A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units

Braun, Claus; Halder, Sebastian; Wunderlich, Hans-Joachim

Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14) Atlanta, Georgia, USA, 23-26 June 2014

doi: http://dx.doi.org/10.1109/DSN.2014.48

Abstract: Graphics processing units (GPUs) enable large-scale scientific applications and simulations on the desktop. To allow scientific computing on GPUs with high performance and reliability requirements, the application of software-based fault tolerance is attractive. Algorithm-Based Fault Tolerance (ABFT) protects important scientific operations like matrix multiplications. However, the application to floating-point operations necessitates the runtime classification of errors into inevitable rounding errors, allowed compute errors in the magnitude of such rounding errors, and into critical errors that are larger than those and not tolerable. Hence, an ABFT scheme needs suitable rounding error bounds to detect errors reliably. The determination of such error bounds is a highly challenging task, especially since it has to be integrated tightly into the algorithm and executed autonomously with low performance overhead. In this work, A-ABFT for matrix multiplications on GPUs is introduced, which is a new, parallel ABFT scheme that determines rounding error bounds autonomously at runtime with low performance overhead and high error coverage.

Preprint

A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units

Claus Braun, Sebastian Halder, and Hans-Joachim Wunderlich

Institute of Computer Architecture and Computer Engineering, University of Stuttgart
Pfaffenwaldring 47, D-70569, Germany, Email: {braun,wu}@informatik.uni-stuttgart.de

***Abstract*—Graphics processing units (GPUs) enable large-scale scientific applications and simulations on the desktop. To allow scientific computing on GPUs with high performance and reliability requirements, the application of software-based fault tolerance is attractive. Algorithm-Based Fault Tolerance (ABFT) protects important scientific operations like matrix multiplications. However, the application to floating-point operations necessitates the runtime classification of errors into inevitable rounding errors, allowed compute errors in the magnitude of such rounding errors, and into critical errors that are larger than those and not tolerable. Hence, an ABFT scheme needs suitable rounding error bounds to detect errors reliably. The determination of such error bounds is a highly challenging task, especially since it has to be integrated tightly into the algorithm and executed autonomously with low performance overhead.**

In this work, *A-ABFT for matrix multiplications on GPUs* is introduced, which is a new, parallel ABFT scheme that determines rounding error bounds autonomously at runtime with low performance overhead and high error coverage.

***Keywords*-Algorithm-Based Fault Tolerance, Rounding Error Estimation, GPU, Matrix Multiplication**

I. Introduction

Scientific computing and simulation technology are constantly gaining importance in many fields of research, industrial development, and even daily life. Computer-based simulations ("*in silico*") are increasingly used to substitute practical scientific experiments since they are often faster, cheaper, and in many cases even more comprehensive than experimental installations [1]. Most scientific computing applications have two major characteristics in common: First, they are typically associated with extensive runtimes and a high computational demand, which push even latest computing systems to their limits. Second, they have strict requirements regarding their reliability. Computed results have to be trustworthy to draw reliable conclusions, which can have direct influence on scientific, economical, social or political processes. Therefore, *dependable scientific computing* represents a major challenge for scientists, hardware architects and software developers.

General-purpose computations on Graphics Processing Units (GPUs) [2] have become increasingly popular in recent years, since latest generation GPUs deliver tremendous floating-point performance at very low cost. This offers far-reaching and new possibilities to bring complex scientific simulations from very different domains like EDA [3, 4], biology [5] or thermodynamics [6] to the desktop with substantial speedups in the order of several magnitudes. However, like most CMOS semiconductor devices manufactured in 28nm technology and below, GPUs are increasingly prone to transient effects [7], process variations and latent defects, as well as stress and aging mechanisms [8]. In contrast to traditional high-performance computing (HPC) systems, where high effort and cost are spent to ensure reliability through hardware- and software-based fault tolerance measures [9], GPUs are still designed and manufactured with the graphics and multimedia mass-market in mind.

As a consequence, to achieve the goal of *dependable scientific computing on GPUs*, the application of software-based fault tolerance (SBFT) is highly attractive. Unfortunately, the integration of SBFT into scientific GPU applications is often a challenging task, especially when low performance overhead and transparent operation without user interaction have to be achieved.

Algorithm-Based Fault Tolerance (ABFT) [10] encodes input data with checksums before the data are processed by modified algorithms that produce encoded results. To check the results for errors, new checksums are computed on the result data and compared to the original ones that went through the algorithm. Checksum mismatches are used to detect and locate errors. ABFT has proven to allow the effective and efficient protection of important scientific kernels, for instance from linear algebra (e.g. matrix multiplication, LU decomposition [10, 11], QR factorization [12], and singular value decompositions [13]). Some of these ABFT schemes have also been adapted for GPU accelerator architectures [14–17].

Since GPUs are floating-point accelerators, their results are inevitably prone to *rounding errors*. This generates an extraordinary challenge for ABFT, because now, the check procedure has to cope with rounding errors in the checksums, which make a fast and direct comparison impossible.

Hence, suitable rounding error bounds are required that enable the efficient and effective runtime classification of errors into *inevitable rounding errors*, *tolerable compute errors in the magnitude of the rounding errors*, and *intolerable critical compute errors* that are larger than the two aforementioned types of errors. Although there exist different approaches to tackle this problem, none of them allows a fully autonomous operation of ABFT, which however, is the key to a successful use of ABFT on GPUs for dependable scientific computing.

In this work, *A-ABFT for matrix multiplications on GPUs* is introduced, which is a new parallel ABFT scheme that determines rounding error bounds for the runtime error classification autonomously. *A-ABFT* is based on a probabilistic model for the rounding error distribution in floating-point computations and utilizes the parallel compute power of GPUs for the determination of error bounds with low performance overhead. Although *A-ABFT* is introduced on the example of an ABFT matrix multiplication, the approach itself is much more general and can be extended to other operations as well. As a by-product, *A-ABFT* is able to deliver error functions or rounding error analyses for the performed operation with little additional overhead. However this is not in the scope of this paper. In contrast to most existing approaches, *A-ABFT* enables the fully autonomous operation without any calibration runs or external intervention by the user.

II. ABFT for Matrix Operations on GPUs

ABFT for matrix operations has been introduced in [10]. Given the $m \times n$ matrix A and the $n \times q$ matrix B

$$A = \begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{bmatrix}, \quad B = \begin{bmatrix} b_{1,1} & \dots & b_{1,q} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \dots & b_{n,q} \end{bmatrix}.$$

ABFT encodes matrix A into a $m+1 \times n$ *column-checksum matrix* A_{cc}:

$$A_{cc} = \begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \\ \boldsymbol{a_{m+1,1}} & \dots & \boldsymbol{a_{m+1,n}} \end{bmatrix}, \quad \boldsymbol{a_{m+1,j}} = \sum_{i=1}^{m} a_{i,j} \tag{1}$$

and matrix B into a $n \times q+1$ *row-checksum matrix* B_{rc}:

$$B_{rc} = \begin{bmatrix} b_{1,1} & \dots & b_{1,q} & \boldsymbol{b_{1,q+1}} \\ \vdots & \ddots & \vdots & \vdots \\ b_{n,1} & \dots & b_{n,q} & \boldsymbol{b_{n,q+1}} \end{bmatrix}, \quad \boldsymbol{b_{j,q+1}} = \sum_{j=1}^{q} b_{i,j}. \tag{2}$$

For the matrix multiplication, the result $C_{fc} = A_{cc} \cdot B_{rc}$ is a $m+1 \times q+1$ *full-checksum matrix* with an additional checksum row and column respectively:

$$C_{fc} = \begin{bmatrix} c_{1,1} & \dots & c_{1,q} & \boldsymbol{c_{1,q+1}} \\ \vdots & \ddots & \vdots & \vdots \\ c_{m,1} & \dots & c_{m,q} & \boldsymbol{c_{m,q+1}} \\ \boldsymbol{c_{m+1,1}} & \dots & \boldsymbol{c_{m+1,q}} & \boldsymbol{c_{m+1,q+1}} \end{bmatrix}. \tag{3}$$

For the check, new checksums $c^*_{i,j}$ are computed on the result data and compared to the checksums that went through the linear operation to detect errors. Errors can be located at the intersection of mismatching checksum rows and columns. For the column and row checksums this leads to

$$c^*_{m+1,j} = \sum_{i=1}^{m} c_{i,j}, \quad \text{and } c^*_{i,q+1} = \sum_{j=1}^{q} c_{i,j} \tag{4}$$

with $1 \leq j \leq q$ and $1 \leq i \leq m$. In the error free case, the original and the reference checksum elements have to be equal

$$\boldsymbol{c_{m+1,j}} = c^*_{m+1,j}, \text{ and } \boldsymbol{c_{i,q+1}} = c^*_{i,q+1}. \tag{5}$$

With rounding errors, appropriate error bounds ε are required for the error detection. This changes (5) to

$$|c^*_{m+1,j} - \boldsymbol{c_{m+1,j}}| < \varepsilon_j \text{ and } |c^*_{i,q+1} - \boldsymbol{c_{i,q+1}}| < \varepsilon_i. \tag{6}$$

Error bounds that are chosen too tight will cause *false-positive* detections, which increase the overhead and affect the performance on the GPU negatively, because they trigger unnecessary corrections. In contrast, error bounds that are chosen too loose will allow significant errors to slip through the error check (*false-negative*) and affect the final result.

Contemporary GPUs are tightly coupled many-core processors that gain high floating-point performance through very large numbers of concurrently executing threads. To ease the handling of many threads and the development of parallel GPU code (kernels), the abstraction of thread blocks is often used. A thread block describes a group of threads that are scheduled together to execute the same code, following the *single instruction, multiple data* (SIMD) paradigm. ABFT matrix multiplications on GPUs are performed in a block-based manner. The input matrices A and B are subdivided into rows and columns of smaller, square sub-matrices. The dimensions of these sub-matrices typically match the dimensions of the thread blocks. To compute a sub-matrix of the result matrix C, a row of sub-matrices from A and a column of sub-matrices from B are processed by a thread block. Details on efficient block-based GPU matrix multiplications can be found in [18, 19].

The *A-ABFT* matrix multiplication uses a partitioned encoding scheme [20], which encodes the sub-matrices of A and B with a block size of $BS \times BS$. Figure 1 shows the partitioned encoding for the column-checksum matrix A_{cc} and the row-checksum matrix B_{rc}. The complete

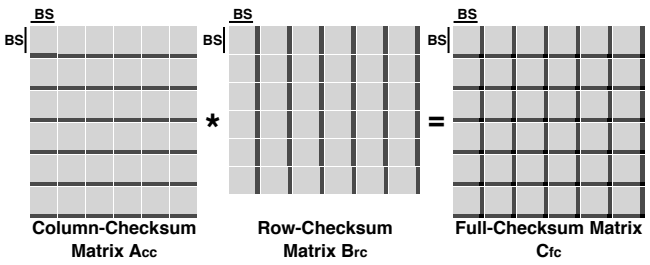

Figure 1. Block-based ABFT with partitioned encoding for row and column checksums of the sub-matrices.

algorithm for the *A-ABFT* matrix multiplication is given in Section VI, together with the extensions required to perform fault injection experiments (see Algorithm 3).

III. State of the Art

Different approaches for the determination of rounding error bounds in the ABFT checking procedure have been published. In [21] and [22] the experimental evaluation of error bounds has been proposed by performing multiple calibration runs of the target operation on similar data sets. An initial error bound is set and increased after each operation until no more false-positives are detected. Besides the high computational and time overhead for such calibration runs, the determined error bounds are dependend on the problem size and very likely to fail if slightest changes happen to the characteristic of the input data. This makes such calibration approaches highly unattractive and prohibits a real autonomous operation of ABFT schemes without external user intervention.

For matrix multiplications, [23] proposed to treat the mantissa of the involved matrix elements as integers and to use the integer ALU to check the operation of the floating-point multiplication. Although this approach covers the multiplication step within an ABFT matrix multiplication, it leaves out the floating-point addition and it only detects errors within the lower bits of mantissa products. In [24], the authors extended this approach with floating-point checksums for the addition, which still makes appropriate error bounds for the comparison necessary.

Another option for the determination of rounding error bounds is the evaluation of classic analytical error estimations. Inter alia, such techniques are described in [25] and [26]. Besides the computational evaluation overhead, these approaches are in most cases very pessimistic and hence often lead to error bounds which are too loose. This increases the risk of *false-negatives*.

In [27] the authors introduced input-independent tests to check results of different numerical operations for errors. The tests are based on analytical error bounds and involve the computation of matrix and vector norms. User interaction is required to select the optimal test for a given operation, as well as the appropriate parameters.

Based on the aforementioned analytical rounding error estimates, a simplified error analysis (SEA) approach for ABFT is introduced in [28]. The authors neglect second order rounding error terms and derive their bounds on the total error for groups of variables, like all the elements within a row or column of a matrix. SEA for ABFT matrix multiplications is presented by considering the matrix-vector product $A \cdot b = c$, where A is an $m+1 \times n$ column-checksum matrix, b is a $1 \times n$ column vector from matrix B and c is a $1 \times n$ column vector from the result matrix C. The SEA error bound is defined as:

$$\begin{aligned} |c_{n+1} - c^*_{n+1}| &< ((n+2m-2) \cdot \|b\|_2 \cdot \sum_{i=1}^{m} \|a_i\|_2 \\ &+ n \cdot \|a_{m+1}\|_2 \cdot \|b\|_2) \cdot \varepsilon_M, \end{aligned}$$

where c_{n+1} is the original column-checksum element that went through the multiplication and c^*_{n+1} is the reference column-checksum element. a_i denotes the i-th row vector and a_{m+1} the column-checksum vector from A. $\varepsilon_M = 2^{-t}$ is the machines unit rounding error and t is the number of mantissa bits. Although this approach reduces the overhead compared to the classic analytical bounds, it still contains the compute-intensive evaluation of numerous vector norms. Moreover, the difference between actual rounding errors and error bounds determined by this approach can be large.

IV. Probabilistic Rounding Error Determination for ABFT on GPUs

A-ABFT utilizes a probabilistic model for rounding errors occurring in floating-point operations to determine error bounds for the comparison of ABFT checksums. Since contemporary GPUs follow the IEEE-754 floating-point standard, the model is introduced in the following for base $B = 2$.

The core idea is to determine a confidence interval for a checksum element $\boldsymbol{c}$ comprising the mean value $\mathtt{EV}(\boldsymbol{c})$ and the variance $\mathtt{Var}(\boldsymbol{c})$. With the standard deviation $\sigma(\boldsymbol{c}) = \sqrt{\mathtt{Var}(\boldsymbol{c})}$ and a scaling factor ω, the confidence interval is defined as

$$[\mathtt{EV}(\boldsymbol{c}) - \omega \cdot \sigma(\boldsymbol{c}), \mathtt{EV}(\boldsymbol{c}) + \omega \cdot \sigma(\boldsymbol{c})]. \tag{7}$$

In [29], the distribution of rounding errors in different floating-point operations is investigated, starting with the four cardinal operations $\odot \in \{+, -, *, /\}$. Let

$$s \equiv a \odot b \equiv s^* + \epsilon \tag{8}$$

be the *exact* result of such an operation $\odot$, a and b t-digit floating-point numbers, and s^* the result computed in floating-point arithmetic, like on GPUs. The occuring rounding error is denoted by ϵ. Under the assumption that s and s^* have the same exponent E — the case where the exponents differ arises with about probability $O(2^{-t})$ — s and s^* are represented as

$$s = x \cdot 2^E \text{ and } s^* = x^* \cdot 2^E, \tag{9}$$

with the mantissas $x, x^* \in [\frac{1}{2}, 1]$. Then

$$\epsilon = (x - x^*) \cdot 2^E = \beta \cdot 2^E, \tag{10}$$

where β is called the *mantissa error*.

For all four cardingal floating-point operations $\odot$ it is assumed that the mantissas follow a *reciprocal distribution* on $[\frac{1}{2}, 1)$ (see Section IV-A). Under this assumption, a distribution for the mantissa error β can be derived and the mean $\mathtt{EV}(\beta)$, as well as the variance $\mathtt{Var}(\beta)$ can be determined. Following [29], these values can be used to compute the mean and variance of the occuring rounding error:

$$\mathtt{EV}(\epsilon) = sgn(s^*) \cdot 2^E \cdot \mathtt{EV}(\beta) \tag{11}$$

$$\mathtt{Var}(\epsilon) = 2^{2E} \cdot \mathtt{Var}(\beta) \tag{12}$$

$$E = \lceil log_2 |s^*| \rceil. \tag{13}$$

A. *Reciprocal Distribution of Mantissa Bits*

Benford's Law [30] describes the observation that mantissas of floating-point numbers in arbitrary data sets tend to follow the reciprocal distribution (base $B = 2$ case) given by

$$r(x) = \frac{1}{x \cdot ln(2)}, \quad x \in [\frac{1}{2}, 1). \tag{14}$$

In [31], Hamming showed that floating-point operations influence the distribution of the mantissas to follow the reciprocal distribution. The assumption taken in Section IV is further justified in [32, 33] and summarized in [34].

For the *A-ABFT* matrix multiplication $A_{cc} \cdot B_{rc} = C_{fc}$, each element $c_{i,j}$ of the result full-checksum matrix C_{fc} can be described by the inner product of the row vector a_i of the column-checksum matrix A_{cc} and the column vector b_j of the row-checksum matrix B_{rc}:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j} = \sum_{k=1}^{n} \hat{c}_k. \tag{15}$$

The following sections introduce the probabilistic rounding error determination for these inner products.

B. *Probabilistic Error Bounds for Summations*

For the summation of n intermediate products $c_1, ..., c_n$ the following recursion applies for the GPU-compute results and the rounding error ϵ_k:

$$s^*_{k+1} + \epsilon_{k+1} = s^*_k + \hat{c}^*_{k+1}, \text{ for } k = 1, ..., n-1. \tag{16}$$

The recursion for the difference between the exact and the GPU-computed result $\Delta s_k = s_k - s^*_k$ is $\Delta s_{k+1} = \Delta s_k + \epsilon_{k+1}$ and can be expressed as

$$\Delta s_n = \sum_{k=2}^{n} \epsilon_k. \tag{17}$$

For the probabilistic rounding error determination of the sum s_n the confidence interval based on the mean $\mathtt{EV}(\Delta s_n)$ and the variance $\mathtt{Var}(\Delta s_n)$ has to be determined with

$$\mathtt{EV}_{\mathtt{Sum}}(\Delta s_n) = \sum_{k=2}^{n} \mathtt{EV}(\epsilon_k) \tag{18}$$

$$\mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) = \sum_{k=2}^{n} \mathtt{Var}(\epsilon_k). \tag{19}$$

For the addition and subtraction of two floating-point numbers we have:

$$\mathtt{EV}(\beta) = 0 \tag{20}$$

and

$$\mathtt{Var}(\beta) \leq \frac{1}{8} \cdot 2^{-2t}. \tag{21}$$

With the Equations (11) and (12) the mean and variance are

$$\mathtt{EV}_{\mathtt{Sum}}(\Delta s_n) = 0 \tag{22}$$

and

$$\begin{aligned} \mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) &= \sum_{k=2}^{n} \mathtt{Var}(\epsilon_k) && (23)\\ &\leq \sum_{k=2}^{n} 2^{2E_k} \cdot \mathtt{Var}(\beta) && (24)\\ &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} 2^{2E_k}, && (25) \end{aligned}$$

where 2^{2E_k} denotes the exponent of the intermediate result s^*_k after addition of the k-th addend. With normalized floating-point numbers, we have $E_k \leq s^*_k$, which allows to consider Equation (25) for the intermediate results s^*_k of the summation:

$$\mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) \leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (s^*_k)^2. \tag{26}$$

For an upper bound y with $s^*_k \leq k \cdot y$, Equation (26) can be bounded by

$$\begin{aligned} \mathtt{Var}_{\mathtt{Sum}}(\Delta sn) &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (k \cdot y)^2 \\ &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \left(\frac{n \cdot (n+1) \cdot (2n+1)}{6} \right) \cdot y^2. \end{aligned}$$

With the standard deviation σ, the confidence interval is determined by

$$\mathtt{EV}(\Delta s_n) = 0 \tag{27}$$

and

$$\sigma(\Delta s_n) \leq \sqrt{\frac{n \cdot (n+1) \cdot (2n+1)}{48}} \cdot y \cdot 2^{-t}. \tag{28}$$

C. *Probabilistic Error Bounds for Inner Products*

For the inner product, the additional rounding error α_k caused by the multiplication $\hat{c}_k = a_{i,k} \cdot b_{k,j}$ has to be considered:

$$\hat{c}^*_k + \alpha_k = \hat{c}_k. \tag{29}$$

The recursion for the summation within the inner product is then $\Delta s_{k+1} = \Delta s_k + \epsilon_{k+1} + \alpha_{k+1}$ and can be expressed as

$$\Delta s_n = \sum_{k=2}^{n} \epsilon_k + \sum_{k=1}^{n} \alpha_k. \tag{30}$$

Analogous to the Equations 18 and 19, mean and variance can be defined as

$$\begin{aligned} \mathtt{EV}_{\mathtt{InProd}}(\Delta s_n) &= \mathtt{EV}_{\mathtt{Sum}}(\Delta s_n) + \mathtt{EV}_{\mathtt{Prod}}(\Delta s_n) && (31)\\ &= \sum_{k=2}^{n} \mathtt{EV}(\epsilon_k) + \sum_{k=1}^{n} \mathtt{EV}(\alpha_k) && (32) \end{aligned}$$

and

$$\begin{aligned} \mathtt{Var}_{\mathtt{InProd}}(\Delta s_n) &= \mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) + \mathtt{Var}_{\mathtt{Prod}}(\Delta s_n) \\ &= \sum_{k=2}^{n} \mathtt{Var}(\epsilon_k) + \sum_{k=1}^{n} \mathtt{Var}(\alpha_k). && (33) \end{aligned}$$

To complete the rounding error determination for the inner product, the mean and variance of the multiplication has to be considered. Following [29], mean and variance for the mantissa error in the multiplication/division of floating-point numbers with symmetric rounding is given as

$$\texttt{EV}(\beta) = \frac{1}{3} \cdot 2^{-2t} \tag{34}$$

and

$$\texttt{Var}(\beta) = \frac{1}{12} \cdot 2^{-2t}. \tag{35}$$

The contribution of the rounding error variance after k multiplications is the sum of all partial variances

$$\texttt{Var}_{\texttt{Prod}}(\Delta s_n) = \sum_{k=1}^{n} \texttt{Var}(\alpha_k). \tag{36}$$

For $y = a_d \cdot b_d$ with $d \in \{1, ..., n\}$ where $\texttt{Var}(\alpha_d)$ is maximal, Equation (37) can be bounded:

$$\texttt{Var}_{\texttt{Prod}}(\Delta s_n) \leq n \cdot \texttt{Var}(\alpha_d). \tag{37}$$

The variance is maximal if the exponent E of the multiplication's result is maximal:

$$\texttt{Var}(\alpha_d) = 2^{2E_d} \cdot \texttt{Var}(\beta) \geq 2^{2E_k} \cdot \texttt{Var}(\beta) = \texttt{Var}(\alpha_k). \tag{38}$$

y can be used as upper bound, which leads to

$$\begin{aligned} \texttt{Var}_{\texttt{Prod}}(\Delta s_n) &\leq n \cdot \texttt{Var}(\alpha_y) && (39)\\ &\leq n \cdot y^2 \cdot \texttt{Var}(\beta) && (40)\\ &= \frac{n}{12} \cdot 2^{-2t} \cdot y^2. && (41) \end{aligned}$$

Analog, we derive the mean value

$$\begin{aligned} \texttt{EV}_{\texttt{Prod}}(\Delta s_n) &= n \cdot y \cdot \texttt{EV}(\beta) && (42)\\ &\leq \frac{n}{3} \cdot 2^{-2t} \cdot y. && (43) \end{aligned}$$

With the variances and mean values from the summation and the multiplication, all components are determined that are required for the standard deviation of the inner product's rounding error:

$$\begin{aligned} \sigma(\Delta s_n) &= \sqrt{\texttt{Var}_{\texttt{InProd}}(\Delta s_n)} && (44)\\ &= \sqrt{\texttt{Var}_{\texttt{Prod}}(\Delta s_n) + \texttt{Var}_{\texttt{Sum}}(\Delta s_n)} && (45)\\ &\leq \sqrt{\frac{n \cdot (n+1) \cdot (n+\frac{1}{2}) + 2n}{24}} \cdot 2^{-t} \cdot y. \end{aligned}$$

D. Rounding, Truncation and Fused MAD

The probabilistic model for rounding errors in floating-point arithmetic applies for operations that are carried out with symmetric rounding, as well as truncation with only minor changes. Modern GPUs and CPUs that implement the IEEE-754-2008 standard often provide so-called *fused multiply-add* operations to allow for higher performance and higher accuracy. Such operations perform the multiplication with increased precision and round only the result of the addition. For these operations, the rounding error contribution of the multiplication is not present and hence only the probabilistic error bound for the summation is used.

E. Determination of the Upper Bound

To apply the probabilistic rounding error determination for the checksum comparison of a result matrix element $c_{i,j}$, an upper bound $y_{i,j}$ has to be determined:

$$\forall c_k = a_{i,k} \cdot b_{k,j} : \quad y_{i,j} \geq c_k. \tag{46}$$

The determination of this upper bound depends on the indices i and j. The required row and column elements can be pre-processed in a sorting step. For each vector a_i from matrix A and each vector b_j from matrix B, the p elements with the largest absolute values and the corresponding indices are determined. Let A_{idx} be the set of the p elements with the largest absolute values from vector a_i and let B_{idx} be the set of these p elements from vector b_j. For any indices i and j the upper bound $y_{i,j}$ can be determined as the **maximum of the three cases**:

- $S = A_{idx} \cap B_{idx} \neq \emptyset$ and two large values from vector a_i and b_j are multiplied. $\Rightarrow y = max(|a_s \cdot b_s|)$ with $s \in S$.
- $S = A_{idx} \cap B_{idx} = \emptyset$, but the maximum of the p absolute values from vector a_i is multiplied with any of the elements from vector b_j. $\Rightarrow max(|a_r|) \cdot min(|b_s|)$, with $r \in A_{idx}$ and $s \in B_{idx}$.
- $S = A_{idx} \cap B_{idx} = \emptyset$, but the maximum of the p absolute values from vector b_j is multiplied with any of the elements from vector a_i. $\Rightarrow max(|b_s|) \cdot min(|a_r|)$, with $r \in A_{idx}$ and $s \in B_{idx}$.

The quality of the error bound can be improved by increasing the number p of considered largest absolute values. However, this also increases the computational overhead.

For each element $c_{i,j}$ of the full-checksum result matrix C_{fc}, this upper bound gives an expectation value and a variance. This value is directly applied for the column-checksum elements $c_{n+1,j}$ and the row-checksum elements $c_{i,p+1}$ in C_{fc}.

V. Parallel Implementation of A-ABFT on GPUs

As introduced in Section II, the proposed *A-ABFT* scheme for matrix multiplications is block-based and therefore integrates the runtime determination of the rounding error bounds into the checksum encoding and checking procedures. *A-ABFT* comprises of the following algorithmic steps:

1) **GPU kernel for checksum encoding and maximum absolute value determination:**
   a) Computation of column checksums for the sub-matrices of matrix A.
   b) Determination of the p largest absolute values of the row elements in each sub-matrix of matrix A.
   c) Computation of the row checksums for the sub-matrices of matrix B.
   d) Determination of the p largest absolute values of the column elements in each submatrix of matrix B.

2) **Computation of the matrix product**
3) **Reduction of blockwise determined largest absolute values to p global largest values per row/column**
4) **GPU kernel for error bounds determination and checksum comparison:**
   a) Determination of the rounding error bounds for the row and column checksum elements of each result sub-matrix of matrix C.
   b) Computation of the reference row and column checksums for each result sub-matrix of matrix C.
   c) Comparison of original and reference checksum elements using the determined rounding error bounds.

In the following, a detailed description is given for the encoding and checking kernels.

A. GPU Kernel for Checksum Encoding and Largest Absolute Value Determination

The encoding kernel combines the computation of row or column checksums with the first step of the rounding error bounds determination, which is to find the p largest absolute values and their indices within the rows or columns of the sub-matrix. It is invoked before the kernel for the computation of the matrix product starts. The kernel reduces expensive accesses to the global GPU device memory by loading the elements of the processed sub-matrix into the shared memory. As shown in Figure 2 for the column-checksum case with a block size of 5×5, the threads iterate over the sub-matrix elements from top to bottom and accumulate them into the respective checksum elements. After an element has been added, it is replaced in shared memory by its absolute value. In a second step, the threads

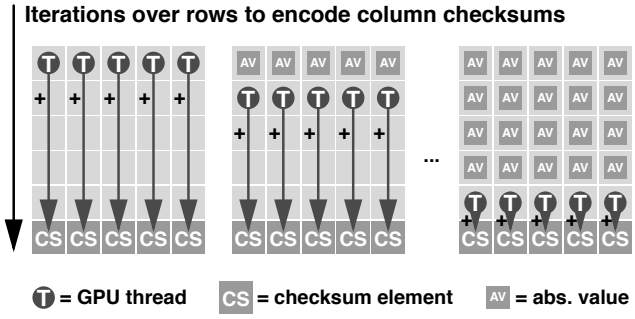

Figure 2. Column-checksum encoding for a block size of 5×5 with absolute value replacement.

iterate over the columns of the sub-matrix to find the p largest elements within each row and store the corresponding indices. This step is depicted in Figure 3. For the sub-matrices with the dimensions $BS \times BS$, the kernel is launched with thread blocks of size $BS \times 1$. The encoding for the row checksums works the same way. Algorithm 1 describes all performed steps in detail. Since the p largest

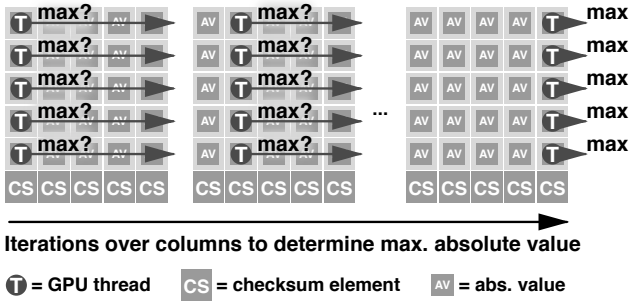

Figure 3. Search for p maximum absolute values and the corresponding indices.

Algorithm 1: Kernel for encoding the column-checksums of matrix A and the determination of the p elements with maximum absolute values per row for this block.

Input: padded matrix A, $numMax$
Output: encoded matrix A, $maxValues$, $maxValueIDs$

Launch Dimensions: $BS \times 1$ threads, one thread block for each $BS \times BS$ sub-matrix of A

Data: local registers sum, $maxVal$, $maxSum$
Data: shared $Asub[BS][BS]$, $localSums[BS]$

```
/* loop through rows of block                          */
for i ← 1 to BS do
    /* each thread calculates column checksum
       for its own column. tid is the thread's
       ID.                                             */
    load one element of sub-matrix of A into Asub[i][tid];
    sum ← sum + Asub[i][tid];
    Asub[i][tid] = abs(Asub[i][tid]);
end
write back column-checksum sum into A;
sync

sum ← abs(sum)
for m ← 1 to numMax do
    /* init local sums and current maximum             */
    maxVal ← 0;
    localSums[tid] ← abs(sum);

    /* search maximum value in row                     */
    for i ← 1 to BS do
        maxVal ← max(maxVal, Asub[tid][i]);
        update maxValueID;
    end

    /* search maximum value of previously
       calculated checksum entries                     */
    maxSum ← maxReduce(localSums);
    update maxSumID;

    /* write back maximum values, their location
       and exclude those values from the next
       round of calculation                            */
    write back maxValues and maxValueIDs;
    Asub[tid][maxID] ← 0;
    if maxSumID == tid then
        write back maxSum and maxSumID;
        sum ← 0;
    end
end
```

absolute values are determined within each sub-matrix, a global reduction step is required to reduce these $\frac{m}{BS} \cdot p$ largest absolute values to the required p per vector. This reduction kernel is executed in parallel to the matrix multiplication kernel and not further discussed at this point.

B. GPU Kernel for Error Bound Determination and Checksum Comparison

The checking kernel is invoked after the matrix multiplication has finished. It combines the computation of the rounding error bounds with the computation of the reference checksums and the final checksum comparison. The indices that have been stored by the encoding kernel are loaded and the possible combinations are checked. Depending on these combinations, the rounding error bounds are computed. After the error bounds, the reference checksums are computed on the result sub-matrix elements and then compared to the original checksum elements that went through the matrix multiplication. If the difference between these checksum elements is larger than the determined rounding error bound, an error is detected. Algorithm 2 shows the different steps in detail.

Algorithm 2: Kernel for computation of rounding error bounds and checking of result sub-matrices.

Input: matrix C, $maxA$, $maxB$, $maxAIDs$, $maxBIDs$, $numMax$
Output: corrected matrix C or error information
Launch Dimensions: $BS \times 1$ threads, one threadblock for each submatrix of C
Data: local registers $A_{idx}[numMax]$, $B_{idx}[numMax]$, max, sum, eps
Data: shared $Csub[BS][BS]$

```
/* load max indices                                          */
for i ← 1 to numMax do
    A_idx[i] ← maxAIDs; B_idx[i] ← maxBIDs;
end
/* check for combinations                                    */
max ← 0;
for m ← 1 to numMax do
    for n ← 1 to numMax do
        if A_idx[m] == B_idx[n] then
            max ← maxA[n] · maxB[m];
        end
    end
end
/* use minimum if no combination was found                   */
if max == 0 then
    max ← min(maxA) · min(maxB);
end
/* recalculate column checksums                              */
for i ← 1 to BS do
    load one element of submatrix of C into Csub[i][tid];
    sum ← sum + Csub[i][tid];
end
/* load checksum element of C, calculate
   epsilon and compare checksums                             */
ref ← columnChecksum(C, tid);
eps ← calculateEpsilon(max);
if abs(ref − sum) > eps then
    write back error location or start correction;
end
/* analogously recalculate and check the row
   checksums, all necessary data is already in
   the shared memory...                                      */
```

VI. Experimental Results

The proposed *A-ABFT* approach has been evaluated with respect to the computational performance, the quality of the determined rounding error bounds, and the achievable error coverage. The term quality refers in this context to the difference between the actual rounding error and the computed error bounds.

A. Performance

To evaluate the performance overhead of the introduced *A-ABFT* scheme due to the autonomous computation of rounding error bounds, a comparison has been performed against three software-based fault tolerance approaches. The first contender is a standard ABFT scheme for matrix multiplications on GPUs, whose error bounds have to be set manually by the user. This scheme has no additional overhead compared to an unprotected matrix multiplication besides the encoding and checking procedure. The second candidate is an ABFT scheme based on the simplified error analysis approach (SEA-ABFT) from [28]. This scheme determines the required rounding error bounds at runtime based on the computation of norms of the involved row and column vectors. The third candidate is a triple modular redundancy (TMR) approach, which performs the matrix multiplication three times and which compares the three results at the end. Hence, the TMR approach does not incorporate any overhead due to checksum encoding and comparison, or due to the computation of rounding error bounds. The experiments are performed over random input values and across matrix dimensions from 512×512 to 8192×8192. All experiments have been performed on an Nvidia K20C GPU accelerator in double precision. This accelerator is based on the Nvidia GK110 Kepler GPU architecture, offers 5GB of GDDR5 global device memory and 2496 processing cores, which deliver a peak double precision floating-point performance of about 1.17 TFLOPs.

Table I shows the results of this performance evaluation in GFLOPS. The standard ABFT matrix multiplication with manually set error bounds achieves the highest performance with a peak value of 942.6 GFLOPs for 8192×8192 matrices. However, this scheme is not able to operate autonomously and in a transparent way. The user has to know the full characteristics of the input data and suitable error bounds have to be set manually before each operation. In real applications, this is very often not possible.

The TMR matrix multiplication performs well for small matrices, which are often not of interest for large-scale scientific applications. For growing matrix dimensions, the expected overhead of TMR becomes clearly visible. For this

evaluation, a very basic TMR scheme has been implemented, which executes an identical kernel three times and which performs a direct comparison of the result matrices. While this is good for the performance in this evaluation, in real applications one would prefer to use three different kernels with different implementations to ensure different execution paths. This in turn would cause different rounding errors in the final results, which makes the direct comparison of the results impossible and which makes the computation of rounding error bounds necessary.

Table I
COMPARISON OF PERFORMANCE BETWEEN ABFT WITH FIXED ERROR BOUND, THE PROPOSED A-ABFT, SIMPLIFIED ERROR ANALYSIS ABFT, AND TMR.

MATRIX $[n \times n]$	ABFT [GFLOPS]	A-ABFT [GFLOPS]	SEA-ABFT [GFLOPS]	TMR [GFLOPS]
512	382.30	279.19	307.75	185.56
1024	659.02	514.17	499.53	322.22
2048	807.91	706.85	635.67	335.65
3072	872.93	772.64	657.28	339.33
4096	894.14	829.10	686.39	345.26
5120	924.38	848.43	690.51	344.95
6144	926.61	874.59	703.91	346.76
7168	944.50	885.23	705.51	347.68
8192	942.61	903.44	712.75	348.09

The simplified error analysis approach (SEA-ABFT) performs quite well with increasing matrix dimensions and reaches a peak performance of about 712.8 GFLOPS for 8129×8192 matrices. The lower performance compared to *A-ABFT* and the standard ABFT scheme is mainly due to the fact that the necessary computation of the vector norms uses only a small fraction of the available GPU threads. This suboptimal utilization of the GPU's compute resources leads to a noticable performance penalty.

With 903.4 GFLOPS for 8192×8192 matrices, the performance of *A-ABFT* almost reaches the level of the standard ABFT scheme and the gap between both approaches becomes smaller and smaller with increasing matrix dimensions. A completely unprotected matrix multiplication, which almost fully utilizes the GPU's hardware, delivered up to 1048.4 GFLOPS peak performance for 8192×8192 matrices in several tests. Although the proposed *A-ABFT* scheme incorporates the overhead of encoding and comparing ABFT checksums, as well as the autonomous determination of rounding error bounds – tasks which exhibit a rather low hardware utilization – it reaches 86.2% of this top performance. In other words, the overhead of *A-ABFT* can be as low as 13.8% while delivering a fault tolerant matrix multiplication and exceeding the performance of TMR and SEA-ABFT by far, especially for larger matrix dimensions.

B. *Quality of the Error Bounds*

To evaluate the quality of the computed rounding error bounds, multiple series of *A-ABFT* and SEA-ABFT matrix multiplications have been performed in double precision for random input values from the ranges -1.0 to 1.0, -100.0 to 100.0 as well as matrices with high value range dynamic generated by the formula

$$A = 10^{\alpha} \cdot U \cdot D_{\kappa} \cdot V^{T} \tag{47}$$

as proposed by [27]. For the probabilistic rounding error determination the parameter p has been set to $p = 2$. All experiments have been performed across matrix dimensions from 512×512 to 8192×8192. The determined rounding error bounds are compared against exact rounding errors that have been computed using GMP, a multi-precision floating-point library. The experiments have been performed on the GPU's host server using two Intel Xeon E5-2650 CPUs. Table II shows the results of the evaluation runs for random input values in the range of -1.0 to 1.0 and Table III shows the results for random input values from -100.0 to 100.0, respectively. Table IV shows the results for the paramters $\alpha = 0$ and $\kappa = 2$. The tables clearly state that the rounding

Table II
COMPARISON OF DETERMINED ROUNDING ERROR BOUNDS FOR THE RANDOM INPUT VALUE RANGE -1.0 TO 1.0.

MATRIX $[n \times n]$	AVG. RND. ERROR	AVG. A-ABFT	AVG. SEA-ABFT
512	$2.25 \cdot 10^{-14}$	$1.68 \cdot 10^{-11}$	$8.58 \cdot 10^{-10}$
1024	$4.53 \cdot 10^{-14}$	$4.88 \cdot 10^{-11}$	$3.30 \cdot 10^{-9}$
2048	$9.09 \cdot 10^{-14}$	$1.46 \cdot 10^{-10}$	$1.29 \cdot 10^{-8}$
3072	$1.35 \cdot 10^{-13}$	$2.77 \cdot 10^{-10}$	$2.88 \cdot 10^{-8}$
4096	$1.81 \cdot 10^{-13}$	$4.27 \cdot 10^{-10}$	$5.09 \cdot 10^{-8}$
5120	$2.25 \cdot 10^{-13}$	$6.21 \cdot 10^{-10}$	$7.95 \cdot 10^{-8}$
6144	$2.71 \cdot 10^{-13}$	$8.15 \cdot 10^{-10}$	$1.14 \cdot 10^{-7}$
7168	$3.17 \cdot 10^{-13}$	$1.06 \cdot 10^{-9}$	$1.56 \cdot 10^{-7}$
8192	$3.62 \cdot 10^{-13}$	$1.28 \cdot 10^{-9}$	$2.03 \cdot 10^{-7}$

Table III
COMPARISON OF DETERMINED ROUNDING ERROR BOUNDS FOR THE RANDOM INPUT VALUE RANGE -100.0 TO 100.0.

MATRIX $[n \times n]$	AVG. RND. ERROR	AVG. A-ABFT	AVG. SEA-ABFT
512	$2.22 \cdot 10^{-10}$	$1.61 \cdot 10^{-7}$	$8.65 \cdot 10^{-6}$
1024	$4.55 \cdot 10^{-10}$	$4.92 \cdot 10^{-7}$	$3.30 \cdot 10^{-5}$
2048	$9.07 \cdot 10^{-10}$	$1.48 \cdot 10^{-6}$	$1.29 \cdot 10^{-4}$
3072	$1.36 \cdot 10^{-9}$	$2.81 \cdot 10^{-6}$	$2.88 \cdot 10^{-4}$
4096	$1.81 \cdot 10^{-9}$	$4.27 \cdot 10^{-6}$	$5.10 \cdot 10^{-4}$
5120	$2.26 \cdot 10^{-9}$	$6.10 \cdot 10^{-6}$	$7.93 \cdot 10^{-4}$
6144	$2.71 \cdot 10^{-9}$	$8.15 \cdot 10^{-6}$	$1.14 \cdot 10^{-3}$
7168	$3.16 \cdot 10^{-9}$	$1.04 \cdot 10^{-5}$	$1.55 \cdot 10^{-3}$
8192	$3.62 \cdot 10^{-9}$	$1.29 \cdot 10^{-5}$	$2.03 \cdot 10^{-3}$

error bounds determined by the proposed *A-ABFT* scheme are typically two orders of magnitude closer to the exact rounding error, compared to the SEA-ABFT rounding error bounds. This enables *A-ABFT* to detect more errors than SEA-ABFT. For the *A-ABFT* approach, all experiments have been performed with a very conservative setting of 3σ for the rounding error determination. This parameter can also

Table IV
COMPARISON OF DETERMINED ROUNDING ERROR BOUNDS FOR THE RANDOM INPUT VALUE RANGE $\alpha = 0$, $\kappa = 2$.

MATRIX $[n \times n]$	AVG. RND. ERROR	AVG. A-ABFT	AVG. SEA-ABFT
512	$6.19 \cdot 10^{-11}$	$7.99 \cdot 10^{-8}$	$1.34 \cdot 10^{-6}$
1024	$2.44 \cdot 10^{-10}$	$5.12 \cdot 10^{-7}$	$1.02 \cdot 10^{-5}$
2048	$9.72 \cdot 10^{-10}$	$3.22 \cdot 10^{-6}$	$7.96 \cdot 10^{-5}$
3072	$2.20 \cdot 10^{-9}$	$9.51 \cdot 10^{-6}$	$2.69 \cdot 10^{-4}$
4096	$3.89 \cdot 10^{-9}$	$2.02 \cdot 10^{-5}$	$6.31 \cdot 10^{-4}$
5120	$6.04 \cdot 10^{-9}$	$3.61 \cdot 10^{-5}$	$1.22 \cdot 10^{-3}$
6144	$8.77 \cdot 10^{-9}$	$5.88 \cdot 10^{-5}$	$2.28 \cdot 10^{-3}$
7168	$1.20 \cdot 10^{-8}$	$8.82 \cdot 10^{-5}$	$4.08 \cdot 10^{-3}$
8192	$1.54 \cdot 10^{-8}$	$1.24 \cdot 10^{-4}$	$8.04 \cdot 10^{-3}$

be set to 2σ and σ and hence lead to error bounds that are even closer to the actual rounding error. However, these rounding error bounds are typically within the same order of magnitude, so we decided to report the "worst case" numbers in these tables.

C. *Error Detection*

The most important aspect of an ABFT scheme is the ability to detect errors reliably. The runtime classification of *A-ABFT* distinguishes therefore three classes of value errors that can occur during the computation:

1) *Inevitable rounding errors*, which are not counted as significant errors.
2) *Tolerable compute errors in the magnitude of the rounding errors*, which actually differ slightly from the correct result, but whose difference is insignificant for the final result.
3) *Intolerable critical compute errors that are larger than the rounding error* and which have to be detected and corrected.

Control errors can be covered by ABFT schemes to a certain extent, as long as they lead to value errors and as long as they do not prevent the ABFT-protected operation to finish execution, including checksum encoding and checking steps. However, control errors that alter the execution path are not in the scope of this work.

To evaluate the error detection capabilities of the *A-ABFT* scheme, fault injection experiments have to be performed. On GPUs with multiple streaming multiprocessors and large numbers of processing cores, simple fault injections into the global GPU device memory would potentially affect the results of all the processors that access the erroneous data word, and hence cause a rather significant error impact. Such errors are typically easier to detect for ABFT schemes. To generate local value errors within the streaming multiprocessors, which are harder to detect, the fault injection has to target single floating-point operations like the addition (fadd) or multiplication (fmul). Faults injected into such operations can lead to erroneous outputs unless they are not masked during the processing of the instruction. However, this kind of fault injection requires detailed knowledge about the algorithm and the kernel implementation. In case of the block-based matrix multiplication (see Sections II and V), the GPU kernel has been modified to support the injection of faults into different floating-point operations. Algorithm 3 gives a detailed overview of the steps performed for the matrix multiplication and the fault injection. In this algorithm, each thread calculates $RX \cdot RY$ elements of a $BN \cdot BM$ block of the result matrix and reduces memory latency issues by using shared memory blocks of size BK. These parameters can be adjusted for specific architectures to improve the performance of this kernel, but do not affect our fault injection method. For further details we refer to [19].

The addition of values is performed at two points: During the accumulation within the inner loop (*Inner Loop Addition*) and at the end of the algorithm, where the final results are summed up (*Final Sum Addition*). The multiplication of values is only performed within the inner loop (*Inner Loop Multiplication*). During the execution of the matrix multiplication, the fault injection routine randomly selects a streaming multiprocessor and one of the floating-point operations to inject a single- or a multi-bit flip. The injection targets all three critical parts of a floating-point number: The sign bit, the mantissa bits and the exponent bits. Within the mantissa and the exponent, the position of the bit flip is chosen randomly. In case of the injection of a multi-bit flip, a position is determined randomly for two bits and the remaining bits are flipped randomly between these two bits. This allows to create multi-bit flips with certain neighborhood characteristics. The bit flips are generated using bitwise XOR operations. An error vector $errorVec$ is determined with the desired number of faulty bits. This error vector is used as mask for the bitwise XOR with the data word $dataVec$.

$$\begin{aligned} dataVec &= 01111\ldots01011000 \\ \oplus errorVec &= 01000\ldots00000001 \\ result &= 00111\ldots01011001 \end{aligned}$$

Algorithm 3 shows the performed steps in detail.

The following parameters are passed to the GPU kernel for the fault injection:

- The processor-ID of the streaming multiprocessor.
- The fault type, which determines if an addition or multiplication instruction is the target.
- The module-ID to determine which of the $RX \cdot RY$ adders or multipliers shall be affected.
- The error vector $errorVec$ as a bit mask.
- The parameter $kInjection$, which determines the point in time during the execution when the fault injection takes place.

For the evaluation of the error detection capabilities, the proposed *A-ABFT* scheme is compared against the simplified

Algorithm 3: Kernel for matrix multiplication with fault injection infrastructure.

Input: matrix A, matrix B, parameters for fault injection
Output: matrix C

Data: local registers $accum[RX \cdot RY]$, $rA[RX]$, $rB[RY]$
Data: shared memory $smA[BK][BM]$, $smB[BK][BN]$
Data: local registers $errorVecMul[RX][RY]$, $errorVecAdd1[RX][RY]$, $errorVecAdd2[RX][RY]$

$accum[0..RX][0..RY] \leftarrow 0$;

load one $BM \cdot BK$ block of A into $smA[BK][BM]$;
load one $BK \cdot BN$ block of B into $smB[BK][BN]$;
sync

while $K > 0$ **do**
  $K = K - 1$;
  for $ki \leftarrow 1$ **to** BK **do**
    /* initialize local bit masks for inner loop error injection */
    init $errorVecMult[0..RX][0..RY]$;
    init $errorVecAdd1[0..RX][0..RY]$;

    load one column of A in smA into $rA[0..RX]$;
    load one row of B in smB into $rB[0..RY]$;

    /* accumulation with inner loop error injection */
    $accum[0..RX][0..RY] += ((rA[0..RX] * rB[0..RY]) \oplus errorVecMult[0..RX][0..RY])$;
    $accum[0..RX][0..RY] = (accum[0..RX][0..RY] \oplus errorVecAdd1[0..RX][0..RY])$;
  end

  load one $BM \cdot BK$ block of A into $smA[BK][BM]$;
  load one $BK \cdot BN$ block of B into $smB[BK][BN]$;
  sync
end

/* error injection for merging results */
init $errorVecAdd2[0..RX][0..RY]$;
Merge $accum[0..RX][0..RY]$ with $BM * BN$ block of C;
$BM * BN$ block of $C = BM * BN$ block of $C \oplus errorVecAdd2[0..RX][0..RY]$;

error analysis approach (SEA-ABFT) from [28]. A single fault is injected per matrix multiplication. The injections target the two additions and the multiplications. Separate experiments are performed for single-bit-flips and multi-bit-flips with 3 and 5 flipped bits, respectively. Since the experimental evaluation for 1, 3 and 5 flipped bits did not show significant differences in the behavior of *A-ABFT* and SEA-ABFT – the trend in the results was consistent across all experiments – only the results for single-bit-flips are presented in the following. The experiments are performed for matrix dimensions from 512×512 to 8192×8192 and for homogenous input value ranges from -1.0 to 1.0, -100.0 to 100.0 and for the matrices with high value range dynamic and parameters $\alpha = 0$ and $\kappa = 65536$. To set the baseline for the error classification, the expectation value and the variance of the occurring rounding error for the affected matrix element $c_{i,j}$ of the result matrix C are determined (*inevitable rounding error*). An error is classified as significant, if the absolute error of the affected matrix element is smaller than 3σ of the probabilistically determined rounding error (*intolerable critical compute error*). Errors in the magnitude of the expectation value of the rounding error are neglected (*tolerable compute errors*).

The performed experiments showed that *A-ABFT*, as well as SEA-ABFT detected all faults that have been injected into the sign bit or the exponent. Figure 4 presents the percentage of detected errors for injections of single-bit-flips into the mantissa bits over the input value ranges -1.0 to 1.0, -100.0 to 100.0 and for the matrices with high value range dynamic. The new *A-ABFT* scheme for matrix multiplications on GPUs achieves a significantly better error detection throughout all combinations of affected operation, input values, matrix dimensions and number of flipped bits. In many cases, the percentage of detected errors is well over 90%. This applies also for the not depicted results of multi-bit-flips with 3 and 5 flips. The weaker rounding error bounds that are determined by the SEA-ABFT scheme lead to significantly worse error detection rates. The figure also shows that the error detection capability of *A-ABFT* does not depend on the size of the input matrices, a drawback that becomes evident for the SEA-ABFT approach, where the percentage of detected errors tends to decrease for increasing matrix dimensions.

VII. Conclusion

In this work, *A-ABFT* for matrix multiplications on graphics processing units has been introduced. For the first time, this new, parallel ABFT scheme autonomously determines rounding error bounds for the ABFT runtime error classification without requiring any calibration runs or manually set error bounds by the user. The determined rounding error bounds are up to two orders of magnitude closer to the actual rounding error, compared to other state of the art approaches. This increases the error detection capabilities of *A-ABFT* significantly and leads to error detection rates of well over 90%. Moreover, the error detection capability of *A-ABFT* is not influenced by the size of the processed matrices. *A-ABFT* integrates the computation of the rounding error bounds tightly into the encoding and checking procedures, which leads to low performance overheads and peak double-precision floating-point performance values of over 900 GFLOPS. Therefore, *A-ABFT* is an important step towards the goal of *dependable scientific computing on GPUs*.

VIII. Acknowledgment

The authors would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart.

References

[1] E. Winsberg, *Science in the Age of Computer Simulation*. University of Chicago Press, 2010.

[2] J. Nickolls and W. Dally, "The GPU Computing Era", *IEEE Micro*, vol. 30, no. 2, pp. 56–69, march-april 2010.

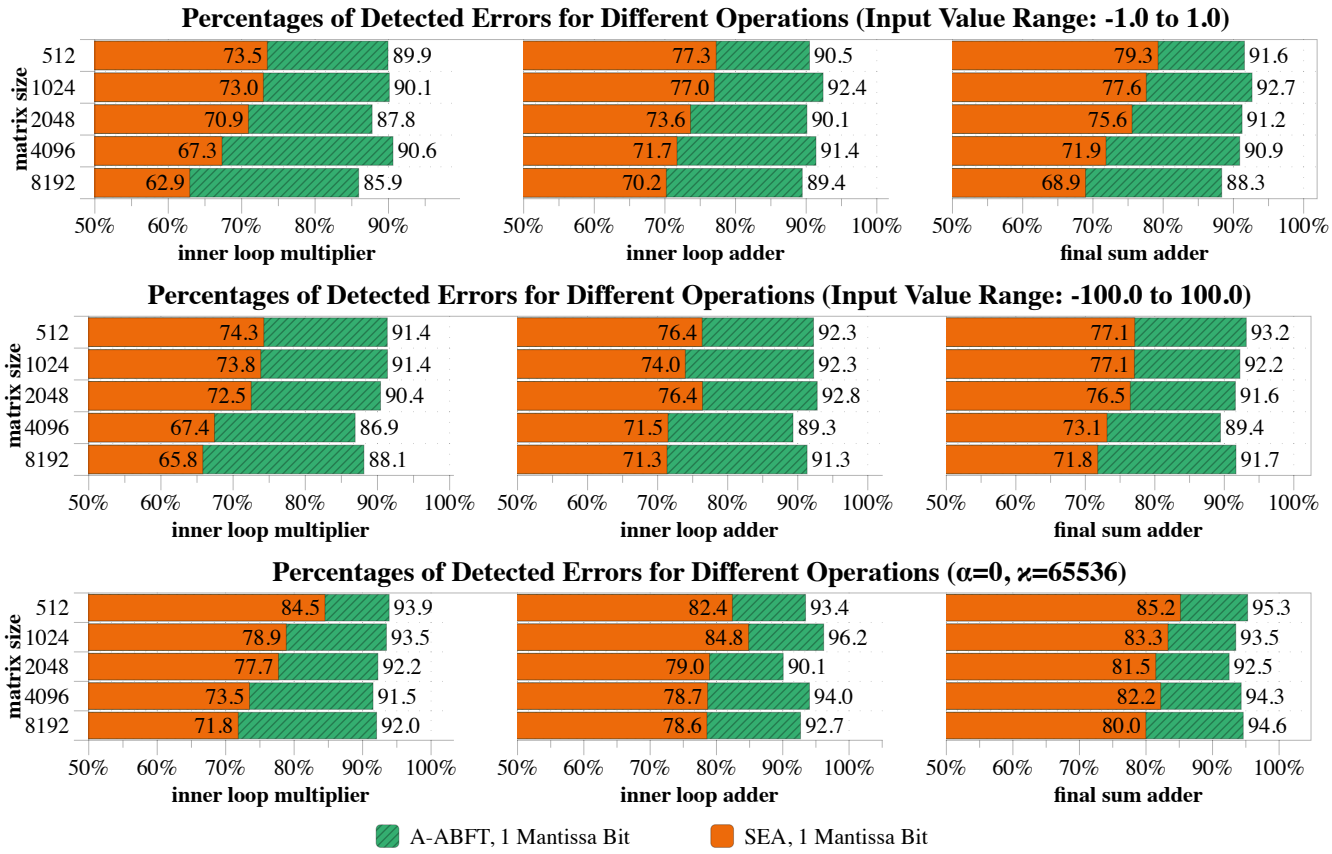

Figure 4. Comparison of detected errors between A-ABFT and SEA-ABFT.

[3] M. A. Kochte *et al.*, "Efficient Fault Simulation on Many-core Processors", in *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC'10)*. ACM, 2010, pp. 380–385.

[4] S. Holst, E. Schneider, and H.-J. Wunderlich, "Scan Test Power Simulation on GPGPUs", in *Proceedings of the 21st IEEE Asian Test Symposium (ATS12)*. IEEE Computer Society, 2012, pp. 155–160.

[5] C. Braun *et al.*, "Parallel Simulation of Apoptotic Receptor-Clustering on GPGPU Many-Core Architectures", in *IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 2012*, oct. 2012, pp. 1–6.

[6] C. Braun *et al.*, "Acceleration of Monte-Carlo Molecular Simulations on Hybrid Computing Architectures", in *IEEE 30th International Conference on Computer Design (ICCD), 2012*, 30 2012-oct. 3 2012, pp. 207–212.

[7] I. Haque and V. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU", in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, 2010, pp. 691–696.

[8] "The International Technology Roadmap for Semiconductors 2012 Edition", http://www.itrs.net/Links/2012ITRS/Home2012.htm. [Online]. Available: http://www.itrs.net/Links/2012ITRS/Home2012.htm

[9] F. Cappello, "Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities", *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009. [Online]. Available: http://hpc.sagepub.com/content/23/3/212.abstract

[10] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518 –528, June 1984.

[11] J.-Y. Jou and J. A. Abraham, "Fault-Tolerant Matrix Operations on Multiple Processor Systems using Weighted Checksums", in *SPIE Proc. 28th Annual Technical Symposium*. International Society for Optics and Photonics, 1984, pp. 94–101.

[12] A. Reddy and P. Banerjee, "Algorithm-Based Fault Detection for Signal Processing Applications", *IEEE*

Transactions on Computers, vol. 39, no. 10, pp. 1304–1308, 1990.

[13] C.-Y. Chen and J. A. Abraham, "Fault-Tolerant Systems for the Computation of Eigenvalues and Singular Values", in *SPIE Proc. 30th Annual Technical Symposium*. International Society for Optics and Photonics, 1986, pp. 228–237.

[14] C. Braun and H. Wunderlich, "Algorithm-Based Fault Tolerance for Many-Core Architectures", in *15th IEEE European Test Symposium (ETS'10)*, 2010, pp. 253–253.

[15] C. Ding *et al.*, "Matrix Multiplication on GPUs with On-Line Fault Tolerance", in *IEEE 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA'11)*, 2011, pp. 311–317.

[16] P. Rech *et al.*, "An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs", *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2797–2804, 2013.

[17] H.-J. Wunderlich, C. Braun, and S. Halder, "Efficacy and Efficiency of Algorithm-Based Fault-Tolerance on GPUs", in *IEEE 19th International On-Line Testing Symposium (IOLTS'13)*, 2013, pp. 240–243.

[18] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra", in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 31:1–31:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413402

[19] G. Tan *et al.*, "Fast Implementation of DGEMM on Fermi GPU", in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 35:1–35:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063431

[20] J. Rexford and N. Jha, "Partitioned Encoding Schemes for Algorithm-Based Fault Tolerance in Massively Parallel Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 6, pp. 649–653, 1994.

[21] P. Banerjee *et al.*, "Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor", *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1132–1145, 1990.

[22] V. Balasubramanian, "The Analysis and Synthesis of Efficient Algorithm-Based Error-Detection Schemes for Hypercube Multiprocessors", Illinois Univ., Urbana, IL (United States), Tech. Rep., 1991.

[23] F. Assad and S. Dutt, "More Robust Tests in Algorithm-Based Fault-Tolerant Matrix Multiplication", in *Digest of Papers Twenty-Second International Symposium on Fault-Tolerant Computing, 1992. (FTCS-22)*. IEEE, 1992, pp. 430–439.

[24] S. Dutt and F. Assaad, "Mantissa-Preserving Operations and Robust Algorithm Based Fault Tolerance for Matrix Computations", *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 408–424, 1996.

[25] N. J. Higham, *Accuracy and Stability of Numberical Algorithms*. Siam, 1996, no. 48.

[26] G. H. Golub and C. F. Van Loan, *Matrix Computations*. JHU Press, 2012, vol. 3.

[27] M. Turmon *et al.*, "Tests and Tolerances for High-Performance Software-Implemehted Fault Detection", *IEEE Transactions on Computers*, vol. 52, no. 5, pp. 579–591, 2003.

[28] A. Roy-Chowdhury and P. Banerjee, "Tolerance Determination for Algorithm-Based Checks using Simplified Error Analysis Techniques", in *Digest of Papers., The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993. (FTCS-23)*. IEEE, 1993, pp. 290–298.

[29] J. L. Barlow and E. Bareiss, "On Roundoff Error Distributions in Floating Point and Logarithmic Arithmetic", *Computing*, vol. 34, no. 4, pp. 325–347, 1985.

[30] F. Benford, "The Law of Anomalous Numbers", *Proceedings of the American Philosophical Society*, vol. 78, no. 4, pp. pp. 551–572, 1938. [Online]. Available: http://www.jstor.org/stable/984802

[31] R. W. Hamming, "On the Distribution of Numbers", *Bell System Technical Journal*, vol. 49, no. 8, pp. 1609–1625, 1970.

[32] R. S. Pinkham, "On the Distribution of First Significant Digits", *The Annals of Mathematical Statistics*, vol. 32, no. 4, pp. 1223–1230, 1961.

[33] B. J. Flehinger, "On The Probability that a Random Integer has Initial Digit A", *The American Mathematical Monthly*, vol. 73, no. 10, pp. 1056–1061, 1966.

[34] D. E. Knuth, "The Art of Computer Programming, Volume 2: Seminumerical Algorithms", 1981.