Fault–Tolerant High–Performance Matrix Multiplication*

John A. Gunnels
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
gunnels@cs.utexas.edu

Daniel S. Katz
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Daniel.S.Katz@jpl.nasa.gov

Enrique S. Quintana–Ortí
Dept. de Informática
Universidad Jaume I
12080 Castellón
Spain
quintana@inf.uji.es

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
rvdg@cs.utexas.edu

FLAME Working Note #2

December 1, 2000

Abstract

In this paper, we extend the theory of algorithmic fault-tolerant matrix-matrix multiplication, $C = AB$, in a number of ways. First, we propose low-overhead methods for detecting errors introduced not only in C but also in A and/or B. Second, we theoretically show that the methods will detect all errors as long as only one entry is corrupted. Third, we propose a low-overhead rollback approach to correct errors once detected. Finally, we give a high-performance implementation of matrix-matrix multiplication that incorporates these error detection and correction methods. Empirical results demonstrate that the methods work well in practice with an acceptable level of overhead relative to high-performance implementations without fault-tolerance.

*This work was partially performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The work was funded by the Remote Exploration and Experimentation Project (a part of the NASA High Performance Computing and Communications Program funded by the NASA Office of Space Science.)

1 Introduction

The high-performance implementation of many linear algebra operations depends on the ability to cast most of the computation in terms of matrix-matrix multiplication [2, 3, 6, 12]. High-performance for matrix-matrix multiplication itself results from the fact that, for this operation, the cost of moving $b \times b$ blocks of the operands between the layers of the memory hierarchy is proportional to b^2 which can be amortized over $O(b^3)$ computations. These observations impact algorithmic fault-tolerance for linear algebra routines that spend most of their time in matrix-matrix multiplication in the following sense:

- If the matrix-matrix multiplication kernel used is fault-tolerant, the operation is largely fault-tolerant.

- Ensuring the integrity of a $b \times b$ block of a matrix can be expected to cost $O(b^2)$ time. This time can be amortized over the $O(b^3)$ operations performed with that data.

Thus, not only is the availabity of a fault-tolerant matrix-matrix multiplication an important first step towards creating fault-tolerance linear algebra libraries, but there is inherently an opportunity for adding fault-tolerance to matrix-matrix multiplication while retaining high-performance.

The primary goal for our mechanism is to detect a maximal fraction of errors while introducing minimal overhead. As argued in the previous paragraph, for the matrix product, with a cubic cost in floating–point arithmetic operations, we can expect to pay at least a quadratic cost. Thus, the goal is to find a mechanism with a quadratic cost. We follow, in that sense, the technique described in [13]. There, in essense, the correctness of $C = AB$ is established by looking at $d = Cw - ABw$ for a checksum vector w. The matrix-matrix multiplication is assumed to have been successful if d is of the order of the errors that could be introduced due to the use of finite precision arithmetic (round-off errors). In this paper, we generalize this method to $C \leftarrow \alpha AB + \beta C$, the form of matrix-matrix multiplication that is part of the level 3 Basic Linear Algebra Subprograms (BLAS) [5], and sharpen the theory behind the method.

The methods we present are closely related to those described in [11]. That paper proposes to augment matrices A, B, and C as

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

(Here, both v^T and w are checksum vectors.) By noting that in the absence of errors

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

they show how a comparison of $v^T C$ with $v^T AB$ and Cw with ABw can detect and correct errors introduced in matrix C.

On the surface, when comparing our methods to [11], it may appear that from an implementation point of view we simply perform the matrix-vector multiplies separately rather than as part of augmented matrices. However, our approach differs in a number of ways. First, we go well beyond the approach in [11] by also developing a sound theory behind the detection of errors introduced in A and B. Second, by adopting the techniques developed in [13] we explicitly deal with the question of how to differentiate errors due to corruption from errors due to round-off. Third, we take a very different approach to the correction of detected errors by using a rollback method. Finally, by adding fault-tolerance to a *high-performance* implementation of matrix-matrix multiplication we verify that the theoretical results can be implemented without sacrificing high performance.

The rest of the paper is structured as follows. In Section 2 we briefly describe the intended domain of application for our methods. In Section 3 we expound upon our theory concerning the effects of the introduction of one error in one of the matrices during a matrix-matrix multiplication. In Section 4 we describe how to take the results from Section 3 from theory to practice (although still at a high level of abstraction). A working fault-tolerant implementation of the matrix product based on a high-performance matrix-matrix multiplication implementation (ITXGEMM [8, 9]) is subsequently given in Section 5. The experimental results in Section 6 reveal the low overhead introduced in the matrix product by our fault–detection mechanism. We briefly discuss the current status of the project in Section 7 and concluding remarks are given in Section 8.

2 Target Application

Within NASA's High Performance Computing and Communications Program, the Remote Exploration and Experimentation (REE) project [1] at the Jet Propulsion Laboratory aims to enable a new type of scientific investigation by taking commercial supercomputing technology into space. Transferring such computational power to space will enable highly-autonomous, flexible missions with substantial on-board analysis capability, mitigating control latency issues due to fundamental light-time delays, as well as inevitable bandwidth limitations in the link between spacecraft and ground stations. To do this, REE does not intend to develop a new computational platform, but rather to define and demonstrate a process for rapidly transferring commercial high-performance computing technology into ultra-low power, fault-tolerant architectures for space.

The traditional method for protecting spacecraft components against faults caused by natural galactic cosmic rays and energetic protons has been radiation-hardening. However, radiation-hardening lowers the clock speed and may increase the required power of a component. Even worse, the time needed to design and bring a radiation-hardened component into production guarantees that it will be outdated when it is ready for use in space. Furthermore, it has a high cost which must be spread over a small number of customers. Typically, at any given time, radiation-hardened components have a power:performance ratio that is an order of magnitude lower, and a cost that is several orders of magnitude higher than contemporary commodity off-the-shelf (COTS) components. The REE project is therefore attempting to use COTS components in space and handling, via software, the faults that will occur.

Most of the transient faults encountered due to radiation in space will be single event effects (SEEs); their presence requires that the applications be self-checking, or tolerant of errors, as the first layer of fault-tolerance. Additional software layers will protect against errors that are not caught by the application [4]. For example, one such layer would automatically restart programs which have crashed or hung. This works in conjunction with self-checking routines: if an error is detected, and the computation does not yield correct results after a set number of retries, the error handling scheme aborts the program so that it can be automatically restarted.

SEEs affecting data are particularly troublesome because they typically have fewer obvious consequences than an SEE that impacts code — the latter would be expected to cause an exception. Note that since memory will be error-detecting and correcting, faults to memory will largely be screened; most faults will therefore impact the microprocessor or its L1 cache.

Due to the nature of most scientific codes, including the data processing applications currently being studied by REE, much of their time is spent in certain common numerical subroutines — as much as 70% in one NGST (Next Generation Space Telescope, the planned successor to the Hubble Space Telescope) application, for example. Protecting these subroutines from faults provides one level of protection in an overall software-implemented fault-tolerance scheme.

3 Detecting Errors

In this section we develop a theoretical foundation for error detection in the operation $C = AB$ where C, A, and B are $m \times n$, $m \times k$, and $k \times n$, respectively. Here, we use partitionings of A and B by columns and rows, respectively:

$$A = \left(\begin{array}{c|c|c} a_1 & \cdots & a_k \end{array} \right) \quad \text{and} \quad B = \left(\begin{array}{c} \hat{b}_1^T \\ \hline \vdots \\ \hline \hat{b}_k^T \end{array} \right).$$

We also use two (possibly different) checksum vectors:

$$w = \begin{pmatrix} \omega_1 \\ \hline \vdots \\ \hline \omega_n \end{pmatrix} \qquad \text{and} \qquad v^T = \left(\begin{array}{c|c|c} \nu_1 & \cdots & \nu_m \end{array} \right).$$

For simplicity, we first assume that exact arithmetic is employed and then we discuss the tolerance threshold for the case where round-off errors are present.

3.1 Exact arithmetic

Consider the operation $C = AB$ and let $\tilde{C}$ be the matrix computed when at most one element of one of the three matrices is corrupted during the computation. (We primarily consider a single corruption since most errors will be SEEs.) In other words, view the operation as atomic and assume that before the computation one element of A or B is corrupted or after $C = AB$ has been formed one element of C is corrupted. We can think of the error as a matrix of the form $\eta e_i e_j^T$ added to one of the three matrices; here η is the magnitude of the error and e_k denotes the k–th column of the identity matrix. The possible computed results are then given in Table 1 in the row labeled "$\tilde{C}$". Naturally, we wish to detect when $F = \tilde{C} - C$ is nonzero (or, in the presence of round-off error, "significant"). Thus, we must compute or approximate the magnitude of F, e.g., as $\|F\|_\infty$, but we must do so without being able to form F. Moreover, relative to the cost of computing C, the computation of the estimation of $\|F\|_\infty$ must be cheap.

	Matrix Corrupted		
	$\tilde{A} = A + \eta e_i e_j^T$	$\tilde{B} = B + \eta e_i e_j^T$	$\tilde{C} = C + \eta e_i e_j^T$
$\tilde{C}$	$\tilde{A}B$	$A\tilde{B}$	$AB + \eta e_i e_j^T$
$F = \tilde{C} - C$	$\eta e_i \hat{b}_j^T$	$\eta a_i e_j^T$	$\eta e_i e_j^T$
$\Vert F\Vert_\infty$	$\lvert\eta\rvert \Vert\hat{b}_j^T\Vert_\infty$	$\lvert\eta\rvert \Vert a_i\Vert_\infty$	$\lvert\eta\rvert$
$d = Fw$	$\eta e_i \hat{b}_j^T w$	$\eta \omega_j a_i$	$\eta \omega_j e_i$
$\Vert d\Vert_\infty$	$\lvert\eta\rvert \lvert\hat{b}_j^T w\rvert$	$\lvert\eta\rvert \lvert\omega_j\rvert \Vert a_i\Vert_\infty$	$\lvert\eta\rvert \lvert\omega_j\rvert$
$e^T = v^T F$	$\eta \nu_i \hat{b}_j^T$	$\eta v^T a_i e_j^T$	$\eta \nu_i e_j^T$
$\Vert e^T\Vert_\infty$	$\lvert\eta\rvert \lvert\nu_i\rvert \Vert\hat{b}_j^T\Vert_\infty$	$\lvert\eta\rvert \lvert v^T a_i\rvert$	$\lvert\eta\rvert \lvert\nu_i\rvert$
criterion	$\Vert e^T\Vert_\infty (= \lvert\nu_i\rvert \Vert F\Vert_\infty)$	$\Vert d\Vert_\infty (= \lvert\omega_j\rvert \Vert F\Vert_\infty)$	$\Vert d\Vert_\infty (= \lvert\omega_j\rvert \Vert F\Vert_\infty)$ or $\Vert e^T\Vert_\infty (= \lvert\nu_i\rvert \Vert F\Vert_\infty)$

Table 1: Some measurements and error detection criteria.

Right-sided error detection criterion

Consider now the computation of $d = \tilde{C}w - Cw$, where w is a vector with entries $\omega_i = 1$, $i = 1, \ldots, n$. From Table 1 we see that *if* the corruption is in matrix B or C, $\|d\|_\infty = \|F\|_\infty$. As we do not have C, but a possibly corrupted approximation $\tilde{C}$, we use $A(Bw)$ instead of Cw in the computation of d; only three matrix-vector multiplications are then required to compute d. These matrix-vector multiplications are cheap relative to a matrix-matrix multiplication. Computing d and its norm is exactly the procedure suggested in [13].

However, if the corruption occurs in A, $\|d\|_\infty = |\eta||\hat{b}_j^T w|$, which can be small even if $\|F\|_\infty$ is large. In particular, if the elements of the j–th row of B sum to zero, $\|d\|_\infty = 0$ regardless of the the magnitude $\|F\|_\infty$. While this is not likely to happen in practice, the method is clearly not bulletproof for detecting corruption in A. A simple example of a matrix encountered in practice which has entries in rows and/or columns that sum to zero is the matrix derived from a discretization of Poisson's equation using a five-point stencil.

We will refer to the error detection criterion which places checksum vector w on the right as a *right-sided* error detection criterion. This criterion is guaranteed to detect a single error introduced in B or C. It is highly likely to detect such an error introduced in A.

Left-sided error detection criterion

Next, consider the computation $e = v^T\tilde{C} - v^T C$ where v is a vector with entries $\nu_i = 1$, $i = 1, \ldots, m$. From Table 1 we see that *if* the corruption is in matrix A or C, $\|e\|_\infty = \|F\|_\infty$. Again, by computing $v^T C = (v^T A)B$ we can obtain e with only three matrix-vector multiplications. In this case, if the corruption was in B, $\|e\|_\infty = |\eta||v^T a_i|$, which can be small even if $\|F\|_\infty$ is large. In particular, if the elements of the i–th column of A sum to zero, $\|e\|_\infty = 0$. Thus, the method is clearly not completely full-proof for detecting corruption of B.

We will refer to the error detection criterion which places checksum vector w on the left as a *left-sided* error detection criterion. This criterion is guaranteed to detect a single error introduced in A or C. It is highly likely to detect such an error introduced in B.

Two-sided error detection criterion

Clearly, in order to guarantee the detection of the corruption of a single element in one of the three matrices, one must compute $\|d\|_\infty$ if the error is in either B or C, and $\|e\|_\infty$ if the error is in either A or C.

3.2 Tolerance threshold and round-off errors

Unfortunately, computers are not equipped to deal with infinite precision arithmetic and rounding errors due to finite precision arithmetic will occur. In our error detection setting this means that, even if no error is introduced in any of the matrices, it may well be the case that $\|\tilde{C} - C\| \neq 0$.

Round-off error analysis of matrix operations has been a classic area of numerical analysis for the last half century. A result found in standard textbooks (e.g., [7]) is that for an implementation of the matrix product $C = AB$, based on *gaxpy*, *dot product*, or *outer product* computations, the computed results, $\mathrm{fl}(AB)$, satisfies

$$\|\mathrm{fl}(AB) - AB\|_\infty \leq \max(m, n, k)\ \mathbf{u}\ \|A\|_\infty \|B\|_\infty + O(\mathbf{u}^2),$$

where $\mathbf{u}$ is the unit round-off of the machine (the difference between 1 and the next larger floating-point number representable in that machine).

Therefore, our error detection mechanism should declare that an error has occured when

$$\|d\|_\infty > \tau\ \|A\|_\infty \|B\|_\infty \quad \text{or} \quad \|e^T\|_\infty > \tau\ \|A\|_\infty \|B\|_\infty,$$

with $\tau = \max(m, n, k)\ \mathbf{u}$.

These results on thresholds for detecting errors merely reiterate the observations made in [13].

3.3 Specialization to our situation

As mentioned in Section 2, in the specific situation we are trying to address a corruption occurs primarily when data reside in the L1 cache of the processor. Thus this corruption does not necessarily persist during the entire matrix-matrix multiplication. Therefore, it may be more informative to view matrices C, A, and B partitioned as follows:

$$C = \left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline \vdots & \ddots & \vdots \\ \hline C_{M1} & \cdots & C_{MN} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & \ddots & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right),$$

where C_{ij} is $m_i \times n_j$, A_{ip} is $m_i \times k_p$, and B_{pj} is $k_p \times n_j$.

Now C_{ij} is computed as a sequence of updates $C_{ij} \leftarrow A_{ip}B_{pj} + C_{ij}$ and the corruption will be encountered in exactly one such update. In other words, for one tuple of indices (i, j, p) one of the operands is corrupted by changing one element. Let us assume that B_{pj} is corrupted by $\eta e_r e_s^T$. Then the computed matrix $\tilde{C}$ is equal to C except in the (i, j) block, which equals $C_{ij} + \eta a_r^{(i,p)} e_s^T$, where $a_r^{(i,p)}$ denotes the r–th column of A_{ip}. If w again equals the vector of all ones, $\|\tilde{C} - C\|_\infty = |\eta| \|a_r^{(i,p)}\|_\infty$ and $\|\tilde{C}w - Cw\|_\infty = |\eta| \|a_r^{(i,p)}\|_\infty$. It follows that the right-sided detection criterion for detecting errors in B or C still works. The theory behind the left-sided and two-sided detection criteria can be extended similarly.

4 Towards a Practical Implementation

In this section we deal with two issues concerning the practical implementation of a fault-tolerant high-performance matrix-matrix multiplication kernel. First, in addition to error detection, we must also be able to correct any errors that are exposed. Second, in order to maintain high-performance, we must let the theory guide us to a scheme that imposes as little overhead as is possible.

Consider $C = \alpha AB + \beta C$ where C, A, and B have dimensions $m \times n$, $m \times k$ and $k \times n$, respectively. The cost of this operation is given by $2mnk$ floating point operations (FLOPs).

4.1 Right-sided error detection method

Thus, a simple approach is to compute $D = AB$, and check the computed $\tilde{D}$ by testing if $\|\tilde{D}w - A(Bw)\|_\infty < \tau\ \|A\|_\infty \|B\|_\infty$. If the condition is met, then $C \leftarrow \alpha D + \beta C$; otherwise D is recomputed. (Note: our assumption is that a *copy* of A or B is corrupted in some level of cache memory. Thus, the recomputation can use the original data in A and B.) If a more stringent threshold is used a false error due to roundoff can occur. In this case one can check if $\|\tilde{D}w - A(Bw)\|_\infty$ is exactly equal twice in a row in which case C is updated since this would indicate that the scheme resulted in a false detection due to round-off error.

The overhead from error detection is $2mn$ flops for forming $\tilde{D}w$ and $2kn + 2mk$ flops for forming $A(Bw)$ for a total of $2mn + 2kn + 2mk$ flops. In addition, the computations of $\|A\|_\infty$ and $\|B\|_\infty$ cost $O(mk)$ and $O(kn)$, respectively. If even a single error is detected, the cost of the operation doubles. Also storage for D, mn floating point numbers, is required.

4.2 Left-sided error detection method

A simple approach is to again compute $D = AB$, and check the computed $\tilde{D}$ by testing if $\|v^T\tilde{D} - (v^TA)B\|_\infty < \tau\ \|A\|_\infty \|B\|_\infty$. If the condition is met, then $C \leftarrow \alpha D + \beta C$; otherwise D is recomputed. If $\|v^T\tilde{D} - (v^TA)B\|_\infty$ is exactly equal twice in a row, C is updated since it is assumed that a corruption was erroneously detected.

A more sophisticated approach partitions B, C, and D as

$$(1) \qquad B = \left(B_1 \mid \cdots \mid B_N \right), C = \left(C_1 \mid \cdots \mid C_N \right), \text{ and } D = \left(D_1 \mid \cdots \mid D_N \right),$$

and computes $D_j = AB_j$. After each such computation, the magnitude of $\|v^T\tilde{D}_i - y^TB_i\|_\infty$ is checked, where $y^T = v^TA$ can be computed once and reused. As before, if no error is detected, $\tilde{C}_i \leftarrow \alpha\tilde{D}_i + \beta C_i$; otherwise D_i is recomputed. Now only workspace for one D_i is required and fewer computations need to be repeated when an error is detected. (Note that this is not possible for the right-sided approach since for each B_iw the product $A(B_iw)$ must be computed, which is expensive when B_i has few columns, as it is in our implementation described in the experimental section.)

Given a column partition of matrices D_j and B_j of width n_b, the overhead from error detection is now $2mk$ flops for forming $y^T = v^T A$, $2mn_b$ flops for forming $v^T \tilde{D}_j$ and $2kn_b$ flops for forming $v^T B_j$. Taking into account that n/n_b panels of D must be computed, the total overhead becomes $2mn+2kn+2mk$ flops, equivalent to the cost of the right-sided error detection scheme above. In addition, the computations of $\|A\|_\infty$ and $\|B_j\|_\infty$, $j = 1, \ldots, N$, cost $O(mk)$ and $O(kn)$, respectively. If a single error is detected during the update of C, only $2mn_bk$ flops are repeated. In this case, only storage for one panel D_j, mn_b floating point numbers, is required.

4.3 Two-sided error detection

Naturally the two above mentioned techniques can be combined to yield a two-sided error detection method. Here all of D is computed using a left-sided error detection method, after which a right-sided error detection method is used to verify that no undetected errors slipped by. If no errors are detected, C is appropriately updated.

The computational cost of two-sided error detection is exactly twice that of the one-sided error detection methods. Storage for all of D is required, or mn floating point numbers. However, most of the time the left-sided error detection scheme will detect errors and thus the overhead for correcting a single error is only $2mn_bk$ flops.

4.4 Reducing overhead

Even in the case where no error is ever detected, the above schemes, particularly the right- and two-sided approaches, carry a considerable overhead in required workspace. In addition, if an error is detected for the methods, the cost of recomputation can double the overall cost of the matrix-matrix multiplication. In this section we discuss how both of these overheads can easily be overcome.

Specifically, partition C, A, and B as

$$(2) \quad C = \left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline \vdots & \ddots & \vdots \\ \hline C_{M1} & \cdots & C_{MN} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & \ddots & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right)$$

where C_{ij} is $m_i \times n_j$, A_{ip} is $m_i \times k_p$, and B_{pj} is $k_p \times n_j$. (While this partitioning looks remarkably like the one in Section 3.3, the discussion in that section has no bearing on the discussion below.) Then C can be computed by a scaling $C \leftarrow \beta C$ followed by updates $C_{ij} \leftarrow \alpha A_{ip} B_{pj} + C_{ij}$, $i = 1, \ldots, M$, $j = 1, \ldots, N$, $p = 1, \ldots, K$. Each of these individual updates can use the error detection schemes described above. Now workspace can be greatly reduced as can the cost of a recomputation. Moreover, there are a number of opportunities for the reuse of results $B_{pj}w$, $v^T A_{ip}$, $\|B_{pj}\|_\infty$, and $\|A_{ip}\|_\infty$, where now w and v have length n_j and m_i, respectively.

Notice that the proposed error detection and correction scheme can now handle multiple errors, as long as only one error occurs during the computation $A_{ip}B_{pj}$.

5 An Actual Implementation

In this section we briefly outline our implementation of the ideas presented above.

We start by describing a high-performance implementation of matrix-matrix multiplication, ITXGEMM [8], developed at UT-Austin in collaboration with Dr. Greg M. Henry at Intel Corp. To understand how ITXGEMM uses hierarchical memory to attain high performance recall that the memory hierarchy of a modern microprocessor is often viewed as a pyramid (see Fig. 1). At the top of the pyramid there are the processor registers, with extremely fast access. At the bottom, there is disk and even slower media. As one goes down the pyramid, the amount of memory increases along with the time required to access that memory.

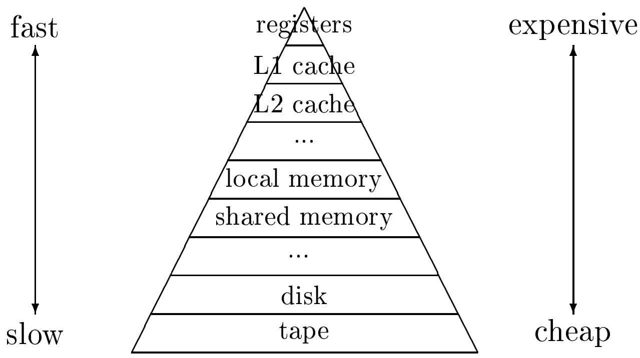

Figure 1: Hierarchical layers of memory.

As is well-known, processor speed has been increasing much faster than memory speed and it is thus memory bandwidth that hinders the speed attained in practice for a given operation. Fortunately, matrix-matrix multiplication involves $2mnk$ flops and only $2mn+mk+kn$ data items. Thus, by carefully moving data between layers of memory, high-performance can be attained. *Note that the cost of error detection is of the same order as the cost for loading and storing to and from a memory layer.*

The particular implementation of matrix-matrix multiplication in ITXGEMM, which we modified as part of this research, partitions C, A, and B as in (2). The partitioning scheme used for A is selected so that A_{ip} fills a large part of the L2 cache. For the architecture chosen for this testing, an Intel Pentium$^{\mathrm{TM}}$ III, the optimal partitioning turns out to be

Method	Overhead ($m_i = k_p = 128$, $n_j = 512$, $b = 8$) $m = n = 512$, $k = 128$ Detection	Correction	$m = n = k = 512$ Detection	Correction
right-sided	2.2%	25%	2.2%	6%
left-sided	2.2%	0.4%	2.2%	0.1%
two-sided	4.4%	0.4%	4.4%	0.1%

Table 2: Theoretical overhead for error detection and correction.

$m_i = k_p = 128$. Then, B is partitioned so that a reasonable amount of workspace is required for our right-sided error detection scheme. In particular, we chose $n_j = 512$. This means that the matrices are partitioned exactly as in (1) and updated as required by the left-sided error detection scheme, with $n_b = 8$. Code for error detection and correction was a straightforward addition to an implementation that naturally blocked for efficient utilization of the L1 and L2 caches of the Pentium$^{\text{TM}}$ III processor.

If we consider all floating point operations to be equal and we count the cost of computing the norm of an $m \times n$ matrix as mn flops, we expect the ratios of overhead to useful computation shown in Table 2. The overhead for correction is for the case when exactly one corruption occurs during the entire computation. This correction overhead scales linearly with the number of corruptions. The cost per flop of a matrix-vector multiplication is often an order of magnitude greater than the cost per flop of a matrix-matrix multiplication. Thus the above analysis for the cost of error detection may be optimistic by an order of magnitude. On the other hand, as mentioned, there are opportunities for amortizing the cost of the computation of matrix-vector multiplies and norms of matrices which are not taken into account in the above analysis.

6 Experimental Results

All our experiments were performed on a Intel Pentium$^{\text{TM}}$ III processor with a 650 MHz clockrate, 16 Kbytes of L1 data cache and 256 Kbytes of L2 cache, using IEEE double-precision floating-point arithmetic ($\mathbf{u} \approx 2.2 \times 10^{-16}$).

6.1 Fault-tolerance under simulated fault conditions

In order to evaluate the reliability of our error detection and correction techniques we decided to mirror in our experiments what we expect to be a more realistic fault condition behavior in practice. Thus, instead of introducing an error either in A or B before the computation starts, we introduce the error before one of the updates of the form $C_{ij} \leftarrow \alpha A_{ip} B_{pj} + \beta C_{ij}$ is computed. The exact update, the entry were the error appeared (including the matrix, A or B), and its magnitude are randomly determined.

We do not analyze the case in which the error appears in C since, as stated in our theory (see Table 1), that error will always be detected using any of the detection methods, (at least, as long as it makes a non-negligible difference in the result).

The error detection mechanisms performed exactly as expected:

- All significant errors that were introduced in matrix A were detected by the left-sided sided detection method.
- All significant errors that were introduced in matrix B were detected by the right-sided sided detection method.
- All significant errors that were introduced in matrices A or B were detected by the two-sided sided detection method.
- In practice both left- and right-sided methods detected significant errors introduced in either A or B.
- Whenever we created a matrix A such that the elements in individual columns added to zero, the left-sided detection method had trouble detecting errors introduced in B.
- Whenever we created a matrix B such that the elements in individual rows added to zero, the right-sided detection method had trouble detecting errors introduced in A.

6.2 Performance evaluation

Next, we evaluated the overhead introduced in practice by our error detection/correction techniques. We added the error detection and correction mechanisms described in the previous sections to the implementation of matrix-matrix multiplication described in ITXGEMM. In [9] we show that this implementation (without error detection and correction) is highly competitive with other efforts (e.g. [14]) to provide high-performance matrix-matrix multiplication for the Intel Pentium$^{\text{TM}}$ III processor.

We report results for the following fault-tolerant matrix-matrix multiplication implementations:

- L/R/2-sided detect: ITXGEMM-based implementation with left/right/two-sided detection.
- L/R/2-sided correct: ITXGEMM-based implementation with left/right/two-sided detection and correction.