Making Strassen Matrix Multiplication Safe

Himeshi De Silva, John L. Gustafson, Weng-Fai Wong
School of Computing, National University of Singapore
Singapore
Email: {himeshi, john, wongwf}@comp.nus.edu.sg

***Abstract*—Strassen's recursive algorithm for matrix-matrix multiplication has seen slow adoption in practical applications despite being asymptotically faster than the traditional algorithm. A primary cause for this is the comparatively weaker numerical stability of its results. Techniques that aim to improve the errors of Strassen stand the risk of losing any potential performance gain. Moreover, current methods of evaluating such techniques for safety are overly pessimistic or error prone and generally do not allow for quick and accurate comparisons.**

In this paper we present an efficient technique to obtain rigorous error bounds for floating point computations based on an implementation of unum arithmetic. Using it, we evaluate three techniques—exact dot product, fused multiply-add, and matrix quadrant rotation—that can potentially improve the numerical stability of Strassen's algorithm for practical use. We also propose a novel error-based heuristic rotation scheme for matrix quadrant rotation. Finally we apply techniques that improve numerical safety with low overhead to a LINPACK linear solver to demonstrate the usefulness of the Strassen algorithm in practice.

I. Introduction

Matrix-matrix multiplication is a heavily used operation in many scientific and mathematical applications. Strassen's recursive algorithm for matrix multiplication has long been known to be asymptotically faster than the traditional algorithm [1]; Figure 1 shows the higher performance of Strassen's algorithm versus traditional matrix multiplication obtained from a recent paper [2]. Cache effects cause a non-monotonic increase, but there is a clear advantage for matrices larger than about 1000-by-1000. The algorithm's ability to improve the performance using multiple levels of recursion can also be seen here. Strassen-type algorithms which minimize communication overhead in parallel implementations also exist [3]. Because Strassen's algorithm tiles the matrix into submatrices, it is naturally amenable to the same sort of cache-level optimizations that are currently used to improve traditional matrix multiplication such as those used in LAPACK. Given that the solution of the linear system run by the TOP500 benchmark supercomputers uses a matrix-matrix multiplication kernel in which matrix tiles can be tens of thousands of elements on a side, significant performance gains are possible with Strassen's algorithm [4].

Despite the obvious advantages of Strassen's algorithm, it is not widely used by the scientific community. Most Level 3 Basic Linear Algebra Subprogram (BLAS) libraries, for example, do not use it. A primary reason for this is the weaker numerical stability of Strassen's algorithm as it has been implemented to date. Figure 2 shows this with probability density of the worst-case decimal accuracy. Improving Strassen's numerical stability to levels comparable to that of the traditional algorithm will improve the likelihood of Strassen's algorithm being used in practical applications that will benefit from its performance gains.

Numerical instability is caused by *rounding* of the exact result of a computation to a floating point number. Rounding after every operation as the calculation progresses compounds such errors and can result in grave loss of accuracy. In this work, we apply techniques—exact dot product, fused multiply-add, and matrix quadrant rotation—that can alleviate rounding errors in Strassen's algorithm, and compare their numerical stability improvements against that of the traditional algorithm's, using unum arithmetic. Based on our evaluation we then apply the techniques with the most promising performance gain and numerical stability to the LINPACK benchmark [4] and measure the residuals to compare their usefulness in practical applications.

Evaluating different matrix multiplication algorithms based on their numerical stability can be performed by obtaining bounds to the absolute error or by statistical averaging the error of results of pseudo-random floating point inputs [5]. Deriving theoretical bounds requires careful mathematical analysis and detailed understanding of floating point arithmetic. To make the bounds tractable expressions, such

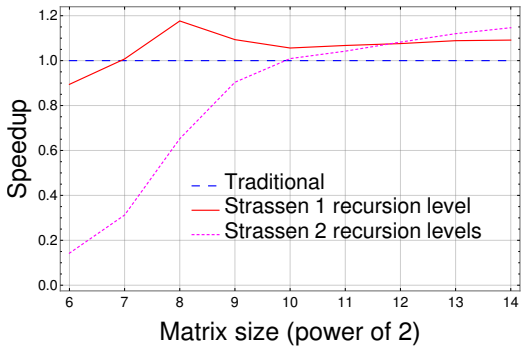


Figure 1. Traditional vs. Strassen square matrix-matrix multiplication performance on an Intel® Xeon® E5-2680 processor. (The work only supports Sandy Bridge and Ivy Bridge microarchitectures at present.)

deriviations make simplifying assumptions that overestimate the actual errors incurred. The statistical averaging of actual errors also has a drawback; we will show in section II-D how even simple inputs can produce surprisingly erroneous results due to cancellation. Instead, we use unum arithmetic, which implements variable precision interval arithmetic to obtain rigorous floating point error bounds on the results and thereby bound the accuracy loss. Unums provides a practical, fast and highly flexible mechanism for gauging numerical stability. Unlike other interval libraries, unums support exact dot products, which we will show is a key technique for making Strassen's algorithm numerically safe.

Our contributions in this paper are summarized as follows:

- Introduce unum arithmetic as a practical, quick, and rigorous method for measuring floating point error. We adapt existing definitions of error to unums and demonstrate how to use them to compare the numerical stability of algorithms. We also provide a rigorously tested software library that implements unums.
- Apply three techniques that reduce the errors of Strassen's algorithm without decreasing performance, and compare them for numerical stability.
- Propose a heuristic algorithm which will output a rotation scheme that improves the numerical stability for Strassen's algorithm when there is more than one level of recursion.
- Demonstrate the effectiveness of selected techniques with Strassen's algorithm applied to a practical application: a LINPACK linear equation solver.

The remainder of the paper is organized as follows. Section II presents the background for our work. Section III presents unum arithmetic, our implementation, and new error and accuracy definitions for unums. In Section IV, we apply the aforementioned techniques to Strassen's algorithm and compare their numerical stability to that of the traditional algorithm, using unums. Section V applies selected techniques to a LINPACK linear equation solver to demonstrate the use of Strassen's algorithm in practice. Section VI discusses related work and Section VII concludes.

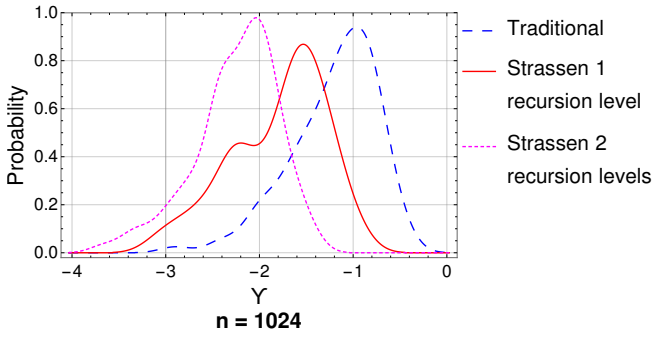


Figure 2. Histograms of worst-case accuracy (see section III-B for Υ definition) of traditional vs. Strassen algorithms for a 1024-by-1024 matrix

II. Background

A. Matrix Multiplication

Multiplying two matrices $\mathbf{A}$ and $\mathbf{B}$ to obtain a result matrix $\mathbf{C}$ can be done in many ways. For the purpose of this work we assume that $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are $n \times n$ matrices and n is evenly divisible by 2^r where r is the number of recursions, described below. Non-square matrices can be accommodated by tiling with square submatrices, or padding with zeros. If we denote the $(i,j)^{\text{th}}$ entry of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ as a_{ij}, b_{ij}, and c_{ij} respectively, then the classical algorithm for matrix multiplication can be written as Equation 1:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{1}$$

Each of the n^2 elements in the result matrix requires n multiplications and $n-1$ additions, giving this algorithm a time complexity of $O(n^3)$. Strassen's algorithm improves this time complexity by replacing some $O(n^3)$ matrix-matrix multiplications with matrix-matrix additions which are of $O(n^2)$ complexity, and by using recursion to obtain the product of submatrices. The first step in Strassen's algorithm is to partition matrices $\mathbf{A}$ and $\mathbf{B}$ into quadrants such that

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

While Strassen's original algorithm required 18 quadrant additions and 7 quadrant multiplications, for our work we use the Winograd variation which requires the minimum number of quadrant additions, i.e. 15. The following equations define Strassen-Winograd matrix multiplication.

$$\begin{aligned} s_1 &= a_{21} + a_{22} & s_5 &= b_{12} - b_{11} \\ s_2 &= s_1 - a_{11} & s_6 &= b_{22} - s_5 \\ s_3 &= a_{11} - a_{21} & s_7 &= b_{22} - b_{12} \\ s_4 &= a_{12} - s_2 & s_8 &= b_{21} - s_6 \end{aligned} \tag{2}$$

$$\begin{aligned} m_1 &= a_{11} \cdot b_{11} & m_5 &= s_3 \cdot s_7 \\ m_2 &= a_{12} \cdot b_{21} & m_6 &= s_4 \cdot b_{22} \\ m_3 &= s_1 \cdot s_5 & m_7 &= a_{22} \cdot s_8 \\ m_4 &= s_2 \cdot s_6 \end{aligned} \tag{3}$$

$$\begin{aligned} t_1 &= m_1 + m_4 & t_2 &= t_1 + m_5 \\ t_3 &= t_1 + m_3 \\ c_{11} &= m_1 + m_2 & c_{12} &= t_3 + m_6 \\ c_{21} &= t_2 + m_7 & c_{22} &= t_2 + m_3 \end{aligned} \tag{4}$$

The multiplications $m_1, ..., m_7$ are of submatrices computed by multiplying matrix quadrants of the input and these form the recursive steps of the algorithm by invoking the Strassen routine again. Thus, the algorithm has a time complexity of $O(n^{\log_2 7}) \approx O(n^{2.8})$. In practice, recursion is not applied all the way down to individual elements, since the extra additions increase execution time more than the

eliminated multiplications reduce. Instead, it is only applied r times, down to matrix sizes for which Strassen's algorithm is faster than the traditional algorithm. The size below which applying Strassen's algorithm is no longer beneficial is known as the *crossover point* (see Figure 1). Beyond the *crossover point*, traditional matrix multiplication is used to compute $m_1, ..., m_7$.

B. Error in Floating Point Computations

Typical floating point numbers contain a sign bit s, a k-bit value for the exponent e, and a p-bit value for the mantissa m. The exponent bias, b, is $2^{k-1} - 1$. The value of such a representation is

$$\hat{x} = \begin{cases} (-1)^s \times 2^{e-b} \times 1.m & \text{if } e \neq 0 \\ (-1)^s \times 2^{1-b} \times 0.m & \text{if } e = 0 \end{cases}. \tag{5}$$

The use of a fixed, finite number of bits to represent real quantities makes rounding error an unavoidable hazard of floating point arithmetic. The IEEE 754 standard for floating point arithmetic defines the rules for rounding [6]. There are several common ways to describe the error between a correct value x and its floating point approximation, $\hat{x}$. For example:

- *Absolute error*, Δ_a, is simply the absolute difference between the actual value and its floating point representation: $\Delta_a = |x - \hat{x}|$.
- *Relative error*, Δ_r, gives a measure of the error relative to the actual value of the answer: $\Delta_r = |\frac{x-\hat{x}}{x}|$.
- *ULP error* is a useful indicator of how many digits are accurate in a floating point number. "ULP" stands for "unit in the last place." If the exponent value from Equation 5 is z, then the ULP error is $\Delta_u = |\hat{x} - x| \times (2^{p-z})$.

C. Numerical Stability of Strassen's Algorithm

Two techniques have predominantly been used to measure numerical stability: theoretical derivation of upper and lower bounds to the error, and empirically calculating the absolute error of the result, obtained by running the algorithm with pseudo-random floating point inputs against the result computed at a higher precision. Both techniques indicate that Strassen's algorithm and its variations cannot achieve the same numerical stability as the traditional algorithm [7].

Derivations for numerical stability deem the traditional algorithm *component-wise numerically stable* since the rounding error in the output depends only on the errors in the corresponding row and column in the input. For n-by-n matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, the error in computing $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ is bounded by Inequality 6, where $\hat{\mathbf{C}}$ is the result matrix computed using floating point arithmetic, u is the unit roundoff error, and $|\cdot|$ denotes the absolute value of matrix elements:

$$|\mathbf{C} - \hat{\mathbf{C}}| \leq n \times u \times |\mathbf{A}| \times |\mathbf{B}| + O(u^2). \tag{6}$$

Any polynomial algorithm for multiplying $n \times n$ matrices that satisfies Inequality 6 must perform at least n^3 multiplications [7]. Because Strassen's algorithm performs fewer than n^3 multiplications, it cannot satisfy component-wise numerical stability. Instead, the error bound of the Strassen-Winograd algorithm satisfies Inequality 7 [8]. This is referred to as *normwise numerical stability*. Here n_o is the *crossover point* at which traditional matrix multiplication is used, and $\|\cdot\|$ is the matrix norm. While these theoretical error bounds are important, they are "loose" (overly pessimistic).

$$\|\mathbf{C} - \hat{\mathbf{C}}\| \leq \left[\left(\frac{n}{n_o}\right)^{\log_2 18}(n_o^2 + 6n_o) - 6n\right] \times u \times \|\mathbf{A}\| \times \|\mathbf{B}\| \; + O(u^2). \tag{7}$$

D. Hazards of Floating Point Arithmetic

Empirical measurements of numerical stability are subject to hazards such as cancellation that are inherent to floating point arithmetic [9]. Matrix-matrix multiplication is such a common task in scientific computing that most people assume it is numerically well-behaved. However, consider this situation: In multiplying 4-by-4 matrices $\mathbf{A}$ and $\mathbf{B}$, suppose one row of $\mathbf{A}$ is $\vec{a} = \left(3.2 \times 10^8, 1, -1, 8 \times 10^7\right)$ and one column of $\mathbf{B}$ is $\vec{b} = \left(4 \times 10^7, 1, -1, -1.6 \times 10^8\right)$. All the components of those two vectors are integers that are exactly expressible with IEEE single-precision floats. The matrix-matrix product requires that we find the dot product $\vec{a} \cdot \vec{b}$. A straightforward single-precision computation of $\vec{a} \cdot \vec{b}$ produces the result 0, indicating that the vectors are orthogonal. This is confirmed by computing it in IEEE double precision. Furthermore, it does not matter whether we do the double-precision sum left-to-right, right-to-left, or with a pairwise binary sum collapse. All three methods reassure us that the dot product is 0. However, the correct answer is **2**. When statistically averaging errors using pseudo-random inputs, these kinds of test cases can easily be missed, resulting in misleadingly optimistic guidance.

III. Bounding Errors with Unums

A. Unum Arithmetic and Its Implementation

As seen in section II-D, the empirical technique of evaluating numerical stability is not rigorous, whereas theoretical bounds are overly pessimistic. Unum arithmetic provides a number format and arithmetic operator rules that produce a real interval (closed, open, or half-open) in which the actual answer of the computation lies, instead of discarding accuracy information by rounding. While unums enjoy the benefits of traditional interval arithmetic such as being able to measure errors due to approximation and non-exact inputs, they are superior to interval arithmetic in that they support variable precision and perfect set operations (complement, union, and intersection) [10].

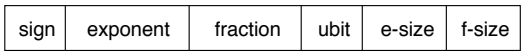


Figure 3. unum format

The unum number format is shown in Figure 3. Unum arithmetic operates in a predefined *environment* which is determined by two parameters, *esizesize* and *fsizesize*. The length of the *e-size* (*es*) and *f-size* (*fs*) fields are determined by these environment parameters. Their values in turn specify the length of the exponent and fraction fields. Figure 5 is an example of a unum in the environment $\{3,5\}$ (i.e. *esizesize* = 3 and *fsizesize* = 5). In this environment the *es* field is 3 bits long while *fs* is 5 bits long. The 3-bit *es* field can represent an *exponent* field of length 1 to 8 bits; similarly the 5-bit *fs* field can represent a *fraction* field of length 1 to 32 bits. The variable length *exponent* and *fraction* fields contain values with the smallest bit representation required to represent the tightest interval containing the value.

Changing the *es* and *fs* values allows for experimenting with different types of floats—single and double precision as well as many types not in the IEEE 754 Standard. The *ubit* is a single-bit field which when set indicates that the number is *inexact*. When a unum is inexact, the actual value lies in the open interval between two exact unums separated by one ULP. Instead of committing rounding error when faced with insufficient bits to represent a result, unums simply use the *ubit* to indicate that the answer is inexact and represent the interval that contains the result.

The value represented by an exact unum is given by Equation 8, similar to that for IEEE floats.

$$u = \begin{cases} (-1)^s \times 2^{e-(2^{es}-1)} \times (1 + \frac{f}{2^{fs+1}}) & \text{if } e \neq 0 \\ (-1)^s \times 2^{-2^{es}+2} \times \frac{f}{2^{fs+1}} & \text{if } e = 0 \end{cases} \quad (8)$$

If the value lies in an interval more than one ULP wide, it is represented with two unums, one each for the left and right bounds. This is known as a *ubound*. This provides a rich vocabulary for expressing open, half-open, and closed bounds with independent precision for each endpoint of the bound. Rules for operations of the form $(x \text{ op } y)$ where op $\in \{+, -, \times\}$, which are relevant to matrix multiplication, are given in Tables I and II, which indicate how open (parenthesis) and closed (square bracket) endpoints are handled. Because physical quantities in nature contains uncertainty, our analysis assumes inputs are intervals that are 1 ULP wide. Note: in the tables, directed rounding of $(x \text{ op } y)$ is used to assure containment of the exact answer.

We have implemented these unum routines as a C library with APIs similar to other multi-precision libraries, for ease of adoption. Figure 4 shows a sample code for the first product $(3.2 \times 10^8 \times 4 \times 10^7)$ of the dot product example given in section II-D. While 32-bit floats incorrectly produce $1.27999998334861312 \times 10^{16}$ as the exact answer for this multiplication, 32-bit unums produce a correct

```
#include <unum.h>

int main() {
    set_env(3,5);
    ubound_t a1, b1, product;

    ubound_init(&a1);
    ...
    x2ub(3.2E+8, &a1);
    x2ub(4E+7, &b1);
    timesubound(&product, a1, b1);
    ...
```

Figure 4. Unum C code for one multiplication

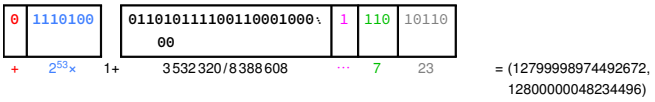


Figure 5. Example of a unum

bound indicating that the value lies in the open interval $(12799998974492672, 12800000048234496)$ (Figure 5).

B. Relative Decimal Accuracy

We can now propose a new measure of error for unums to compare different algorithms. The bound for the maximum absolute error of a computation is half the absolute width of the unum or ubound returned by it. If a computation returns the unum or ubound for the interval (x, y), then the maximum absolute error ϵ_{abs} in that range occurs at either x or y when the correct value is at the midpoint. Therefore we can write:

$$\begin{aligned} \epsilon_{abs} &= \left| \frac{x+y}{2} - y \right| \\ &= \frac{|x-y|}{2} \end{aligned} \quad (9)$$

Even as the computation progresses to use a ubound to represent the interval as its width grows beyond 1 ULP, unum arithmetic still envelopes the interval with the nearest exact or inexact unum values, thus maintaining Equation 9.

lower bound	$[-\infty$	$(-\infty$	$[y$	$(y$	$[\infty$
$[-\infty$	$[-\infty$	$[-\infty$	$[-\infty$	$[-\infty$	(NaN
$(-\infty$	$[-\infty$	$(-\infty$	$(-\infty$	$(-\infty$	$[\infty$
$[x$	$[-\infty$	$(-\infty$	$[x \pm y$	$(x \pm y$	$[\infty$
$(x$	$[-\infty$	$(-\infty$	$(x \pm y$	$(x \pm y$	$[\infty$
$[\infty$	(NaN	$[\infty$	$[\infty$	$[\infty$	$[\infty$
upper bound	$-\infty]$	$y)$	$y]$	$\infty)$	$\infty]$
$-\infty]$	$-\infty$	$-\infty]$	$-\infty]$	$-\infty]$	NaN)
$x)$	$-\infty]$	$x \pm y)$	$x \pm y)$	$\infty)$	$\infty]$
$x]$	$-\infty]$	$x \pm y)$	$x \pm y]$	$\infty)$	$\infty]$
$\infty)$	$-\infty]$	$\infty)$	$\infty)$	$\infty)$	$\infty]$
$\infty]$	NaN)	$\infty]$	$\infty]$	$\infty]$	$\infty]$

Table I
RULES FOR UNUM ADDITION AND SUBTRACTION

lower bound	$[0$	$(0$	$[y$	$(y$	$[\infty$
$[0$	$[0$	$[0$	$[0$	$[0$	(NaN
$(0$	$[0$	$(0$	$(0$	$(0$	$[\infty$
$[x$	$[0$	$(0$	$[x.y$	$(x.y$	$[\infty$
$(x$	$[0$	$(0$	$(x.y$	$(x.y$	$[\infty$
$[\infty$	(NaN	$[\infty$	$[\infty$	$[\infty$	$[\infty$
upper bound	$0]$	$y)$	$y]$	$\infty)$	$\infty]$
$0]$	$0]$	$0]$	$0]$	$0]$	NaN)
$x)$	$0]$	$x.y)$	$x.y)$	$\infty)$	$\infty]$
$x]$	$0]$	$x.y)$	$x.y]$	$\infty)$	$\infty]$
$\infty)$	$0]$	$\infty)$	$\infty)$	$\infty)$	$\infty]$
$\infty]$	NaN)	$\infty]$	$\infty]$	$\infty]$	$\infty]$

Table II
RULES FOR UNUM MULTIPLICATION

	Float	LLNL Implementation	Our Implementation
Conversion (float to posit)	-	$6.53s \times 10^{-7}$s	1.19×10^{-8}s
Addition	1.99×10^{-10}s	2.03×10^{-5}s	3.92×10^{-7}s
Multiplication	2.18×10^{-10}s	5.58×10^{-6}s	4.72×10^{-6}s

Table III
SPEED OF OPERATIONS OF FLOATS AND UNUMS IN {*3,5*}

For the unum or ubound u representing the interval (x, y), we define the relative error, ϵ_{rel} as,

$$\begin{aligned} \epsilon_{rel} &= \frac{\frac{|x-y|}{2}}{\frac{|x+y|}{2}} \\ &= \left|\frac{x-y}{x+y}\right| \end{aligned} \tag{10}$$

Relative error provides a more relevant comparison since it provides context of the magnitude of the correct value. Single ULP relative error for floating point values "wobbles" as the fraction ranges from 1 to 2; for unums, single ULP relative error also changes due to variable fraction length.

Relative accuracy, α, is the inverse of relative error and thus can be written as:

$$\alpha = \frac{1}{\epsilon_{rel}} \tag{11}$$

Let the *relative decimal accuracy* of a computed value that lies in the interval (x, y) be Υ. Then

$$\begin{aligned} \Upsilon(x, y) &= \log_{10}(\alpha) \\ &= \log_{10}\left|\frac{x+y}{x-y}\right| \end{aligned} \tag{12}$$

For example, if x and y agree to 6 decimal places, then $\Upsilon(x, y)$ will be approximately 6. If x and y differ in sign, $\Upsilon(x, y)$ will be negative.

C. Unums vs. Other Interval Arithmetic Implementations

Some interval arithmetic implementations that are available are libieee1788, JInterval and MPFI [11]–[13]. However, only unum arithmetic offers support for the exact fused dot product which we have employed in our work. Moreover these libraries either rely on the support of multi-precision libraries such as MPFR [14] (making them order of magnitudes slower, thus rendering them unusable for practical use) or use native floating point arithmetic which is susceptible to rounding errors. Unums also support open intervals which ensure the safe execution of the computation as well as a more complete representation of the possible intervals on the number line. By changing the environment and maximum exponent and fraction sizes, unums also allows experimentation with many floating point formats other than IEEE 754 single and double precision.

Apart from these interval arithmetic libraries, there are several other implementations of unum arithmetic. However, for reasons such as being too slow, not flexible enough for our experiments, or not having being tested enough, we chose to develop and thoroughly test our own flexible unum library in C. Table III gives a comparison of speed of operations between floats, our unum implementation, and another unum library developed at the Lawrence Livermore National Laboratory (LLNL) in C [15]. While the LLNL library allows for a wider range of precisions, it comes at the cost of reduced speed as shown by the timings (measured on the same hardware as in Section IV-E), thus requiring us to come up with our own faster implementation.

IV. EVALUATION

Using unums, we evaluate the numerical stability of each of the three techniques—exact dot product, fused multiply add and matrix quadrant rotations—when applied to Strassen's algorithm against the traditional algorithm. For a given matrix size n and recursion level r, we repeatedly execute both algorithms (with Strassen's combined with the proposed technique) with the same inputs to measure their minimum Υ value (worst decimal accuracy). We then plot the probability density function of the results (based on a smooth kernel density estimate) to observe each technique with respect to their worst Υ value. To mimic IEEE floats with unums, we limit the maximum exponent and fraction sizes (*es* and *fs*) of the unum environment to those of IEEE single precision floats. Note that because the traditional algorithm uses an input element only once in the calculation of an output element and thus does not suffer from the *dependency issue* of interval arithmetic [10], the bound on the worst-case floating point error will be tight.

Previous statistical studies of the numerical stability of Strassen's algorithm have used pseudo-random floats selected from the range $(0, 1)$ or $(-1, 1)$ as proxies for the set of all possible floats [16]. Since we wish to capture the effects of adding numbers of different sign, we use $(-1, 1)$ as our proxy range with the numbers uniformly

distributed. This range includes numbers spanning many orders of magnitude, as floats are designed to do. A more informative sampling requires application-specific knowledge of the probability distribution of values.

A. The Exact Dot Product

Given two lists of n floating point numbers $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_n)$, the Exact Dot Product (EDP) is computed by Equation 13 with rounding taking place only after the entire expression has been computed exactly in a high-precision fixed-point accumulator.

$$\text{EDP}(A, B) = \textit{round}(a_1 \cdot b_1 + a_2 \cdot b_2 + ... + a_n \cdot b_n) \quad (13)$$

Kulisch observed that with a "perfect accumulator" of a few hundred bits, the dot product of float vectors can be calculated exactly [17]. Because rounding only happens when converting the EDP to a float, error is at most 0.5 ULP. Recent hardware designs have shown that the EDP is a fast and practical feature to add to an arithmetic unit [18]. Moreover, support for the EDP is growing with 22 open-source and 5 proprietary projects underway or completed to create software or hardware for Type III unums (known as posits), which includes in its upcoming standard a requirement that the EDP be supported [19].

To apply the EDP to Strassen's algorithm, we implement it in the unum C library. Since the results are computed as bounds, we use an EDP accumulator for each endpoint. In the first step, the upper and lower bounds of the product of a_i and b_i are obtained using Table II rules. To add the computed bounds to the accumulators, the fraction is extended to its maximum size in that environment, shifted by its exponent biased with twice the absolute smallest possible exponent in that unum environment (since unums have variable bit lengths, these values change between environments). The shifted values are then added to the long accumulators according to rules in Table I. At the end of the operation, the accumulator results are converted back into a ubound. The length l of an accumulator in bits for this routine is given by Equation 14 where f_{size} is the maximum fraction size, e_{max} is the maximum exponent, e_{min} is the absolute value of the minimum exponent in the environment, and δ is a few additional bits to guard against overflow. δ can be precomputed depending on the $\log_2$ of the maximum number of terms that the EDP should handle.

$$l = 2(e_{\text{max}} + e_{\text{min}} + f_{\text{size}} + 1) + \delta. \quad (14)$$

1) Single recursion level: Figure 6 shows the behavior of the worst decimal accuracy of traditional vs. Strassen's algorithm combined with EDP for a single recursion and various n values. As n increases, Strassen's algorithm combined with the EDP has accuracy that surpasses that of the traditional algorithm and keeps improving.

Note: because submatrices are re-used in Strassen's algorithm, any interval-type bound *overestimates* error because of the "dependency problem." Therefore, the advantage of Strassen's algorithm with EDP is *understated* by these results.

2) Two recursion levels: As with the previous case, Figure 7 shows that the worst Υ of Strassen's with EDP is superior to the traditional case at around $n = 4096$ and keeps on improving, when two recursion levels are used.

B. Fused multiply-add

Given floating point values a, b and c, the fused multiply-add (FMA) is defined as

$$\textit{round}(a + (b \times c)). \quad (15)$$

That is, rounding takes place only after the addition. While it does not reduce the number of rounding operations as dramatically as the EDP, FMA has the advantage of widespread hardware support; the 2008 revision to the IEEE 754 Standard mandates FMA support. An FMA call is implemented in our unum C library, similarly to the EDP.

1) Single recursion level: Figure 8 illustrates the behavior of the worst Υ of Strassen's algorithm combined with FMA with one recursive level, versus the traditional algorithm. Strassen's algorithm does not improve its worst elements beyond that of the traditional algorithm and shows about 0.7 decimals less accuracy at the peak in each case of n.

2) Two recursion levels: When there are two recursion levels of Strassen's algorithm combined with FMA, it still lags behind the traditional algorithm in terms of the worst Υ value but shows some improvement for the larger n value.

C. Sub-matrix Quadrant Rotation

Operations in Strassen's algorithm are not evenly distributed among matrix elements; certain quadrants get compounded with error with each recursion level. However, by changing how the input separation is done in recursions, the error distribution among the quadrants in the result matrix can be modified [20]. The input matrices can be permuted

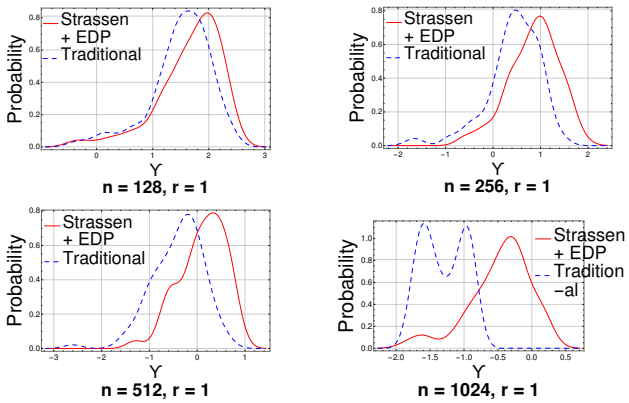


Figure 6. Strassen + EDP vs. Traditional, one recursion level

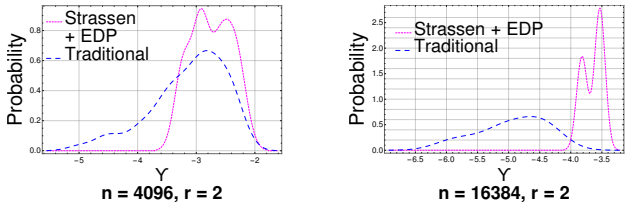


Figure 7. Strassen + EDP vs. Traditional, two recursion levels.

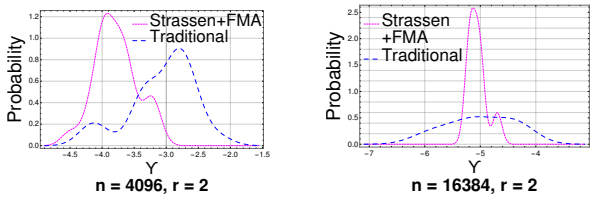


Figure 9. Strassen + FMA vs. Traditional, two recursion levels

using Equations 16 – 19 so that the results with the least or greatest error can be shifted to different quadrants.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \quad (16)$$

$$\begin{pmatrix} a_{21} & a_{22} \\ a_{11} & a_{12} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{21} & c_{22} \\ c_{11} & c_{12} \end{pmatrix} \quad (17)$$

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{12} & b_{11} \\ b_{22} & b_{21} \end{pmatrix} = \begin{pmatrix} c_{12} & c_{11} \\ c_{22} & c_{21} \end{pmatrix} \quad (18)$$

$$\begin{pmatrix} a_{21} & a_{22} \\ a_{11} & a_{12} \end{pmatrix} \times \begin{pmatrix} b_{12} & b_{11} \\ b_{22} & b_{21} \end{pmatrix} = \begin{pmatrix} c_{22} & c_{21} \\ c_{12} & c_{11} \end{pmatrix} \quad (19)$$

When applying Strassen's algorithm at only one level, rotating the input quadrants will have no effect on the overall numerical stability of the result since it will only shift the four quadrants along with their errors. However, if rotation is applied to the seven submatrix multiplications at the second recursive level, the error distributions of m_1,...,m_7 at the first recursive level can be modified, resulting in the change of the error distribution in the result.

D. Rotation Based on Heuristic Errors

The selection of which rotation to apply at each recursion level is important to the improvement of the numerical stability of the result. Previously suggested techniques are *random* and *round robin* selection [20]. Here we propose a new rotation scheme that is based on the distribution of error in the final result. Conventional wisdom dictates that the accuracy of a floating point calculation tends to decrease with the number of rounding operations. While this is not always the case, as Figure 10 demonstrates, the number of rounding operations each result element undergoes is a sufficient heuristic to indicate the error it will finally possess. The heatmap on the left shows the distribution of total rounding operations for each element of the matrix; the right heatmap shows the error distribution corresponding to each element (when Strassen is applied without any rotation).

To work out which permutation defined in Equations 16–19 should be applied at each node in the recursion tree, based on the accumulation of operations throughout the computation such that the numerical stability of the result is improved, let us first denote the matrix permutations in Equations 16 – 19 as $P = \{p_1, p_2, p_3, p_4\}$, respectively, and the number of levels of recursion (the height of the recursion tree not considering the leaf nodes) as h. We define the selected list of permutations as $R_{\text{final}} \leftarrow \{r_1, r_2, ..., r_i, ..., r_k\}$. Here, $k = \sum_{i=0}^{h-1} 7^i$; r_i denotes the selected permutation at the i^{th} node of the recursion tree which is numbered breadth-first, and $r_i \in P$. Since there are four possible permutations, the total number of possibilities for applying permutations to a given set of $m_1, ..., m_7$ is 4^7.

Algorithm 1 outputs Q_{final} for each set of $m_1, ..., m_7$ which is rearranged to obtain R_{final}. Routines MATMUL-TRADITIONAL, MATRIXADD and MATRIXSUB all update the addition and multiplication counts of each matrix element depending on the particular algorithm. Since a test of

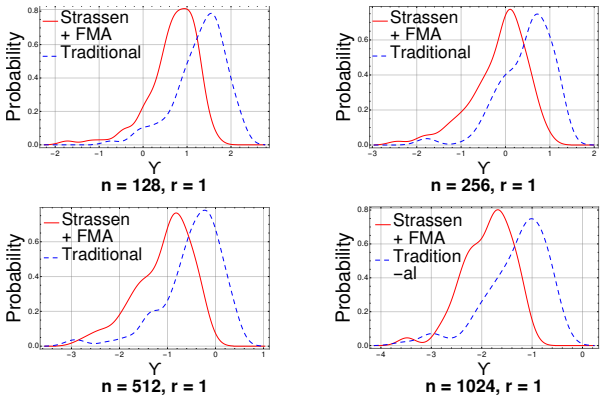
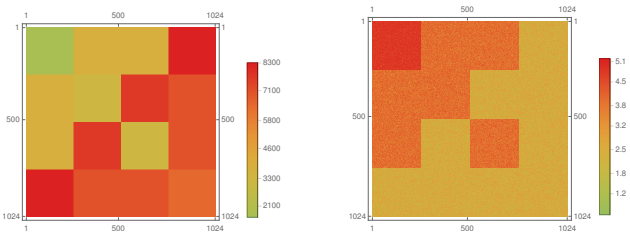


Figure 8. Strassen + FMA vs. Traditional, one recursion level



(a) Total rounding count (b) Υ

Figure 10. Inverse relation between total rounding count and accuracy ($n = 1024$, $r = 2$)

Algorithm 1 Algorithm for computing heuristic rotation

procedure ROTATE($\mathbf{A}, \mathbf{B}, size$) ▷ input $\mathbf{A}$,$\mathbf{B}$, and size
  if $size = \frac{n}{2^h}$ **then** ▷ Crossover point
    $\mathbf{C}$ = MATMULTRADITIONAL($\mathbf{A}, \mathbf{B}$, size)
    return $\mathbf{C}$
  $\{a_{11}, a_{12}, a_{21}, a_{22}\} \leftarrow$ PARTITION($\mathbf{A}$)
  $\{b_{11}, b_{12}, b_{21}, b_{22}\} \leftarrow$ PARTITION($\mathbf{B}$)
  $s_1 =$ MATRIXADD(a_{21}, a_{22}) ▷ Compute $s_1, ..., s_8$
  ⋮ (see Equation 2)
  $s_8 =$ MATRIXSUB(b_{21}, s_6)
  $size \leftarrow size/2$
  $m_1 =$ ROTATE$(a_{11}, b_{11}, size)$
  ⋮ (see Equation 3)
  $m_7 =$ ROTATE$(a_{22}, s_8, size)$
  Let $Q_i \leftarrow \{q_1, ..., q_7\}; q_{1,..,7} \in P$, be a list of permutations that can be applied to a set of $m_1, ..., m_7$
  $L \leftarrow \{Q_1, Q_2, ..., Q_{47}\}$ ▷ Generate all Q_i
  for each $Q_i \in L$ **do**
    $\mathbf{C}_{\text{temp}} \leftarrow$ COMBINE$(m_1, ..., m_7, Q_i)$
    if ISMORESTABLE($\mathbf{C}_{\text{temp}}$, $\mathbf{C}$) **then**
      $\mathbf{C} \leftarrow \mathbf{C}_{\text{temp}}$
      $Q_{\text{final}} \leftarrow Q$
  PRINT(Q_{final}) ▷ Output the optimal combination
return $\mathbf{C}$

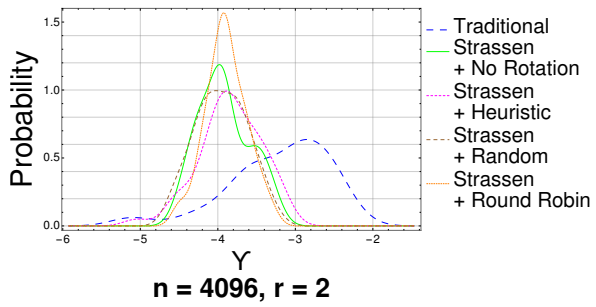


Figure 11. Strassen + Rotation vs. Traditional, Two Recursion Levels

multiplication and addition of random (positive and negative) values shows that multiplications deteriorate ϒ more than addition, the two operations will have different weights in accumulation. The COMBINE routine executes Equations 4. ISMORESTABLE sorts the total operation counts of two matrices, and selects the one that minimizes the maximum operation count per result element.

1) Two recursion levels: Figure 11 compares Strassen with several rotation techniques—random, round robin, heuristic—against the traditional algorithm. Here the heuristic rotation produces a small improvement of ϒ, however there is no significant difference between the rotation schemes. Moreover, Strassen with rotation alone cannot achieve the numerical stability of the traditional algorithm.

E. Discussion

Apart from numerical stability, speed and overhead also affect the selection of an algorithm for practical use. Here we model the performance of the proposed techniques when applied to traditional and Strassen's matrix multiplication algorithms. Recent work has demonstrated that the EDP can be computed with one cycle per element pair of the input vector, a performance matching that of Intel's MKL routine for a dot product [18]. This means that applying EDP to Strassen's algorithm and replacing a dot product routine leaves its performance gains intact without incurring additional overhead. Rotation simply involves changing a pointer, and therefore adds no overhead. While applying EDP to the traditional algorithm will achieve the best numerical stability, Strassen's with EDP will still produce favorable performance. (The Kahan summation technique was not considered for improving numerical stability due to its tripling of addition operation count, which negates any performance gain [21].)

Table IV compares numerical stability, operation counts, and timing (on Intel Xeon CPU E5-2650 v4 @ 2.20 GHz, 32K L1d, 32k L1i, 256K L2, 30720K L3 and using icc 18.0.3 20180410) for an $n \times n$ float matrix multiplication when n is 4096 and 16384, for the different algorithms and techniques proposed. Here, Strassen's algorithm uses $r = 2$. Although applying the EDP to the traditional algorithm will yield the best numerical stability, as seen by the performance numbers, it is *slower* than when the EDP is applied to Strassen's algorithm. Strassen's with the EDP is also the fastest among all the combinations of algorithms/techniques. The traditional algorithm performs the worst. Strassen with rotation has similar performance to Strassen without, confirming that rotation adds no overhead.

Because the performance of the EDP is similar to that of the Intel MKL dot product, we use it to simulate the performance of the EDP. However the dot product call is not typically used for matrix multiplication and instead the optimized level 3 BLAS ?GEMM function for matrix multiplication is used (which uses the traditional algorithm). Using this call with the traditional algorithm takes 1.63 and 105.74 seconds for 4096 and 16384 matrix sizes. Using the same call with Strassen yields a timing of 1.81 and 91.81 seconds for the same sizes. As shown in Figure 1 and demonstrated [2], this level 3 BLAS call can be further optimized for an even better performance for Strassen, surpassing that of the traditional algorithm even for smaller matrix sizes.

Interval arithmetic bounds become loose when the same interval appears multiple times in the calculation. While the traditional algorithm is not subjected to this, Strassen's algorithm is. Unum arithmetic provides a solution to this by optionally subdividing an interval into subintervals that tile each interval perfectly, which reduces the dependency effect

Method	Stab. Rank	Operations	Time(s)	
			4096	16384
Trad.+EDP	0	n^2EDP	10.09	744.15
Strassen+EDP	1	$\frac{165n^2}{16}$add, $\frac{49n^2}{16}$EDP	6.73	498.92
Trad.	2	$n^3 - n^2$add, n^3mul	21.34	1628.57
Strassen+FMA	3	$\frac{165n^2}{16}$add, $\frac{49n^3}{64} - \frac{49n^2}{16}$FMA	7.79	1103.53
Strassen+Rot.	4	$\frac{49n^3}{64} + \frac{29n^2}{4}$add, $\frac{49n^3}{64}$mul	9.68	1090.45
Strassen	5	$\frac{49n^3}{64} + \frac{29n^2}{4}$add, $\frac{49n^3}{64}$mul	9.60	1083.70

Table IV
SPEED AND NUMERICAL STABILITY COMPARISON FOR STRASSEN'S WITH TECHNIQUES WHEN $r = 2$

at the cost of extra work. However, because the intervals were already sufficiently small to show the numerical safety of Strassen's algorithm, bound-tightening methods were not necessary in our work.

V. COMBINING EDP AND ROTATION

Given that the evaluations from the previous section indicate that the EDP improves the numerical stability and speed the most, and rotation improves numerical stability with no overhead, we combined EDP with rotation and applied it to the LINPACK benchmark to demonstrate its usefulness in practice [4]. We implemented the linear solver in C conforming to the ground rules specified for the LINPACK benchmark and had it use matrix multiplication with both the traditional algorithm and the combined technique. For each run of the algorithms, the following residual value was calculated [22]:

$$\frac{||Ax - b||_\infty}{\epsilon \times (||A||_\infty \times ||x||_\infty + ||b||_\infty) \times n} \tag{20}$$

We use the same default ϵ value ($\epsilon = 2^{-16}$), matrix norm calculation, and double precision floating point as in the HPL version of the LINPACK benchmark.

Figure 12 shows the smoothed histogram plot for 100 runs of the solver in which the dimension of A is $N = 8192$,

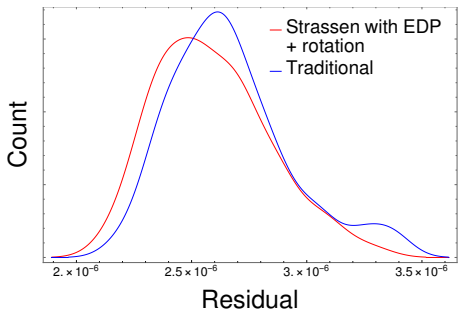


Figure 12. Linear solver residuals: Strassen+EDP+rotation vs. Traditional

using two levels of recursion of Strassen's algorithm. The HPL benchmark specifies a threshold value with which the residual value of a run will be compared against (default is 16). If the residual value is in the same order (order of 1) as the threshold, then the run is considered to have passed; all of our runs passed. Both algorithms show comparable residuals, confirming that Strassen's algorithm with the EDP can safely be used in practice. In fact, Figure 12 shows our modified Strassen to have *superior* numerical accuracy compared to traditional matrix multiplication.

VI. RELATED WORK

Conventional interval arithmetic has been proposed as a means to obtain the worst-case error bounds of floating point computations [23]. Many hardware implementations of interval arithmetic too have been developed [24]. Kahan has stated that what is needed is *variable-precision interval arithmetic*, which is exactly what unum arithmetic provides (in addition to other refinements) [25].

There have been many attempts at deriving theoretical error bounds for Strassen's algorithm [26], [27]. Our reference for numerical stability of both algorithms was based on Higham's textbook on stability of numerical methods [8]. While these bounds are important theoretically, they cannot consider practical aspects that are dependent on input variations. Strassen's algorithm has been computed with intervals using Rump's arithmetic to minimize the effect dependency and produce tighter intervals [28] without trying to improve its numerical stability.

Rotation as a technique for improving the numerical stability of Strassen's algorithm was first proposed by Castrapel and Gustafson [20]. They do not provide empirical evaluation of rotation and only suggest round-robin and random selection of rotation. Dumitrescu proposes the use of scaling when inputs are of widely varying magnitude to improve the error bounds of Strassen's algorithm [29]. Kaporin presents a fast matrix multiplication that has improved numerical stability for certain matrix dimensions [30]. The applicability of both techniques depend on the input. Ballard et. al suggest manipulating the fast matrix multiplication algorithm by applying different algorithms at different nodes of the recursion tree to improve numerical stability as well as diagonal input scaling [16]. This reduces the potential performance gain of applying only Strassen's algorithm.

VII. CONCLUSION

Obtaining rigorous error bounds of an algorithm is an important aspect of providing safety guarantees for the many applications that use floating point arithmetic. Unums provide a practical, quick and accurate way of obtaining this information without the need for complex analysis. The use of unums to measure the the accuracy of Strassen's algorithm demonstrate this.

The combination of the EDP and Strassen's algorithm shows that we can achieve both higher speed *and* better accuracy at the same time. Moreover, with efficient hardware accelerators for EDP this approach becomes even more desirable. For computing environments that have no access to the EDP, we proposed a heuristic rotation scheme that can be easily applied to improve the numerical stability with smaller margins of improvement. Our work enables the safe use of Strassen's algorithm in practical applications.

REFERENCES

[1] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[2] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, "Strassen's algorithm reloaded," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 59.

[3] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel strassen: Implementation and performance," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 101.

[4] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[5] D. Bini and G. Lotti, "Stability of fast algorithms for matrix multiplication," *Numerische Mathematik*, vol. 36, no. 1, pp. 63–72, 1980.

[6] IEEE, "IEEE standard for binary floating-point arithmetic," *IEEE Std. 754-2008*, 2008.

[7] W. Miller, "Computational complexity and numerical stability," *SIAM Journal on Computing*, vol. 4, no. 2, pp. 97–107, 1975.

[8] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[9] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[10] J. L. Gustafson, *The End of Error: Unum Computing*. CRC Press, 2015.

[11] M. Nehmeier, "libieeep1788: A c++ implementation of the ieee interval standard p1788," in *Norbert Wiener in the 21st Century (21CW), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–6.

[12] D. Y. Nadezhin and S. I. Zhilin, "Jinterval library: Principles, development, and perspectives." *Reliable Computing*, vol. 19, no. 3, pp. 229–247, 2013.

[13] "Multiple precision interval arithmetic library," https://gforge.inria.fr/frs/?group_id=157.

[14] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, p. 13, 2007.

[15] L. L. N. Laboratory, "Universal number library," https://github.com/LLNL/unum, accessed: 2018-06-18.

[16] G. Ballard, A. R. Benson, A. Druinsky, B. Lipshitz, and O. Schwartz, "Improving the numerical stability of fast matrix multiplication," *SIAM Journal on Matrix Analysis and Applications*, vol. 37, no. 4, pp. 1382–1418, 2016.

[17] U. W. Kulisch and W. L. Miranker, *Computer arithmetic in theory and practice*. Academic press, 2014.

[18] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic, "A hardware accelerator for computing an exact dot product," in *Computer Arithmetic (ARITH), 2017 IEEE 24th Symposium on*. IEEE, 2017, pp. 114–121.

[19] "Posithub," https://posithub.org/, accessed: 2018-06-18.

[20] R. R. Castrapel and J. L. Gustafson, "Precision improvement method for the strassen/winograd matrix multiplication method," Apr. 24 2007, uS Patent 7,209,939.

[21] W. Kahan, "Pracniques: further remarks on reducing truncation errors," *Communications of the ACM*, vol. 8, no. 1, p. 40, 1965.

[22] "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," http://www.netlib.org/benchmark/hpl/, 2017, [Online; accessed 14-September-2017].

[23] W. Kramer, "A priori worst case error bounds for floating-point computations," *IEEE transactions on computers*, vol. 47, no. 7, pp. 750–756, 1998.

[24] M. J. Schulte and E. E. Swartzlander, "A family of variable-precision interval arithmetic processors," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 387–397, 2000.

[25] "Transcription of The Great Debate: John Gustafson vs. William Kahan on Unum Arithmetic Held July 12, 2016 Moderated by Jim Demmel," http://www.johngustafson.net/pdfs/DebateTranscription.pdf, 2016, [Online; accessed 15-January-2018].

[26] R. P. Brent, "Algorithms for matrix multiplication," DTIC Document, Tech. Rep., 1970.

[27] J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg, "Fast matrix multiplication is stable," *Numerische Mathematik*, vol. 106, no. 2, pp. 199–224, 2007.

[28] M. Ceberio and V. Kreinovich, "Fast multiplication of interval matrices (interval version of strassen's algorithm)," *Reliable Computing*, vol. 10, no. 3, pp. 241–243, 2004.

[29] B. Dumitrescu, "Improving and estimating the accuracy of strassen's algorithm," *Numerische Mathematik*, vol. 79, no. 4, pp. 485–499, 1998.

[30] I. Kaporin, "The aggregation and cancellation techniques as a practical tool for faster matrix multiplication," *Theoretical Computer Science*, vol. 315, no. 2-3, pp. 469–510, 2004.