenBLAS, and BLIS. The DMR-based FT implementations for memory-bound Level-1 and Level-2 BLAS routines (DSCAL, DNRM2, DGEMV, and DTRSV) add 0.06%-2.79% overhead to the non-FT baselines. Our fused fault tolerant strategy for compute-bound Level-3 BLAS routines (DGEMM, DSYMM, DTRMM, and DTRSM) generates 1.17%-3.58% overhead on average over the baseline. With



(a) FT-DSCAL (b) FT-DNRM2

(c) FT-DGEMV (d) FT-DTRSV

(e) FT-DGEMM (f) FT-DSYMM

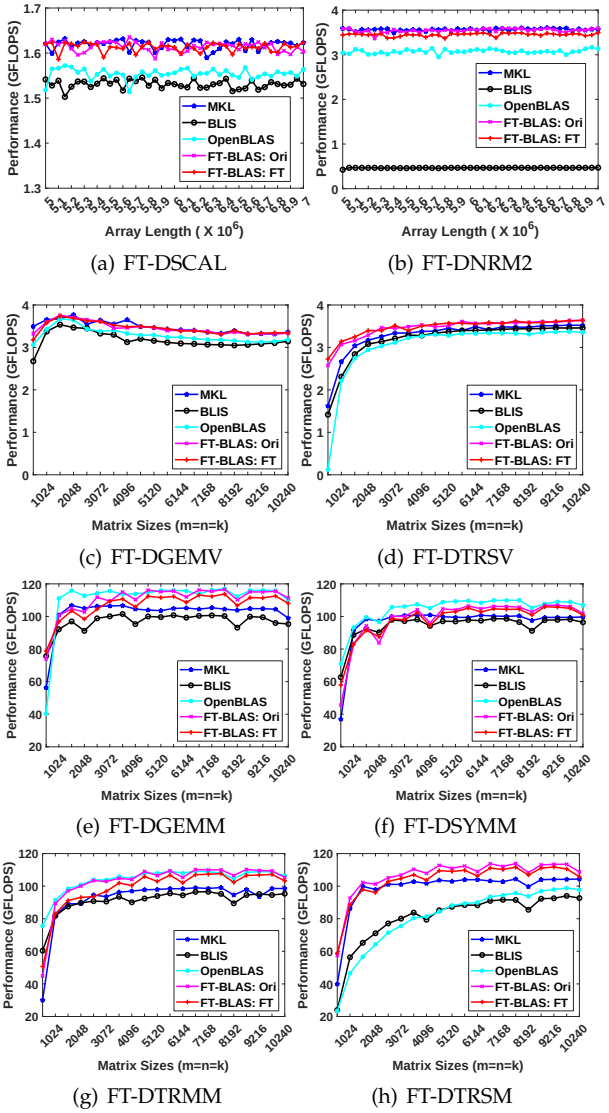
(g) FT-DTRMM (h) FT-DTRSM

Fig. 10. Comparisons of selected BLAS routines with FT on Skylake.

these negligible overheads added to our highly efficient non-FT baselines, our FT-BLAS maintains its performance as comparable to or faster than all of the state-of-the-art BLAS libraries.

6.2.5 Enabling the parallel support

Figure 12 compares the parallel performance of FT-BLAS with FT capability (FT-BLAS: FT) against its baseline: our implementation without FT capability (FT-BLAS: Ori) and reference BLAS libraries on four routines of all three levels of BLAS on an Intel Cascade Lake Xeon W-2255 processor. After enabling parallel support, the memory-bound Level-1 and Level-2 BLAS routines DDOT, DNRM2, and DGEMV, which require mostly embarrassing parallelisms, maintain a negligible overhead (0.15% - 3.53%) similar to that of serial implementations. It is worth mentioning that BLIS supports parallel implementations only for Level-3 BLAS routines, and OpenBLAS also does not provide parallel support for DNRM2. Therefore, enabling multi-threading does not increase their performance. Regarding the compute-bound Level-3 BLAS DGEMM, with our parallel design introduced in Figure 5, we manage to scale the performance



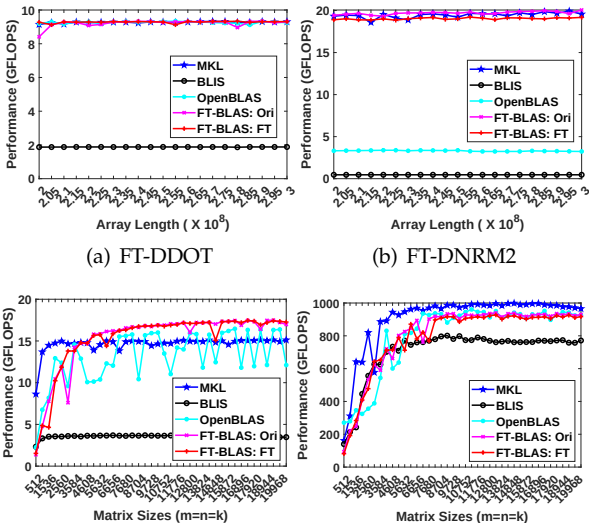
(a) FT-DSCAL (b) FT-DNRM2

(c) FT-DGEMV (d) FT-DTRSV

(e) FT-DGEMM (f) FT-DSYMM

(g) FT-DTRMM (h) FT-DTRSM

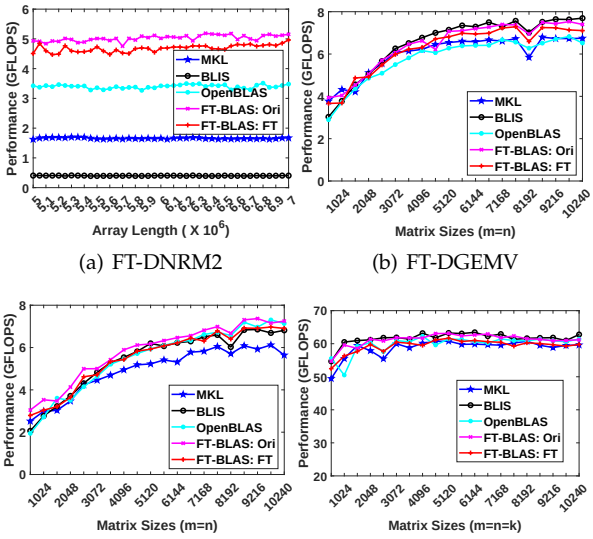
Fig. 11. Comparisons of BLAS routines with FT on Cascade Lake.



(a) FT-DDOT (b) FT-DNRM2

(c) FT-DGEMV (d) FT-DGEMM

Fig. 12. Comparisons of parallel BLAS routines with FT on Cascade Lake.



(a) FT-DNRM2 (b) FT-DGEMV

(c) FT-DTRSV (d) FT-DGEMM

Fig. 13. Comparisons of BLAS routines with FT on AMD Zen2.

of ABFT-DGEMM on the shared-memory multi-core platform, obtaining scalability similar to that of OpenBLAS. With the scalable parallel design and ABFT operations fused into packing routines and assembly kernels, FT-DGEMM presents a negligible overhead (1.79%). The performance of our DGEMM implementation with FT is 16.97% faster than BLIS, comparable to OpenBLAS while slightly underperforming the closed-source Intel MKL.

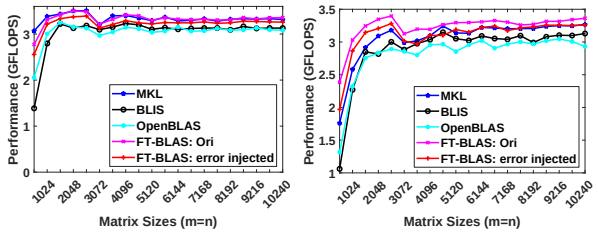
6.2.6 *Extending to AVX2-enabled AMD processors*

Figure 13 compares the performance of four routines spanning all three levels of BLAS on an AVX2-enabled AMD R7 3700X processor. Since the number of SIMD registers on AVX2 ISA is halved compared with AVX-512, we suffer from register pressure for the in-register checkpointing strategy that we have presented. Therefore, we choose to only detect errors for memory-bound routines rather than correcting them online. Experimental results show that both of our DMR- and ABFT-based fault-tolerant optimization strategies remain valid for AVX2 routines. With negligible overhead added to our already highly efficient non-FT baselines, our BLAS implementation with FT capability maintains its performance comparable to or faster than the reference libraries. It is worth mentioning that our AMD Zen2 CPU adopts the Uniform Memory Access (UMA) mode, or namely distributed mode, by default, which enables a single-thread application to take advantage of the entire memory bandwidth delivered by all of the memory channels. Therefore, we observe the performance of single-thread memory-bound Level-1 and Level-2 BLAS routines to be significantly faster than that on Intel processors under the NUMA mode (or local mode) [77].

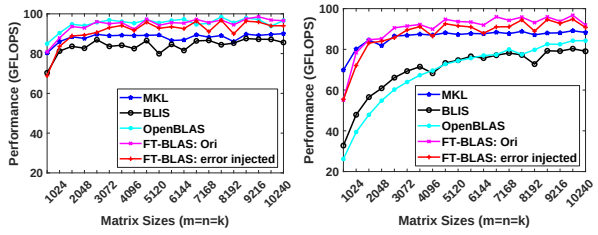
6.3 Error Injection Experiments

We validate the effectiveness of our fault-tolerance scheme by injecting multiple computing errors into each of our computing kernels and verifying our final computation results against MKL. External error injection tools often significantly slow down the native program [78], [79], [80]. Therefore, we inject errors at the source code level to minimize the performance impact on native programs.

We inject 20 errors into each routine. The length of the injection interval k is determined based on the number of errors to inject, that is, we inject one error every k iterations. For ABFT-protected Level-3 BLAS routines, the error injection is straightforward because we can directly operate in C code. An element of matrix C is randomly selected for modification when an injection point is reached. This injected error will lead to a difference in the checksum relationship, and the erroneous element and error magnitude will be computed accordingly. This detected error is then corrected by subtracting the error magnitude from the erroneous position. For DMR-protected Level-1 and Level-2 BLAS routines, the injection is more complicated since the loop body is implemented purely using assembly codes. Therefore, providing an assembly-level error injection mechanism becomes necessary. Once the program reaches an injection point, we redirect the control flow to a faulty loop body to generate an error. This generated error is then detected via comparison with the computed results of the duplicated instruction. After the error is detected, a recovery procedure is activated to recompute the corrupted iteration immediately. In all cases, we validate the correctness of our final computations by comparing them with MKL to ensure all injected errors were truly corrected.



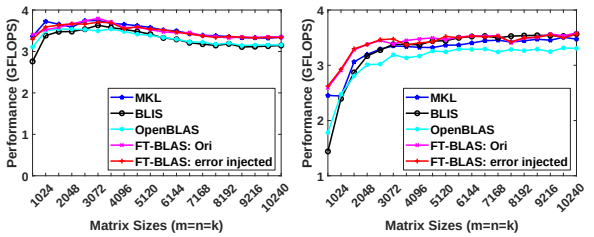
(a) DGEMV with error injection (b) DTRSV with error injection



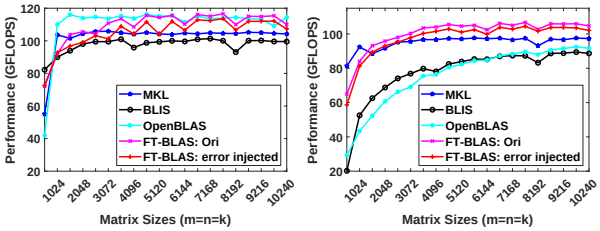
(c) DGEMM with error injection (d) DTRSM with error injection

Fig. 14. Performance under error injection on Skylake.

Figure 14 compares the performance of four routines under error injection. For both DMR-protected (DGEMV, DTRSV) and ABFT-protected (DGEMM, DTRSM) routines, we maintain negligible (2.47%-3.22%) overhead, and the overall performance under error injection remains comparable or faster than reference libraries. In particular, our DTRSM outperforms OpenBLAS, BLIS, and MKL by 21.70%, 22.14%, and 3.50% even under error injection. Our experimental results confirm that our protection schemes do not require significant extra overhead to correct errors. This is because our correction methods—either to recompute the corrupted iteration or to subtract an error magnitude from the incorrect position—generate only a few ALU computations instead of expensive memory accesses.



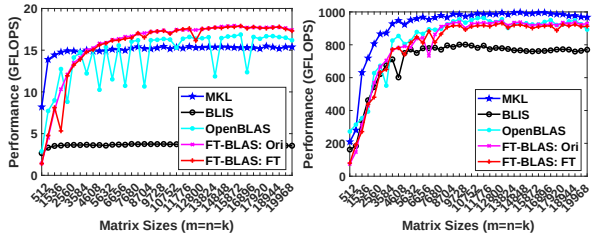
(a) DGEMV with error injection (b) DTRSV with error injection



(c) DGEMM with error injection (d) DTRSM with error injection

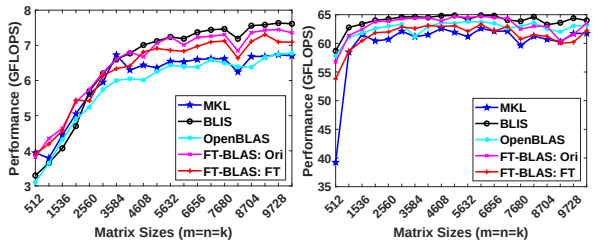
Fig. 15. Performance under error injection on Cascade Lake.

Figure 15 benchmarks FT-BLAS under error injection using another processor, the Intel Cascade Lake W-2255. According to our experimental results, our protection scheme is as lightweight as it was on the Skylake processor and is still able to surpass open-source OpenBLAS and BLIS by 22.89% and 21.56% and the closed-source MKL by 4.98% even while tolerating 20 injected errors. The execution time of DTRSM and DTRSV for 512^2 to 10240^2 matrices ranges from 2 ms to 20 seconds. Therefore, injecting 20 errors into these two routines is equivalent to injecting 1 to 10,000 errors per second. Hence, FT-BLAS is able to tolerate up to thousands of errors per second with comparable and sometimes faster performance than state-of-the-art BLAS libraries—and none of them can tolerate soft errors.



(a) FT-DGEMV (b) FT-DGEMM

Fig. 16. Parallel performance under error injection on Cascade Lake.

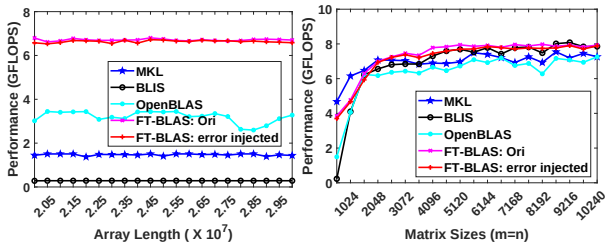


(a) FT-DGEMV (b) FT-DGEMM

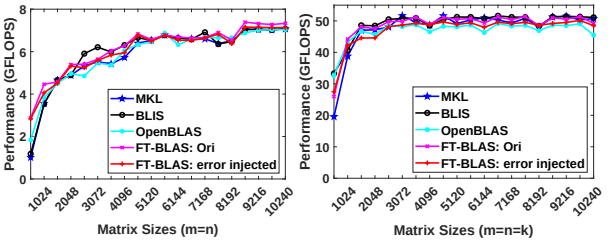
Fig. 17. Performance under error injection on AMD Zen2.

Figure 16 compares the parallel performance of DGEMV and DGEMM under error injection on an Intel Cascade Lake

W-2255 processor. When being threaded, our FT-DGEMV under error injection remains 10.91% and 13.49% faster than OpenBLAS and MKL. Compared with the non-threaded BLIS, our FT-DGEMV is 3.72× faster even under error injection. Regarding the compute-bound DGEMM, our FT-BLAS presents a performance comparable to OpenBLAS and is 16.83% faster than BLIS. Figure 17 further benchmarks the serial performance of these two BLAS routines under error injection on an AMD Zen2 Ryzen 3700X processor, validating that the overall performance of FT-BLAS remains comparable to the best of the-state-of-the-art reference libraries on AVX2-enabled AMD processors.



(a) FT-DNRM2 (b) FT-DGEMV



(c) FT-DTRSV (d) FT-DGEMM

Fig. 18. Comparisons of BLAS routines with FT on AMD EPYC 7713.

Figure 18 further compares the performance of FT-BLAS under error injection on an AMD EPIC 7713 CPU. Similar to other platforms, on this AMD server CPU, our FT-BLAS continues to present a performance comparable to or faster than the state-of-the-art BLAS libraries, MKL, OpenBLAS and BLIS with a negligible overhead added when encountering injected errors. Regarding the four representative BLAS routines spanning all three levels of BLAS we benchmarked, FT-BLAS shows a marginal fault-tolerant overhead of up to 2.70% compared with our non-fault-tolerant baseline.

7 Conclusions

We present a fault-tolerant BLAS implementation that is not only capable of tolerating soft errors, but also achieves comparable or superior performance over the current state-of-the-art libraries, OpenBLAS, BLIS, and Intel MKL. Future work will focus on extending FT-BLAS to more architectures and eventually open-sourcing the code.

8 Acknowledgements

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through the Advanced Computing (SciDAC) program under Award Number DE-SC0022209.

References

[1] J.-C. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15*, pp. 2–11, 1985.

[2] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*. IEEE, 1993, pp. 126–133.

[3] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No. PR00146)*. IEEE, 1999, pp. 86–94.

[4] L. A. B. Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 595–602.

[5] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 57.

[6] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra *et al.*, "DOE advanced scientific computing advisory subcommittee (ASCAC) report: top ten exascale research challenges," USDOE Office of Science (SC)(United States), Tech. Rep., 2014.

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

[8] R. Baumann, "Soft errors in commercial semiconductor technology: Overview and scaling trends," *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, vol. 7, 2002.

[9] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, 2016.

[10] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, "Experimental and analytical study of Xeon Phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 28.

[11] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16.

[12] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[13] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[15] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Improving performance of iterative methods by lossy checkponting," in *Proceedings of the 27th international symposium on high-performance parallel and distributed computing*, 2018, pp. 52–65.

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[17] D. Hakkarinen, P. Wu, and Z. Chen, "Fail-stop failure algorithm-based fault tolerance for cholesky decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1323–1335, 2014.

[18] Z. Chen and J. Dongarra, "A scalable checkpoint encoding algorithm for diskless checkpointing," in *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, 2008, pp. 71–79.

[19] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[20] S. Mitra, P. Bose, E. Cheng, C.-Y. Cher, H. Cho, R. Joshi, Y. M. Kim, C. R. Lefurgy, Y. Li, K. P. Rodbell *et al.*, "The resilience wall: Cross-layer solution strategies," in *Proceedings of Technical Program-2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*. IEEE, 2014, pp. 1–11.

[21] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 587–596.

[22] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[23] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp, "Towards a more complete understanding of SDC propagation," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 131–142.

[24] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[25] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 8.

[26] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[27] N. Oh, P. P. Shirvani, and McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[28] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[29] J. Yu, M. J. Garzaran, and M. Snir, "Esoftcheck: Removal of non-vital checks for fault tolerance," in *2009 International Symposium on Code Generation and Optimization*. IEEE, 2009, pp. 35–46.

[30] Z. Chen, A. Nicolau, and A. V. Veidenbaum, "SIMD-based soft error detection," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016, pp. 45–54.

[31] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

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

[33] P. Sao and R. Vuduc, "Self-stabilizing iterative solvers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2013, pp. 1–8.

[34] S. Di and F. Cappello, "Adaptive impact-driven detection of silent data corruption for HPC applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2809–2823, 2016.

[35] A. Chien, P. Balaji, P. Beckman, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, R. Schreiber *et al.*, "Versioned distributed arrays for resilience in scientific applications: Global view resilience," *Procedia Computer Science*, vol. 51, pp. 29–38, 2015.

[36] S. Li, H. Li, X. Liang, J. Chen, E. Giem, K. Ouyang, K. Zhao, S. Di, F. Cappello, and Z. Chen, "FT-iSort: efficient fault tolerance for introsort," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019, p. 71.

[37] X. Liang, J. Chen, D. Tao, S. Li, P. Wu, H. Li, K. Ouyang, Y. Liu, F. Song, and Z. Chen, "Correcting soft errors online in fast Fourier transform," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 30.

[38] A. Antola, R. Negrini, M. Sami, and N. Scarabottolo, "Fault tolerance in FFT arrays: time redundancy approaches," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 4, no. 4, pp. 295–316, 1992.

[39] D. Tao and C. R. Hartmann, "A novel concurrent error detection scheme for FFT networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 198–221, 1993.

[40] Z. Chen, "Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 167–176.

[41] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen, "New-sum: A novel online ABFT scheme for general iterative methods," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 43–55.

[42] L. Chen, D. Tao, P. Wu, and Z. Chen, "Extending checksum-based abft to tolerate soft errors online in iterative methods," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2014, pp. 344–351.

[43] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "Algorithm-based fault tolerance for convolutional neural networks," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[44] H. Zamani, Y. Liu, D. Tripathy, L. Bhuyan, and Z. Chen, "Greenmm: energy efficient GPU matrix multiplication through undervolting," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 308–318.

[45] L. Tan, S. Kothapalli, L. Chen, O. Hussaini, R. Bissiri, and Z. Chen, "A survey of power and energy efficient techniques for high performance numerical linear algebra operations," *Parallel Computing*, vol. 40, no. 10, pp. 559–573, 2014.

[46] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson, "Investigating the interplay between energy efficiency and resilience in high performance computing," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 786–796.

[47] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers, "A description of the advanced research wrf version 3. ncar technical note-475+ str," 2008.

[48] K. Hanasaki, Z. A. Ali, M. Choi, M. Del Ben, and B. M. Wong, "Implementation of real-time tddft for periodic systems in the open-source pyscf software package," *Journal of Computational Chemistry*, vol. 44, no. 9, pp. 980–987, 2023.

[49] *Intel Math Kernel Library. Reference Manual*. Santa Clara, USA: Intel Corporation, 2009, iSBN 630813-054US.

[50] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[51] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[52] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: automatically generate high performance dense linear algebra kernels on x86 CPUs," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[53] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998, pp. 38–38.

[54] P. Wu, C. Ding, L. Chen, T. Davies, C. Karlsson, and Z. Chen, "On-line soft error correction in matrix–matrix multiplication," *Journal of Computational Science*, vol. 4, no. 6, pp. 465–472, 2013.

[55] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, and R. Van de Gejin, "Fault-tolerant high-performance matrix multiplication: Theory and practice," in *2001 International Conference on Dependable Systems and Networks*. IEEE, 2001, pp. 47–56.

[56] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[57] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, and E. S. Quintana-Ortı, "Toward ABFT for BLIS GEMM."

[58] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1049–1059.

[59] 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. ACM, 2014, pp. 49–60.

[60] 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 ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 31–42.

[61] J. Chen, X. Liang, and Z. Chen, "Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with GPUs," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 993–1002.

[62] P. Wu, D. Li, Z. Chen, J. S. Vetter, and S. Mittal, "Algorithm-directed data placement in explicitly managed non-volatile memory," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 141–152.

[63] Y. Zhai, E. Giem, Q. Fan, K. Zhao, J. Liu, and Z. Chen, "Ft-blas: a high performance blas implementation with online fault tolerance," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 127–138.

[64] S. Wu, Y. Zhai, J. Liu, J. Huang, Z. Jian, B. Wong, and Z. Chen, "Anatomy of high-performance gemm with online fault tolerance on gpus," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 360–372.

[65] S. Wu, Y. Zhai, J. Huang, Z. Jian, and Z. Chen, "Ft-gemm: A fault tolerant high performance gemm implementation on x86 cpus," *arXiv preprint arXiv:2305.02444*, 2023.

[66] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 705–716.

[67] Y. Zhai, C. Jiang, L. Wang, X. Jia, S. Zhang, Z. Chen, X. Liu, and Y. Zhu, "Bytetransformer: A high-performance transformer boosted for variable-length inputs," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 344–355.

[68] H. Chen, Y. Zhai, J. J. Turner, and A. Feiguin, "A high-performance implementation of atomistic spin dynamics simulations on x86 cpus," *Computer Physics Communications*, vol. 291, p. 108851, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465523001960

[69] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*. IEEE, 2000, pp. 25–36.

[70] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[71] AGNER, https://www.agner.org/optimize/instruction_tables.pdf, 2019, online.

[72] OpenBLAS, https://github.com/xianyi/OpenBLAS/blob/develop/common.h#L530, Retrieved in 2021, online.

[73] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," *ACM SIGARCH Computer Architecture News*, vol. 14, no. 2, pp. 386–395, 1986.

[74] I. Corporation, "Intel 64 and ia-32 architectures optimization reference manual," *Intel Corporation, Sept*, 2019.

[75] Intel, https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html, 2014, online.

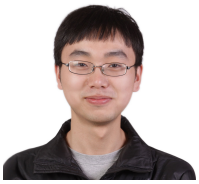
[76] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 BLAS," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 1, pp. 1–14, 2008.

[77] cpuwiki, https://en.wikichip.org/wiki/amd/microarchitectures/zen, 2021, online.

[78] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[79] D. Oliveira, V. Frattin, P. Navaux, I. Koren, and P. Rech, "Carol-fi: an efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators," in *Proceedings of the Computing Frontiers Conference*, 2017, pp. 295–298.

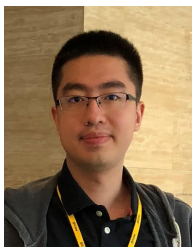
[80] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1245–1254.



Yujia Zhai received his bachelor's degree from the University of Science and Technology of China in 2016, a master's degree from Duke University in 2018, and a Ph.D. degree from the University of California, Riverside in 2023. He is interested in performance optimization for math libraries on GPUs. Email: yzhai015@ucr.edu.

Elisabeth Giem received two bachelor's degrees, one in pure mathematics and one in music (concentration in performance) from the University of California, Riverside (UCR). She received a master's degree in computational and applied mathematics from Rice University, and a master's degree in pure mathematics from UCR. She joined Zizhong Chen's SuperLab in 2018 as a computer science Ph.D. student, and has been awarded the NSF NRT In Computational Entomology Fellowship. Her research interests include but are not limited to high-performance computing, parallel and distributed systems, big data analytics, computational entomology, and numerical linear algebra algorithms and software. She has published many papers in prestigious journals and conferences, such as IEEE-TPAMI, and IEEE-TKD. Email: gieme01@ucr.edu.

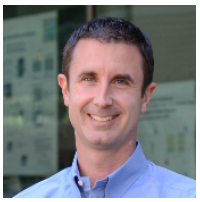
Kai Zhao is an Assistant Professor at the University of Alabama at Birmingham. He received his bachelor's degree from Peking University in 2014 and his Ph.D. degree from the University of California, Riverside in 2022. His research interests include high-performance computing, scientific data management and reduction, and resilient machine learning. Email: kzhao@uab.edu.



Jinyang Liu is a Ph.D. student at the University of California, Riverside. He received his bachelor's degree and master's degree from Peking University. He is also a long-term research intern at Argonne National Laboratory. His research interests include large-scale data management and reduction, deep learning applications on high-performance computing, and AI for science.. Email: jliu447@ucr.edu.

Jiajun Huang received his bachelor's degree in Electronic Information Engineering from the University of Electronic Science and Technology of China (UESTC) and the University of Glasgow (Honors of the First Class) in 2021. He is a long-term intern at Argonne National Laboratory. His research interests include but are not limited to Distributed and Parallel Computing/Systems, High-Performance Computing, and General Artificial Intelligence. Email: jhuan380@ucr.edu.

Bryan M. Wong is a professor in the Materials Science & Engineering Program and a Cooperating Faculty Member in the Department of Chemistry, Department of Physics & Astronomy, and Department of Electrical & Computer Engineering at UC Riverside. Prof. Wong received his Ph.D. in physical chemistry from the Massachusetts Institute of Technology (MIT) and is the recipient of an R&D 100 Award, a Department of Energy (DOE) Early Career Award, and an ACS COMP OpenEye Outstanding Junior Faculty Award in Computational Chemistry. His research interests include electron dynamics, time-dependent density functional theory, computational chemistry, and computational materials science. Email: bryan.wong@ucr.edu.



Christian R. Shelton is a professor of Computer Science & Engineering and a member of the Data Science Center at UC Riverside. He received his Ph.D. in Computer Science from the Massachusetts Institute of Technology (MIT) and the AFOSR Young Investigator Award. His research develops new machine learning methods and applies them to wide-ranging scientific domains. Email: cshelton@cs.ucr.edu.

Zizhong Chen (Senior Member, IEEE) received a bachelor's degree in mathematics from Beijing Normal University, a master's degree in economics from the Renmin University of China, and a Ph.D. degree in computer science from the University of Tennessee, Knoxville. He is a professor of computer science at the University of California, Riverside. His research interests include high-performance computing, parallel and distributed systems, big data analytics, cluster and cloud computing, algorithm-based fault tolerance, power and energy efficient computing, numerical algorithms and software, and large-scale computer simulations. His research has been supported by the National Science Foundation, Department of Energy, CMG Reservoir Simulation Foundation, Abu Dhabi National Oil Company, Nvidia, and Microsoft Corporation. He received a CAREER Award from the US National Science Foundation and a Best Paper Award from the International Supercomputing Conference. He is a Senior Member of the IEEE and a Life Member of the ACM. Email: chen@cs.ucr.edu.