DEGREE PROJECT IN ,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, *SWEDEN 2020*

Comparing the precision in matrix multiplication between Posits and IEEE 754 floating-points

Assessing precision improvement with emerging floating-point formats

FREDRIK EKSTEDT KARPERS

SIMONE DE BLASIO

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Bachelor in Computer Science
Date: June 8, 2020
Supervisor: Stefano Markidis
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Swedish title: Jämförelse av precision i matrismultiplikation mellan Posits och IEEE 754 av flyttal

Abstract

IEEE 754 floating-points are the current standard way to represent real values in computers, but there are alternative formats emerging. One of these emerging formats are Posits. The main characteristic of Posit is that the format allows for higher precision than IEEE 754 floats of the same bit size for numbers of magnitude close to 1, but lower precision for numbers of much smaller or bigger magnitude.

This study compared the precision between IEEE 754 floating-point and Posit when it comes to matrix multiplication. Different sizes of matrices are compared, combined with different intervals which the values of the matrix elements were generated in.

The results showed that Posits outperformed IEEE 754 floating-point numbers in terms of precision when the values are in an interval equal to or larger than $[-0.01; 0.01)$, or equal to or smaller than $[-100; 100)$. Matrix size did not affect this much, unless the intermediate format Quire was used to eliminate rounding error.

For almost all other intervals, IEEE 754 floats performed better than Posits. Although most of our results favored IEEE 754 floats, Posits does have a precision benefit if one can be sure the data is within the ideal interval. Maybe Posits still have a role to play in the future of floating-point formats.

Sammanfattning

IEEE 754 flyttal är den nuvarande standarden för att representera reella tal i datorer, men det finns framväxande alternativa format. Ett av dessa nya format är Posit. Huvudkarakteristiken för Posit är att formatet möjliggör för högre precision än IEEE 754 flyttal med samma bitstorlek för värden av magnitud nära 1, men lägre precision för värden av mycket mindre eller större magnitud

Denna studie jämförde precisionen mellan flyttal av formaten IEEE 754 och Posit när det gäller matrismultiplikation. Olika storlekar av matriser jämfördes, samt olika intervall av värden som matriselementen genererades i.

Resultaten visade att Posits presterade bättre än IEEE 754 flyttal när det gäller precision när värdena är i ett intervall lika med eller större än $[-0.01; 0.01)$, eller lika med eller mindre än $[-100; 100)$. Matrisstorlek hade inte en anmärkningsvärd effekt på detta förutom när formtatet Quire användes för att eliminera avrundningsfel.

I nästan alla andra intervall presterade IEEE 754 flyttal bättre än Posits. Även om de flesta av våra resultat gynnade IEEE 754-flyttal, har Posits en precisions fördel om man kan vara säker på att värdena ligger inom det ideella intervallet. Posits kan alltså ha en roll att spela i framtiden för representation av flyttal.

Contents

Chapter 1

Introduction

Since the beginning of computer science, people have needed formats to represent real numbers in a way that can be processed by a computer. Many different implementations, such as fixed point and proprietary floating-point variants were constructed for various computer architectures. In 1985 the Institute of Electrical and Electronics Engineers introduced the IEEE 754 floating-point standard, which quickly became commonly implemented in most computers' floating-point units. These floating-point numbers are commonly used in matrix calculations for computer graphics, machine learning, big data and high performance computing purposes, just to name a few.

However, some criticism has been raised against the standard. A new real number standard called Posit has been proposed which attempts to improve on IEEE 754 floats in several ways, including exception handling and reduce wasted bit patterns[1]. But because floating-point calculations are hardware implemented, a change would require very compelling reasons to do so.

Another alleged benefit of the Posit standard is precision around the numbers most commonly used in real number arithmetic. This would reduce the effect of rounding errors for these numbers.

Since matrix calculations require several steps of both addition and multiplication of individual floating-point numbers, there is a real possibility these rounding errors can propagate and have a significant negative impact on the end result.

Some previous research has been carried out, but since Posit has only been around since 2017 there is a lot to discover. One study looked at how Posit and Float compete in single operations, such as: addition, multiplication and square root. The study focused on what results are representable and how precise the results were Another study looked at how at the accuracy is differ-

ent for Posit and Float in High Performance Computing (HPC) kernels. HPC kernels consists of larger amount of complex operations compared to the first study.

The earlier studies had either looked at simple operations or advanced combinations of operations. Therefore, there is a gap of medium level operations, such as matrix multiplication. Matrix multiplication is used a lot in HPC and in Deep learning.

The purpose of this study is to further explore how the different floating-point formats compare to and perform against each other, when used in matrix multiplication. Different number intervals and matrix sizes will be used. The goal is to contribute to the knowledge of number systems so that computers use the best formats available when representing numbers.

1.1 Problem statement

This study is a comparison of how the IEEE 754 floating-point and the Posit format compares when it comes to matrix multiplication. *What is the precision improvement when doing matrix multiplication using the Posit 32 bit format compared to IEEE single precision? How does this change as the magnitude of the numbers change?*

1.2 Scope

This study will only look at square matrices: matrices with the same amount of rows and columns. Furthermore, we will only look at matrices of sizes $n \times n$, where n is 2^i and i is between 0 and 7. All our matrices will be dense, meaning that most elements will be filled with non-zero numbers.

Chapter 2

Background

2.1 Fixed point and floating-point representation

Fixed point and floating-point representation are the two most common ways to represent real numbers in a fixed-length, binary string. Fixed point representation is the most basic method to achieve this, and is based on the idea of having a certain number of bits for the integer part of the number and a certain number of bits for the fraction. For example, an unsigned 8 bit fixed point number could have 4 bits for each part and can represent non-zero values from `0000.0001` (0.0625 or $1/16$ in base ten) to `1111.1111` (15.9375 or $255/16$ in base ten). The discrete steps between numbers are fixed and allows for simple hardware to perform operations. However, it lacks dynamic range. The 8 bit example could only represent values smaller than 16.

floating-point representation is another way for computers to approximate real numbers in a standardized, compact form with a predefined number of bits. floating-point representation is very similar to scientific notation in base 2, and is the most common way to represent real numbers in computers. It is more complex, and thus requires more hardware to implement. The following section will explain the IEEE 754 kind of floating-point in greater detail.

2.2 IEEE 754 floating-point

The IEEE 754 standard was established in 1985 and has had minor revisions since then[2], [3]. The standard defines several floating-point formats of different lengths, as well as defines some hard rules that any standard-complying

hardware must adhere to. For the rest of this paper, the terms "float32" and "single precision" are used interchangeably to mean the 32-bit IEEE 754 single precision format, and the term "float64" is be used to refer to the 64-bit IEEE 754 double-precision format.

The standard can be split into four sections:

- The floating-point number format
- mathematical operations
- floating-point exceptions
- conversion between the IEEE 754 & integers, decimal strings as well as other floating-point formats

The main relevant aspects for this study are conversion to other formats and mathematical operations. The standard defines that the conversion to other floating-point formats should be possible for supported formats. Notably, one key aspect is that when converting from higher precision to a format of lower precision, the same rounding rules are used as for Mathematical operations.

The 32-bit IEEE 754 float consists of three parts: one sign bit, eight exponent bits and 23 mantissa bits. This section will give an overview of the IEEE 754 32-bit float and its construction.

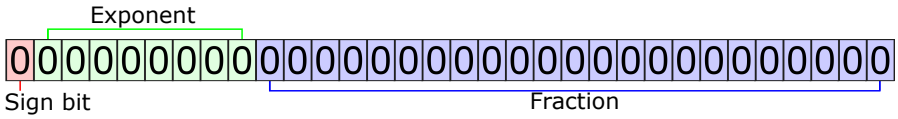

Figure 2.1: The bit structure of a 32-bit float

2.2.1 Sign bit

The first bit is used to represent if the float is a positive or negative number, 1 for negative and 0 for positive.

2.2.2 Exponent bits

The second part is the eight exponent bits. This corresponds to the exponent in base-2 scientific notation. The exponent is represented using an unsigned integer and a so called "bias" is subtracted from the exponent to allow representation of both negative (i.e. float values between 0 and 1) and positive exponents (i.e. float values larger than 1). The bias for a 32 bit float is 127. For

example, if the exponent is 101_2 (5_{10}) then in 32-bit IEEE 754 this exponent would be represented as 101_2 + 1111111_2 = 10000100_2 $(5_{10} + 127_{10} = 132_{10})$.

2.2.3 Fraction bits

The third part consists of 23 fraction bits, sometimes referred to as the mantissa, which represent the decimals in the base 2 scientific notation. There is an implied $1.$ before the decimals. For example, if the bits are $nnnnnn$, the value of the bits are $1.nnnnnn$.

2.2.4 Special bit patterns

This construction is unable to represent the value "0", so special bit patterns were constructed to remedy this. If all bits are 0, this represents a positive 0, or "0.0". If the sign bit is 1, and the rest are 0, this means negative 0, or "-0.0"

The standard also has special bit patterns for negative and positive infinity, as well as exceptional values for not-a-number values (also known as NaN-values) that can occur when invalid operations are performed, such as divide by zero or multiplications with infinity. There are in total 16,777,214 different bit patterns defined as NaN values. But, as this thesis focuses on precision, they are not explained in further detail.

2.2.5 Higher precision formats

IEEE 754 also specifies the 64 bit double-precision format, which works the same way as the 32 bit floats, except with more bits. like a float, A double also only has one sign bit, but instead has 11 bits for the exponent, and 52 explicitly stored fraction bits. This allows for greater dynamic range and precision over floats, but at the cost of memory usage.

There is also the 80 bit long x86 extended precision format. Many compilers, including GCC for x86-64 Linux, will use this format for "long double" variables. With it's 63 fraction bits, this format allows for even more precision over the double-precision format.

This format works similarly to the IEEE 754 standard, but there are some differences (other than the obvious length/precision difference). However, these differences are not relevant to this study and will not be explained further.

2.3 Posit

Posit representation is constructed in a similar way to floating-point format. Just as floats, they have a sign bit, an exponent and a fraction, but they also have a 4th part called the "regime". Each part is explained in the following corresponding subsections.

A general Posit could have any bit length and any exponent length. However, the 32 bit Posit with a 2 bit exponent, often denoted as a "Posit(32,2)", is one of the more common Posits and is the focus of this study. This section will therefore only explain how a Posit(32,2) is constructed to minimise bloat. The Posit(32,2) will from now on simply be referred to as a Posit32.

For a detailed explanation on how a general Posit is constructed, refer to Gustafson and Yonemoto's article[1] and the Posit standard documentation[4].

2.3.1 Sign bit

The sign bit is very similar to a float's sign bit. 0 means positive and 1 means negative. If the sign bit is negative, apply the 2's complement (negate all the bits and subtract 1) before decoding the regime, exponent and fraction.

2.3.2 Regime

The regime is what truly sets the Posit system apart from IEEE 754. Instead of being a specified number of bits, the regime can vary in length. The length of the regime corresponds to the number of consecutive identical bits followed by a terminating opposite bit. In figure 2.2 you can see a 32 bit Posit with a regime of four "1" bits and a total length of five with the terminator.

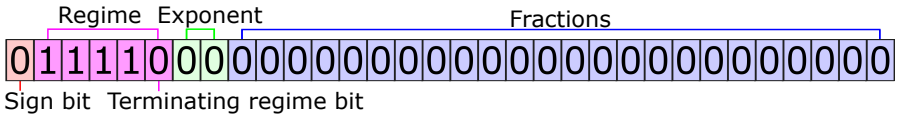

Figure 2.2: The bit structure of a 32-bit Posit with a 2 bit exponent. Note that the regime length is just an example, and can vary in length. This bit pattern represents the value 65536.0

The value of the regime is dependent on the number of exponent bits used, in the 2 bit case $16^{numberofbits-1}$ or $16^{-numberofbits}$

2.3.3 Exponent

The exponent is only 2 bits long, as opposed to the float's 8 bits. This is possible because the regime achieves similar objectives as the float exponent, thus allowing the Posit exponent to be considerably shorter. The exponent does not have a bias or any other way to become negative. The value is simply 2^{exp}

There is a possibility that the exponent may actually be 1 or 0 bits long if the regime takes up 30 or 31 bits, thus not leaving any room for the full 2 bit exponent. This does not have any impact on how it is calculated, but obviously limits the values it can represent.

2.3.4 Fraction

The fraction consists of the left over bits. It can therefore be between 28 and 0 bits long. Other than the variable length, this part behaves almost identical to the float's fraction described in section 2.2.3. There is an implied $1.$ in the beginning, and then the fraction bits follow and make up a fraction in base 2.

2.3.5 Special bit patterns

Posits have much fewer special bit patterns compared to floats[5] (see section 2.2.4). There is only one pattern for 0, which is all bits zero. Then there is a single one bit followed by 31 zero bits which represent negative and positive infinity, undefined and unrepresentable values.

2.3.6 The quire format

Every Posit length has a corresponding datatype called a quire. The 32 bit Posit has a quire length of 512 bits, according to the Posit standard [4]. The quire can be seen as an intermediate format used to accumulate a lot of numbers without rounding, thus eliminating compounding rounding errors (see section 2.4). It is a mandatory part of the Posit standard.

The 512 bit quire is a form of base 2 fixed precision (see section 2.1). It consists of:

- 1 sign bit
- 31 carry-guard bit
- 240 integer bits
- 240 fraction bits

Because of the 31 carry-guard bits, the quire is guaranteed to not overflow when summing up to $2^{31} - 1$ Posit products.

2.4 Precision and loss of precision

Because there is an infinite number of real numbers (some of which are infinitely long, such as π), they have to be rounded to fit into the nearest representable value of a 32 bit float or Posit. Whenever a floating-point operation is performed, there is a chance the result will end up outside of what is representable and the number has to be rounded once again. If these operations are carried out multiple times, the error will propagate and grow larger[6].

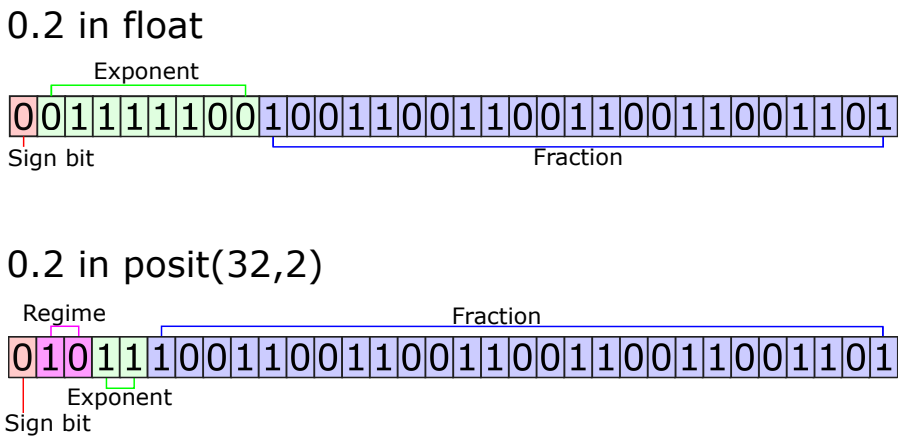

Figure 2.3: The bit values representing an approximation of 0.2_{10} in both a float and a Posit.

What decides the precision of both a float and a Posit is the length of the fraction. The maximum rounding error of a float is essentially what value a potential 24:th fraction bit with the value "1" would have, if it existed, multiplied by 2^{exp}.

In figure 2.3 you can see how the number 0.2_{10} is represented in both floats and Posits respectively. 0.2_{10} is a number that can not be exactly represented in either formats, and has to be rounded.

In figure 2.4 you can see the fractions from figure 2.3 side to side. Both are rounded, but the Posit fraction is longer and is therefore more exact. This is possible because the variable regime is very short for 0.2_{10}, and is the basis for why Posits should be able to be more precise than floats.

However, in cases of very large regimes, the precision can become worse in a Posit than a float. In figure 2.5 the value of 0.000000001_{10} is represented in

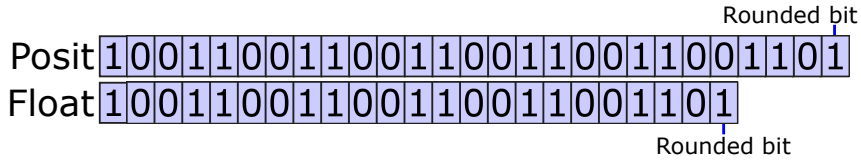

Figure 2.4: Fraction comparison between the values from figure 2.3 representing 0.2_{10}*. The last bit is rounded up in both of these cases.*

both formats, and one can observe that while the float has the usual 23 fraction bits, the Posit only has 20 to make space for the large regime.

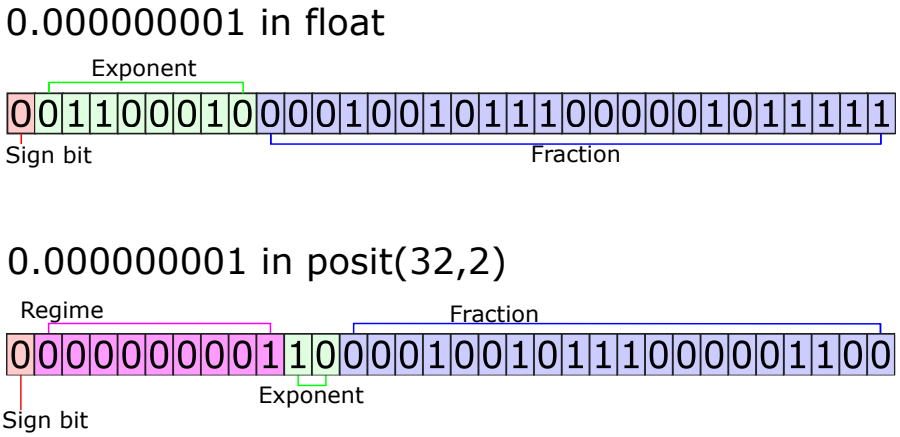

Figure 2.5: The bit values representing an approximation of 0.000000001_{10} *in both float and Posit formats. The float is rounded down, while the Posit is rounded up, which is why the fraction differs at the end.*

There are two common ways to talk about error: absolute error and relative error. Absolute error is simply how far away from the exact answer you are; the absolute value of the calculated answer minus the exact answer. Relative error is the absolute error divided by the exact answer. The relative maximum rounding error of floating-point numbers is constant, as the fraction has a fixed number of bits, while Posits has a varying maximum relative rounding error [7].

2.5 Matrix multiplication

Matrix multiplication is a central operation in linear algebra. It has many different applications within computer science, including physics simulation, computer graphics and machine learning. In some applications matrix multi-

plication can be very time consuming, because many floating-point operations have to be performed, and each of these operations introduces a rounding error.

2.5.1 Available algorithms

There are several algorithms for matrix multiplication. The most common one is the iterative approach. If A, B and C are $n \times n$ matrices, and $C = AB$, then C is calculated in the following way:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj} \tag{2.1}$$

This algorithm has a time complexity of $\mathcal{O}(n^3)$ and is usually implemented as 3 nested loops. This is the algorithm used for the implementation of this study.

There are also other algorithms with a better theoretical time complexity. One of the most common of these algorithms is the Strassen algorithm [8], although it is much more complicated and is only more efficient in practice when multiplying matrices around 500×500[9] or larger. All the other algorithms with better time complexity share this property of only being practical for very large matrices.

2.5.2 Fused multiply-add

The fused multiply-add instruction allows for both a multiply and an add operation to be performed with only a single floating-point rounding. As illustrated in equation 2.1, both multiplication and addition is used in matrix multiplication. This operation can therefore reduce the amount of rounding operations performed and as a result, reduce precision loss.

In 2008, a revision of IEEE 754[3] included the fused multiply-add from an optional extension to a mandatory part of the standard and must therefore always be implemented on standard complying hardware.

2.6 Previous research

2.6.1 Beating floating-point at its Own Game

The research study "Beating floating-point at its Own Game: Posit Arithmetic" [1] from 2017, can be considered the pioneer of Posits, due to it being the first study about the topic. The study compared Posits and Floats when it

came to precision & representability when doing a selection of mathematical operations. A sample of the operations were: Addition, Multiplication and Square root. It concluded that in general Posits performed better than Floats from a precision aspect, as it more often than floats returned a more exact value at a given bit length. This means that there were operations in which Floats performed better in terms of precision than Posits; for the most part Posit was more precise. In the aspect of closure, Posits beat them in multiplication around 25% of the results where not representable in Floats. They either underflowed, overflowed or returned NaN. Posit was able to represent all except for 0.00305%.

Unfortunately, the study has to be considered biased, as one of the authors (John Gustafson) is the chairman of the NGA Working Group and usually credited as the inventor of the Posit. The other author (Isaac Yonemoto) is not in the NGA working group, but has worked together with the team on multiple projects.

2.6.2 Assessing the Precision Improvement in HPC

Another study that has compared the precision of Posit and floats has done so within the HPC-field [10]. The study used various HPC-kernels to compare how well the different formats performed. It concluded that s had higher precision than floats for all of their kernels. The negative aspect of Posit was its performance. The Posit calculations could be around three times slower than float, which was attributed to the absence hardware support.

Since the scope of our study is limited to precision, performance aspects will not be evaluated the same way as the study above. The precision improvement, using Posits, was found to be in the range of 0.6 to 1.4 decimals digits. However, one aspect that the study above did not look at was how the precision propagates with the problems size that is being measured, which our study does.

Chapter 3

Method

3.1 Software

In order to use Posit32 on an x86-64 computer without hardware support, the software implementation of Posit known as Softposit [11] was used. The functionality that is utilised from this library is casting floats to Posits, casting Posits to floats, calculations using the corresponding quire datatype and fused multiply add. How these functionalities are used is described in detail in section 3.2.

To combat hardware implementation differences, to increase portability and to make sure we were using a fair comparison, we used the library Softfloat [12] for representing float32. Softposit is based on Softfloat, which should mean the functionality is equivalent. In listing 3.1, the pseudo code for the program is shown.

3.2 Benchmark program

The code for the benchmark program [13] was written in Python3.6 using the software libraries Softfloat, Softposit, Numpy and CSV. For a brief overview in pseudocode, see listing 3.1. The following subsections will explain the program in greater detail.

3.2.1 Generating the matrices

The matrices were first initialized as zero matrices in their corresponding data type.

```
1  input: float randomInterval
2  output: float[] errors
3
4  begin
5  for size from 2 → 128 where size ∈ power of 2
6    loop 1000 times
7      float64[size][size] float_1 ← random numbers
       ↪ ∈ [−randomInterval; randomInterval)
8      float64[size][size] float_2 ← random numbers
       ↪ ∈ [−randomInterval; randomInterval)
9
10     posit32[size][size] posit_1 ← float_1
11     posit32[size][size] posit_2 ← float_2
12
13     float64[size][size] float_result ← float_1 × float_2
14     posit32[size][size] posit_result ← posit_1 × posit_2
15
16     for a from 0 → size
17       for b from 0 → size
18         error ← error + |posit_result[a][b]−float_result[a][b]|
19       end
20     end
21     errors.append(error)
22   end
23 end
24 output errors
25 end
```

Listing 3.1: Pseudocode outlining an overview of the benchmark program. This only show the error comparison for posit32, but the same process is used for both float32 and posit32 using quire.

Listing 3.2 shows the python function corresponding to Posit matrix generation. This function is also used for initializing the result matrices used to store the output of the matrix multiplication.

```
1 def fillWithZerosPosit32(size):
2     return [
3         [posit32(0.0) for i in range(size)]
4         for j in range(size)
5         ]
```

Listing 3.2: This function generates a zero initialized posit32 matrix of desired size, then returns it. Utilizes the Softposit library[11].

3.2.2 Generating random numbers

The generation of random numbers is done in float64 in order to achieve higher precision than our comparisons. Float64 was deemed to be high enough, as

any float32 and posit32 can be exactly represented in float64[7] without any loss off precision.

The function in listing 3.3 receives interval and size as input parameters. The interval is used in the random uniform call to return a random number between the negative and positive of the interval value. The size is used to define the amount of rows and columns in the matrix.

```
1 def fillWithRandom(interval, size):
2     return [
3         [random.uniform(interval*-1, interval) for
  ↪ i in range(size)]
4         for j in range(size)
5         ]
```

Listing 3.3: This function fills a matrix with random numbers in the given interval range. The format is float64

3.2.3 Casting the randomized high precision to lower precision

To cast the numbers from float64 to posit32 and float32, a function was made which takes in two matrices and iterates through them. See listing 3.4 for the posit32 version. The function iterates through each element in the float64 matrix, casts it to posit32/float32 and stores it in the posit32/float32 matrix.

```
1 def convertFloat64ToPosit32(Float64Matrix,
  ↪ Posit32Matrix):
2     for j in range(rows):
3         for i in range(columns):
4             Posit32Matrix[j][i] =
5             posit32(Float64Matrix[j][i])
```

Listing 3.4: This function converts a float64 matrix to a posit32 matrix using Softposit's casting functionality.

3.2.4 Matrix multiplication

For matrix multiplication, we chose to use the iterative algorithm explained in section 2.5.1. Because this study was limited to matrices of size 128 or

smaller, the Strassen algorithm is unnecessarily complex and will not provide any benefits.

The implementation iterates through each element using three for loops. The outer for loop iterates through the rows of m1 (matrix 1), the middle for loop iterates through the columns of m2 and finally the inner for loop iterates through the rows of m2, implementing the iterative algorithm seen in 2.5.1. For each mathematical operation made between floats, a rounding error is introduced. Therefore the fused multiply add is utilised as you can see on row 6 in listing 3.5.

```
1 def matrixMultiplicationPosit32(m1, m2, result):
2     for i in range(len(m1)):
3         for j in range(len(m2[0])):
4             for k in range(len(m2)):
5                 result[i][j] =
6                 result[i][j].fma(m1[i][k], m2[k][j
  ↪ ])
7                 #Software fused multiply add
```

Listing 3.5: Implementation of the matrix multiplication algorithm for posit32.

The matrix multiplication works almost identically for float32 as it does for posit32 in listing 3.5. However, the implementation for posit32 using quire has couple of extra steps, as can be seen in listing 3.6. Here, a quire variable "q" is created (explained in section 2.3.6) which accumulates the value for each element in the final matrix and only rounds once.

```
1 def matrixMultiplicationQuire32(m1, m2, result):
2     q = sp.quire32()
3     for i in range(len(m1)):
4         for j in range(len(m2[0])):
5             for k in range(len(m2)):
6                 q.qma(m1[i][k],m2[k][j])
7             result[i][j] = q.toPosit()
8             q.clr()
```

Listing 3.6: Function very similar to listing 3.5, but utilising quire to avoid premature rounding.

The comparison between float32 and plain posit32 without using the quire intermediate format is the more fair comparison to make, as they have similar rounding rules. Quire was chosen to be included in this study as it is part of the Posit standard, as described in the documentation [4].

3.2.5 Calculation of the error

The error calculation from the matrix multiplication was performed by comparing the matrix product of the 32 bit formats to the matrix product in float64. The float64 product could be thought of as the answer key that the lower precision formats were compared to.

The error was measured in two ways. First, as the absolute error between two elements: one element from the float64 result matrix and the other element was the corresponding element in posit32/float32 result matrix. Second, as the relative error between the two elements (see section 2.4). These errors are then summed to long double variables (80 bit x86 extended precision as seen in section 2.2.5, but will be platform specific) to calculate the total error for the matrix multiplication. See listing 3.7 for the function's code.

```
def sumDiffOfMatrixes(m1, m2):
    sum = np.longdouble(0)
    sumRelative = np.longdouble(0)
    for j in range(len(m1)):
        for i in range(len(m1)):
            error = np.longdouble(abs(float(m1[j][i
    ↪ ]) - float(m2[j][i])))
            sum += error
            if m2[j][i] != 0:
                sumRelative += error / abs(float(m2
    ↪ [j][i]))
    return (sum, sumRelative)
```

Listing 3.7: This is the code used to calculate the absolute and relative error of a matrix.

3.2.6 Running the program

Several instances of the program was executed at once on an AMD Zen 2, Ryzen 3800X x86-64 processor, with different random intervals specified as program parameters. This way, hardware parallelism was utilised without risking race hazards. The program was executed in Ubuntu using Python 3.6. The output of the program is saved to a file.

Chapter 4

Results

Each test iteration that was run in the benchmark program generated six data points. Three points were the total precision difference (absolute error) between 3 test matrices and a solution matrix (as described in the algorithm in listing 3.1). The test matrices consisted of one posit32 matrix without quire, a posit32 matrix with quire and a float32 matrix. The other three data points were the total relative error on the same three matrices. This was repeated 1000 times for each random interval and matrix size resulting in 264,000 rows of data, with a total of 1,584,000 data points.

In the table in appendix A, the average absolute error is shown in posit32 (with and without quire) and float32 for each matrix size and interval. Each data point is an average of the total error in 1000 generated matrices.

4.1 Posit and float error difference

In figure 4.1 one can see four charts, each chart representing a different matrix size. The X-axis represents the interval in which the random numbers were generated within, represented by the upper bound of the interval. For example, "10" represents the interval $[-10; 10)$. The Y-axis represents the average relative error per element in the matrix. This measure was chosen to more easily convey how far off one can expect the errors of a single matrix element to be for a given matrix size and random interval. Absolute error would likely have produces monotonically increasing graphs that would be hard to differentiate.

As can be seen in the graphs, posit32 outperformed float32 for the intervals $[-0.01; 0.01)$, $[-100; 100)$ and the intervals in between. For the intervals $[-0.001; 0.001)$ and $[-1000; 1000)$ the difference between posit32 and float32 is much smaller, and which one performs better differs on a case by case ba-

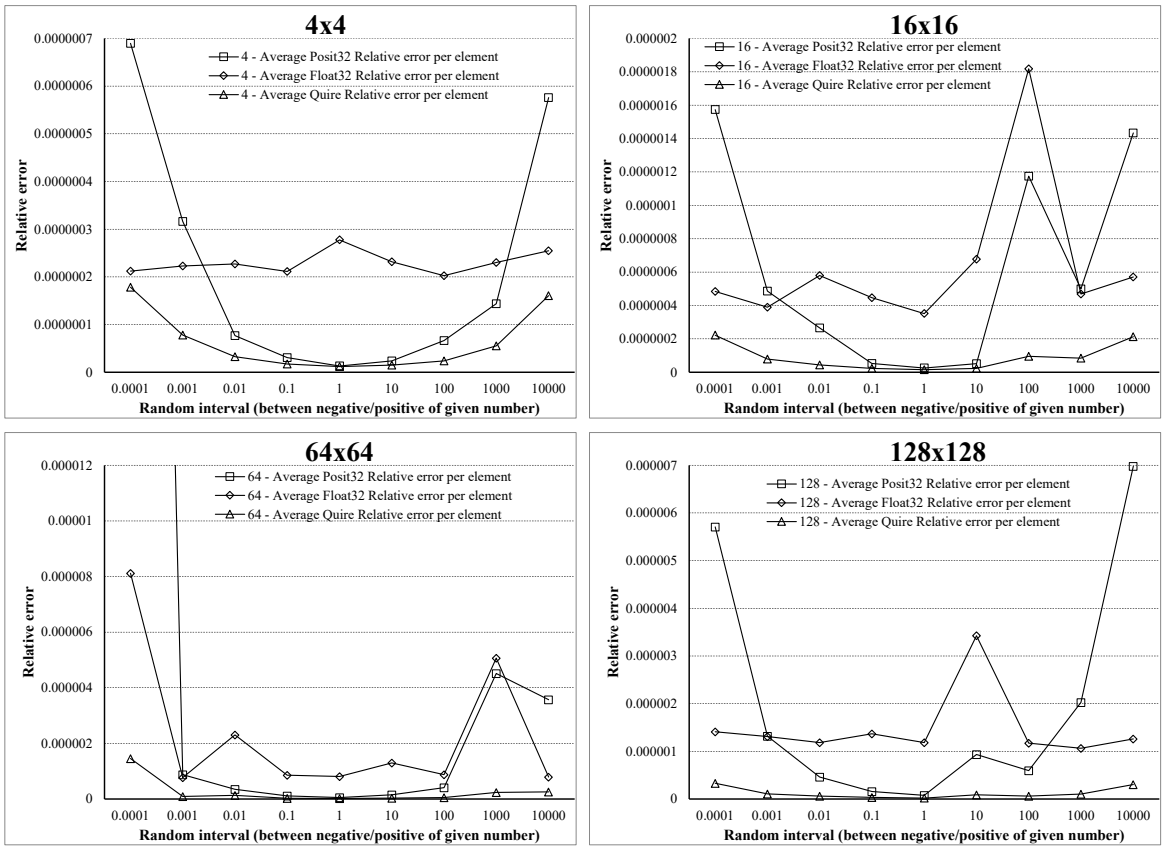

Figure 4.1: These four graphs shows how big the average relative error per element was for different random intervals. Lower error is better. Note that the value for posit32 relative error within the interval $[-0.0001; 0.0001)$ *is outside the chart, as it was too large to display properly.*

sis. As expected, posit32 utilizing quire performed better than posit32 without quire in all cases, and the improvement increased as the matrix size increased.

One feature of figure 4.1 that stands out is that there seem to be a lot of variance in the results, which results in a lot of peaks, such as for the interval 100 in 16x16, and the interval 10 in 128x128. To determine if this was caused by a few statistical outliers, the median was examined instead of the average in figure 4.2. This figure features the two most volatile graphs from figure 4.1, and shows that the median behaves more predictably than the average, suggesting that it might be a better measure than average. Both of the graphs in figure 4.2 shows a similar trajectory for posit32 without quire compared to float32, but when quire is used the improvement once again increases with

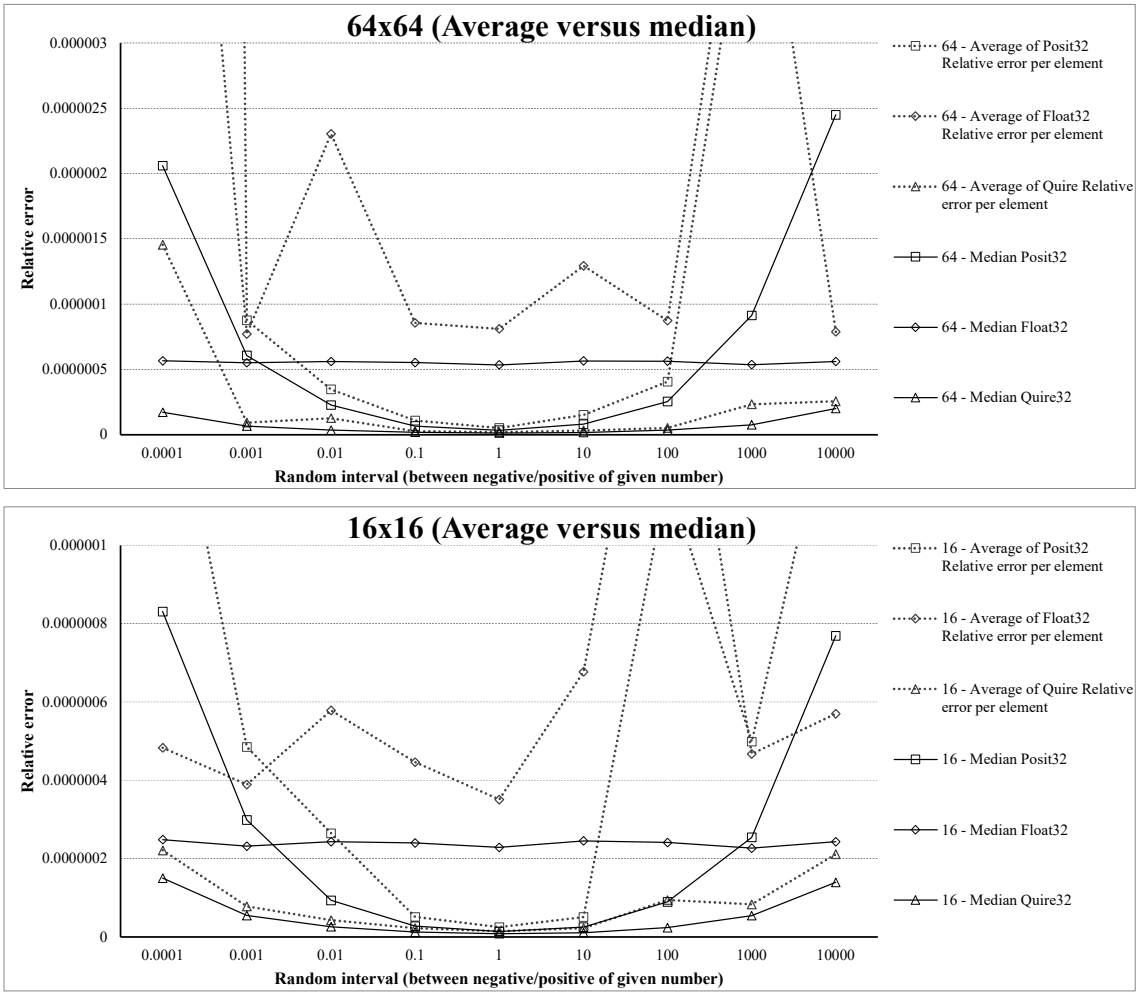

Figure 4.2: This figure compares the average error per element from figure 4.1 to the median error per element. The average is displayed in dotted lines, while the median is displayed in solid lines.

matrix size.

Table 4.1 focuses on showing the improvement of using posit32 in the interval $[-1; 1)$, as this has shown to be the interval where posit32 has the most advantages over float32. This can be viewed as a best case scenario for posit32. But the varying fraction length is a double edged sword, and for other, much larger or smaller intervals, posit32 can perform much worse. For the matrices generated in the interval $[-10^{-9}; 10^{-9})$, the mean relative error for posit32 is 31 times larger than the float32 error. See table 4.2 for a few other selected intervals. The intervals are once again represented by the upper bound of the interval, which means "10^3" represent the interval $[-10^3; 10^3)$.

Table 4.1: This table shows the average and median improvement for using posit32 over float32 within the interval $[-1; 1)$ for all matrix sizes tested.

Matrix size	Average relative error improvement	Median relative error improvement
2×2	22.35547677	16.14542527
4×4	21.24683736	15.20958078
8×8	25.87305381	16.00270937
16×16	13.85138341	16.02109097
32×32	16.74196663	15.78342707
64×64	15.91379806	15.78267919
128×128	15.68357108	16.31120276
Total average	**18.80944102**	**15.89373077**

Table 4.2: This table show how many times better a float32 is compared to a posit32 for some selected intervals.

Random number interval	Median float32 relative improvement
10^{-12}	36641.78851
10^{-9}	1162.108262
10^{-6}	36.4867118
10^{-3}	1.19037652
10^{3}	1.036929118
10^{6}	31.41608596
10^{9}	1001.849231
10^{12}	31410.3665

4.2 A closer look at quire

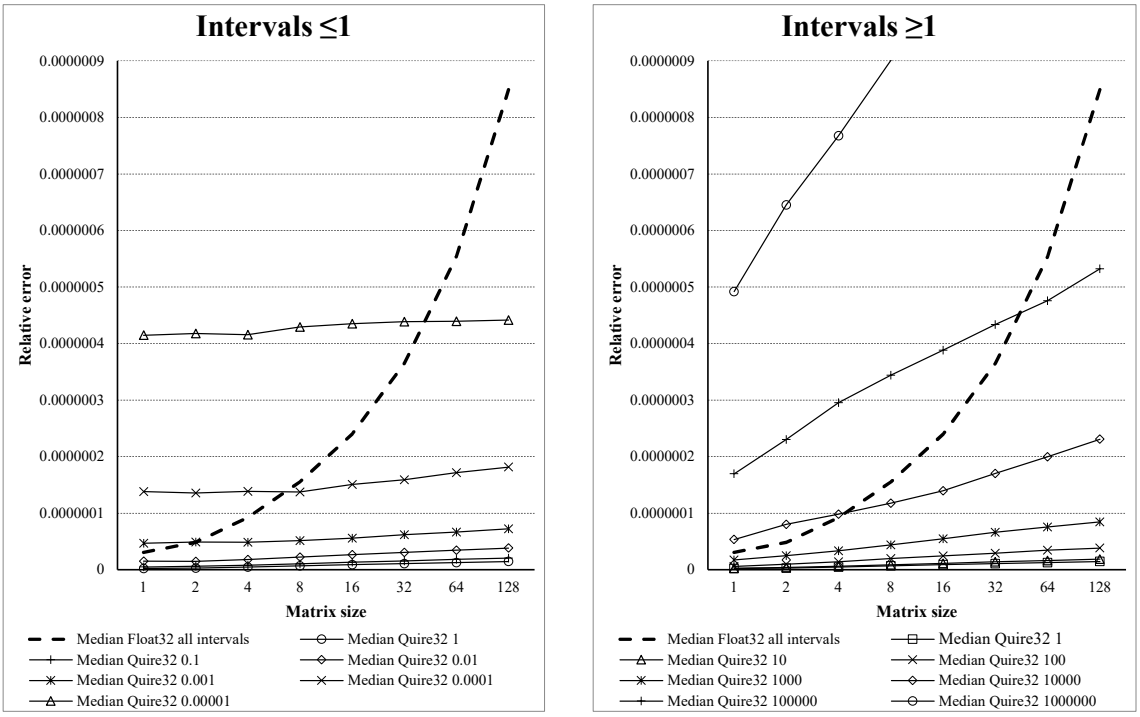

Figure 4.3: This figure compares float32 to posit32 utilising quire for different intervals. Only the intervals that fit on the chart are shown. "Median Float32 all intervals" is an average of the means of float32 relative error per element.

Looking more closely at posit32 utilising quire, it is visualised in figure 4.3 how different intervals affect weather posit32 utilizing quire is better than float32 or not. Median relative error per element were once again used instead of average relative error per element. The line named "Median Quire32 10" symbolised the median relative error per element for a matrix with elements generated in the interval $[-10; 10)$. Note that only one float32 line is present. This is because the maximum relative error of a float32 is constant[7], which in this case resulted in the lines being almost perfectly superimposed. To reduce clutter, the average of all mean float32 intervals was calculated and is shown in the dotted line. More intervals of posit32 using quire than were shown on the graph were calculated, but their relative error was consistently higher than float32.

4.3 Probability of posit32 outperforming float32

Absolute error

Prob. Posit better Random interval ↓	Matrix size → 1 (not a matrix)	2	4	8	16	32	64	128	Total
1E-16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-12	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-11	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-10	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-09	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E-08	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-07	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E-06	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.025%
1E-05	3.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.375%
1E-04	6.6%	2.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.100%
1E-03	16.0%	31.4%	16.7%	7.0%	2.2%	0.3%	0.8%	2.3%	9.588%
1E-02	48.6%	94.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	92.825%
1E-01	78.2%	99.8%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	97.250%
1E+00	91.4%	99.9%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.913%
1E+01	89.5%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.688%
1E+02	75.4%	98.3%	99.9%	100.0%	100.0%	100.0%	100.0%	100.0%	96.700%
1E+03	39.1%	72.1%	62.2%	12.1%	0.0%	0.0%	0.0%	0.0%	23.188%
1E+04	16.1%	9.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	3.150%
1E+05	6.3%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.850%
1E+06	1.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.238%
1E+07	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.050%
1E+08	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+09	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+10	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+11	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E+12	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%

Relative error

Prob. Posit better Random interval ↓	Matrix size → 1 (not a matrix)	2	4	8	16	32	64	128	Total
1E-16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-12	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-11	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-10	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-09	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-08	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.025%
1E-07	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.038%
1E-06	0.4%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.063%
1E-05	2.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.1%	0.0%	0.275%
1E-04	5.9%	3.2%	1.3%	1.8%	0.9%	0.8%	0.7%	0.3%	1.863%
1E-03	17.0%	32.3%	32.1%	24.5%	19.7%	23.8%	29.8%	38.4%	27.200%
1E-02	48.4%	92.5%	97.4%	97.4%	98.3%	98.4%	98.5%	99.1%	91.250%
1E-01	82.1%	99.8%	99.6%	99.9%	99.9%	99.9%	99.9%	100.0%	97.638%
1E+00	93.1%	99.9%	99.9%	100.0%	99.9%	100.0%	99.9%	100.0%	99.088%
1E+01	93.2%	99.9%	100.0%	99.9%	100.0%	100.0%	100.0%	99.9%	99.113%
1E+02	78.1%	98.4%	99.2%	99.1%	98.6%	98.1%	98.2%	98.6%	96.038%
1E+03	41.9%	71.9%	76.6%	69.2%	30.7%	8.3%	3.0%	2.2%	37.975%
1E+04	15.5%	10.7%	3.2%	2.2%	1.4%	0.9%	0.2%	0.4%	4.313%
1E+05	5.9%	1.1%	0.2%	0.5%	0.0%	0.1%	0.0%	0.1%	0.988%
1E+06	2.4%	0.0%	0.0%	0.1%	0.0%	0.1%	0.0%	0.1%	0.338%
1E+07	0.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.075%
1E+08	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+09	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+10	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+11	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+12	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E+13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%

Figure 4.4: This table visualises how often the posit32 matrix multiplication utilizing quire was better than the float32 matrix multiplication. Each table position had a sample size of 1000 randomized matrices generated.

The purpose of this section is to give a better overview of when posit32 is better than float32, and leave out by how much better it is. This is achieved by counting how often the total absolute error and relative error of the posit32 matrix was lower than the total absolute error and relative error of the corresponding float32 matrix. If the absolute/relative error was lower for the posit32 we consider the posit32 matrix to have outperformed the float32 matrix.

Figure 4.4 shows how often the posit32 matrix outperformed the float32 matrix for both of these metrics. Absolute error is shown in the table to the left, and relative error is displayed in the table to the right. For example, for the random interval $[-10^3; 10^3)$ in matrices of size 8×8, Posits outperformed floats in $121/1000$ cases measured by absolute error, resulting in 12.1 % on the chart to the right. For relative error, posit32 outperformed float32 in $692/1000$ cases in this interval, resulting in 69.2 % on the chart to the right. For matrices of the same size, but in the interval $[-1; 1)$ Posits outperformed floats in $1000/1000$ cases in both metrics. As previously stated, this table does not tell us how much better or how much worse Posits are compared to floats for any

given interval or matrix size, merely how often they are. For the performance difference, see section 4.1. This result confirms that posit is better between $[-0.01; 0.01)$ and $[-100; 100)$ for all matrix sizes as figure 4.1 suggested. It should be noted that posit seems to have a small improvement in the interval $[-1000; 1000)$ for matrices of size 2×2 and 4×4.

Figure 4.5 shows tables similar to 4.4 that instead focuses on posit32 matrices calculated using quire. The table to the left shows when posit32 matrices using quire is better than matrices using float32 for absolute error. Compared to the absolute error table in 4.4, this is shows strictly better results for the accuracy. It also shows that unlike table 4.4, the benefits of using posit32 with quire increase as the matrix size increases.

Posit32 using quire compared to Float32

Prob. Quire better Random interval ↓	Matrix size → 1 (not a matrix)	2	4	8	16	32	64	128	Total
1E-16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-12	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-11	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-10	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-09	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E-08	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-07	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E-06	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.025%
1E-05	3.0%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.400%
1E-04	6.6%	4.9%	0.5%	0.0%	0.6%	99.4%	100.0%	100.0%	39.000%
1E-03	16.0%	47.0%	81.0%	100.0%	100.0%	100.0%	100.0%	100.0%	80.500%
1E-02	48.6%	97.1%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	93.213%
1E-01	78.2%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	97.275%
1E+00	91.4%	99.9%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.913%
1E+01	89.5%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.688%
1E+02	75.4%	99.1%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	96.813%
1E+03	39.1%	84.4%	91.6%	100.0%	100.0%	100.0%	100.0%	100.0%	89.388%
1E+04	16.1%	17.9%	4.1%	0.7%	0.0%	0.0%	0.1%	100.0%	17.363%
1E+05	6.3%	1.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.950%
1E+06	1.9%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.250%
1E+07	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.050%
1E+08	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+09	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+10	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+11	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E+12	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%

Posit32 with/without quire compared

Average of Test Random interval ↓	Matrix size → 1 (not a matrix)	2	4	8	16	32	64	128	Total
1E-16	0.0%	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.050%
1E-15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-12	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-11	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E-10	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E-09	0.0%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.063%
1E-08	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.038%
1E-07	0.0%	1.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.163%
1E-06	0.0%	0.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.113%
1E-05	0.0%	2.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.300%
1E-04	0.0%	2.4%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.313%
1E-03	0.0%	5.7%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.738%
1E-02	0.0%	10.0%	0.8%	0.0%	0.0%	0.0%	0.0%	0.0%	1.350%
1E-01	0.0%	19.1%	5.9%	0.0%	0.0%	0.0%	0.0%	0.0%	3.125%
1E+00	0.0%	23.1%	14.4%	0.0%	0.0%	0.0%	0.0%	0.0%	4.688%
1E+01	0.0%	18.9%	6.4%	0.0%	0.0%	0.0%	0.0%	0.0%	3.163%
1E+02	0.0%	14.1%	2.6%	0.0%	0.0%	0.0%	0.0%	0.0%	2.088%
1E+03	0.0%	9.5%	0.9%	0.0%	0.0%	0.0%	0.0%	0.0%	1.300%
1E+04	0.0%	5.3%	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.713%
1E+05	0.0%	4.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.538%
1E+06	0.0%	1.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.188%
1E+07	0.0%	0.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.088%
1E+08	0.0%	0.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.075%
1E+09	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+10	0.0%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.025%
1E+11	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E+12	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+13	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%
1E+15	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.013%
1E+16	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000%

Figure 4.5: The table to the left visualises how often the posit32 matrix using quire performs better than the float32 matrix. The table to the right visualises how often the basic posit32 matrix multiplication without quire performed better than the posit32 matrix multiplication using quire. Error measured using absolute error. Each table position had a sample size of 1000 randomized matrices generated.

The table to the right in figure 4.5 shows how often the posit32 matrix was better than posit32 (again in terms of absolute error). The highest occurrence of posit32 being better than posit32 using quire was at the interval $[-1; 1)$ for 2×2 matrices, and was 231/1000. This shows that using the quire to avoid intermediate rounding errors is more likely to produce a better result.

Chapter 5

Discussion

5.1 Analysis of the results

This section will provide analysis of the results shown in the previous chapter. The effects of quire will be discussed, the usefulness of Posits and how the results compare to previous research. This chapter will also provide some reflection over the methods used.

5.1.1 Variance in the results

As can be seen in figure 4.1, the average relative error varied a lot, especially for float32 and posit32. In theory, the average relative error for float32 should be a constant if enough samples were provided. In section (4.1) of the results, this was briefly attributed to statistical anomalies in the result. Now we are going to evaluate this closer.

Let us consider the calculation $1000.1 \cdot 1000.1 - 1000.2 \cdot 1000$. It is completely reasonable to expect a calculation similar to this one to be randomly generated every once in a while in our benchmark. After all, a lot of numbers are generated. If carried out in float32, the answer would be -0.062500, and if carried out in float64, the answer would be 0.010000. This makes the absolute error 0.0725, and the relative error 7.25. This is significantly higher than the relative error around 0.0000004 that can be seen on the graph, and just a single matrix element with a relative error of 7.25 in one out of 1000 16×16 matrices could increase the average by several hundred times, and the data point would be off the chart. This problem could be remedied by generating several thousand times more samples. Although this would be very time consuming, and there would still be a small chance to get a single data point with such a

large relative error that it still skews the data significantly. We therefore believe our choice to focus more on the median relative error instead of the average relative error was a good decision.

With all this in mind, if we were to redo the study it would probably be wiser to focus on the absolute error instead of the relative error for all calculations. This way, division by very small numbers would be avoided. This would have the negative effect that the errors between different intervals gets harder to compare, and achieving graphs like those in 4.1 would have been difficult. Although dividing by the highest end of the random interval could be a possible solution to this.

Despite this problem, we still believe the result is conclusive. The data using median relative error behaves very predictably in all cases, and it is very unlikely that this happened by chance. We therefore do not think the conclusion would have changed much if some sort of adjusted absolute error would have been used instead.

5.1.2 When is Posit useful

The results showed that posit32 could be up to 19 times better than float32 in the best case scenario. Although this "golden zone" for 32 bit Posits is quite narrow, and they only consistently performed better than float32 in five of the tested intervals. This is because matrix multiplication with very large/small numbers has a tendency to create even larger/smaller numbers respectively. Multiplying two matrices with elements in the thousands is likely to end up with a product with elements in the millions. But while posits can be up to 19 times better than floats in ideal scenarios, they can be much worse than that in others, as shown in table 4.2. We think that this characteristic makes posit32 unsuitable to replace floats for general matrix calculations.

A study by Dinechin, Forget, Muller, *et al.* showed some of the shortcomings of Posits[7] and also came to the conclusion that posit32 performs worse than float32 when mixing large ans small numbers, which can happen in general calculations. However, the strength of Posit can definitely be used in practice. As mentioned in the introduction, matrix multiplication is used in deep learning. In deep learning the values of the elements in the matrices are often between 0 and 1. In this interval our results showed that posit32 performed better than float32 in all cases, regardless of quire usage. Deep Learning has actually to work with much lower precision than float32[14] and the study [7] showed that posit16 was better than posit32 if you compare them to the IEEE 754 format of the same bit size. With these two studies in mind,

we theorise that Posits could have a future in their lower precision versions for deep learning applications.

Another good question to ask is how large matrices one really need to multiply. Even outside of machine learning, normalized matrices are very commonly used in computer science fields. Although the the interval in which posit32 outperforms float32 is arguably somewhat narrow, it could be enough for most applications.

Another argument in favour of Posits is that one could scale the data of small or large magnitudes closer to 1 before running the calculations on them, and then scale it back. That way, one could enjoy the benefits of the increased precision outside of the ideal zone. Unfortunately this does not work with data that mixes a lot of small and large numbers. It is also a potential way to introduce more error and demands a lot of technical skill and knowledge of the data from the user.

5.1.3 The quire intermediate format

On average, the posit32 matrix multiplication using quire performed better than the posit32 matrix multiplication without quire for all matrix sizes and all random intervals. This is to be expected as the same storage format (posit32) is used, but without intermediate rounding for the matrix multiplications. However, there were some individual cases where posit32 using quire resulted in a less exact answer than plain posit32, as can be seen in the right table in figure 4.5. In theory, using quire should strictly perform better than not using it, so we believe this result is simply two or more errors canceling each other out and achieving a smaller error purely by chance. The fact that this phenomenon becomes less and less likely as matrix size increases supports this theory, as there are more opportunities for intermediate rounding errors, and achieving a better answer by luck becomes less and less likely.

Judging from these results only, posit32 using quire is inarguably superior to plain posit32. But there is another trade-off outside of this study's scope; quire would require a significant footprint on the chip if it was to actually get implemented in hardware. It may also be execute slower. Furthermore, the principle of a fixed point accumulator for calculating dot products does not have to be tied to the Posit format. One 2018 study by Koenig looked at the feasibility of implementing such an accumulator for IEEE 754 single or double-precision formats in hardware[15], and the results were promising. The extra precision this could give

5.1.4 Posits results compared to previous research

A previous study by Chien, Peng, and Markidis showed the precision could be improved by 0.6 to 1.4 decimals digits for a certain suite of HPC benchmarks [10]. When the interval in which the random numbers were generated in was around 1, our results showed that Posits outperformed floats. In fact, for all the matrix sizes we calculated Posits were on average 18.8 times better than float32 in terms of relative error. In terms of decimal digits, this would translate to 1.3 decimal digits of improvement. This is in the range of what the study by Chien, Peng, and Markidis had shown Posit to improve by. However, for the intervals that were further from 1, the improvement was either smaller or even negative, meaning that floats performed better.

5.2 Limitations

A downside with the design of this study was the granularity of the results. There were only five tested intervals where posits conclusively performed better than float32. This does leave some to desired. If more intervals than these were tested, it is possible the data would have led us to other conclusions.

The benchmark program developed for this study was implemented using floating-point numbers. The float64 matrix product that the float32 and posit32 matrix products are compared against is treated as if it has no error; an assumption we know is false. However, we expect this rounding error to have minimal impact on our result, as float64 has a 2^{29} times smaller machine epsilon. The float64 matrix product error should be of several magnitudes smaller than our float32 matrix product error.

Another potential pitfall is that the benchmark program sum both the absolute errors and the relative errors into an 80 bit extended float. This could end up inaccurate as addition between two numbers of different sizes is potentially a dangerous operation. The 80 bit extended precision was used in order to compensate for this.

Over all, no numerical analysis to attempt a calculation of a maximum error has been done on the code. Although we believe the error should be small enough to not have a significant impact on the result, we can not guarantee this is the case.

5.3 Future research

Following is a list of potentially interesting subjects to research:

- Implementing both Posit and IEEE 754 floating point in the same type of processor to evaluate execution performance. Critics of the Posit format often claim that the complexity of a varying size regime will negatively affect execution time. This would be an interesting theory to put to the test.
- Implementing a different kind of matrix multiplication algorithm using Posits, such as Strassen's algorithm. To examine the effects of fewer calculations on the error propagation.
- Developing a quire-like intermediate format for IEEE floats and evaluate if it is a worthy tradeoff. A previous study by Koenig tested the feasibility of a hardware register like this, but we believe this is a promising prospect, and more studies would be in order.
- Comparing Posit with 16 bit-length with either float with the same amount of bits or floats with a larger bit length. This could show if posit of a smaller bit length can be a viable choice instead of float of a larger bit length.

Chapter 6

Conclusions

The main conclusion from the study was that for matrix multiplications of elements that are generated in an intervals equal to or larger than $[-0.01; 0.01)$ up to $[-100; 100)$ had higher precision in posit32 than in float when not utilising the intermediate quire format. For almost all tested intervals outside of this range, float32 had better precision than posit32.

When utilising the intermediate data structure quire, the posit32 matrices performed similarly for small matrices, but the precision became better as the matrices grew larger. The 128×128 posit32 matrices utilising quire performed better than the float32 matrices in the intervals $[-10^{-4}; 10^{-4})$, $[-10^{4}; 10^{4})$ and all intervals between.

In their worst case scenario, posits are many times less precise than they are precise in their best case scenario. Because of this, they could be dangerous in the hands of the uninformed user. It is possible to scale the data to make sure it is within the optimal zone for Posit, and therefore getting around the drawbacks for numbers of very large/small magnitude. But that is not always possible, and requires technical know-how from the user. This means that Posit has not shown a compelling reason from a precision point of view to replace IEEE 754 as the main floating-point format.

However, for normalised matrices, Posits excel. The average precision improvement was 18.8, and the median improvement was 15.9 for matrices with elements between -1 and 1.

Bibliography

[1] Gustafson and Yonemoto, "Beating floating point at its own game: Posit arithmetic", *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, pp. 71–86, Jun. 2017, ISSN: 2409-6008. DOI: 10.14529/jsfi170206. [Online]. Available: https://doi.org/10.14529/jsfi170206.

[2] "Ieee standard for binary floating-point arithmetic", eng, *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985.

[3] "Ieee standard for floating-point arithmetic", *IEEE Std 754-2008*, pp. 1–70, 2008.

[4] "Posit standard documentation. release 3.2-draft", Posit Working Group, Singapore, Standard, Jun. 2018. [Online]. Available: https://posithub.org/docs/posit_standard.pdf.

[5] J. L. Gustafson, "Posit arithmetic", *Mathematica Notebook describing the posit number system*, vol. 30, 2017.

[6] M. Martel, "Semantics of roundoff error propagation in finite precision calculations", *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 7–30, 2006, ISSN: 1573-0557. DOI: 10.1007/s10990-006-8608-2. [Online]. Available: https://doi.org/10.1007/s10990-006-8608-2.

[7] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: The good, the bad and the ugly", in *Proceedings of the Conference for Next Generation Arithmetic 2019*, ser. CoNGA'19, Singapore, Singapore: Association for Computing Machinery, 2019, ISBN: 9781450371391. DOI: 10.1145/3316279.3316285. [Online]. Available: https://doi.org/10.1145/3316279.3316285.

[8] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull, "Implementation of strassen's algorithm for matrix multiplication", pp. 32–32, 1996.

[9] 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*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, ISBN: 9781467388153.

[10] S. W. D. Chien, I. B. Peng, and S. Markidis, "Posit npb: Assessing the precision improvement in hpc scientific applications", *Lecture Notes in Computer Science*, pp. 301–310, 2020, ISSN: 1611-3349. DOI: 10.1007/978-3-030-43229-4_26. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-43229-4_26.

[11] "Https://gitlab.com/cerlane/softposit",

[12] "Https://gitlab.com/cerlane/softfloat-python",

[13] "Https://github.com/simonedeblasio/matrix-multiplication-posit",

[14] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications", *arXiv preprint arXiv:1412.7024*, 2014.

[15] J. Koenig, "A hardware accelerator for computing an exact dot product", Master's thesis, EECS Department, University of California, Berkeley, May 2018. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-51.html.

Appendix A

Table of absolute errors

Random interval	Matrix size	1	2	4	8	16	32	64	128
	Float32 Error	8.42E-41	5.9E-40	4.11E-39	2.85E-38	2.04E-37	1.51E-36	1.16E-35	9.02E-35
1E-16	Posit32 with quire Error	2.76E-34	1.55E-33	8.22E-33	4.47E-32	2.47E-31	1.37E-30	7.07E-30	3.43E-29
	Posit32 Error	2.76E-34	1.93E-33	1.34E-32	9.5E-32	6.93E-31	5.26E-30	3.94E-29	2.76E-28
	Float32 Error	9.19E-39	6.21E-38	4.19E-37	2.9E-36	2.06E-35	1.53E-34	1.16E-33	9.05E-33
1E-15	Posit32 with quire Error	1.07E-32	5.72E-32	2.84E-31	1.38E-30	7.04E-30	3.74E-29	2.04E-28	1.13E-27
	Posit32 Error	1.07E-32	7.28E-32	4.9E-31	3.26E-30	2.21E-29	1.57E-28	1.17E-27	8.92E-27
	Float32 Error	8.79E-37	6.39E-36	4.26E-35	2.93E-34	2.08E-33	1.53E-32	1.17E-31	9.06E-31
1E-14	Posit32 with quire Error	2.99E-31	1.77E-30	9.58E-30	5.16E-29	2.54E-28	1.23E-27	6.07E-27	3.12E-26
	Posit32 Error	2.99E-31	2.17E-30	1.57E-29	1.16E-28	8.21E-28	5.65E-27	3.85E-26	2.69E-25
	Float32 Error	8.89E-35	6.06E-34	4.13E-33	2.86E-32	2.04E-31	1.51E-30	1.16E-29	9.03E-29
1E-13	Posit32 with quire Error	9.34E-30	5.04E-29	2.63E-28	1.44E-27	7.88E-27	4.33E-26	2.26E-25	1.1E-24
	Posit32 Error	9.34E-30	6.36E-29	4.36E-28	3.19E-27	2.38E-26	1.83E-25	1.39E-24	9.94E-24
	Float32 Error	8.83E-33	6.31E-32	4.15E-31	2.9E-30	2.06E-29	1.52E-28	1.16E-27	9.05E-27
1E-12	Posit32 with quire Error	3.41E-28	1.84E-27	8.92E-27	4.42E-26	2.23E-25	1.18E-24	6.4E-24	3.55E-23
	Posit32 Error	3.41E-28	2.35E-27	1.56E-26	1.06E-25	7.14E-25	5.1E-24	3.81E-23	2.95E-22
	Float32 Error	8.98E-31	6.21E-30	4.26E-29	2.93E-28	2.08E-27	1.53E-26	1.17E-25	9.06E-25
1E-11	Posit32 with quire Error	9.78E-27	5.47E-26	3.02E-25	1.63E-24	8.12E-24	3.92E-23	1.93E-22	9.86E-22
	Posit32 Error	9.78E-27	6.8E-26	4.97E-25	3.67E-24	2.63E-23	1.81E-22	1.24E-21	8.65E-21
	Float32 Error	9E-29	6.19E-28	4.13E-27	2.87E-26	2.05E-25	1.51E-24	1.16E-23	9.03E-23
1E-10	Posit32 with quire Error	3.07E-25	1.59E-24	8.29E-24	4.51E-23	2.46E-22	1.36E-21	7.15E-21	3.52E-20
	Posit32 Error	3.07E-25	2.04E-24	1.38E-23	9.99E-23	7.51E-22	5.78E-21	4.39E-20	3.18E-19
	Float32 Error	9.48E-27	6.18E-26	4.1E-25	2.87E-24	2.05E-23	1.52E-22	1.16E-21	9.03E-21
1E-09	Posit32 with quire Error	1.13E-23	5.73E-23	2.88E-22	1.4E-21	7.06E-21	3.72E-20	2.01E-19	1.11E-18
	Posit32 Error	1.13E-23	7.42E-23	5.01E-22	3.34E-21	2.27E-20	1.62E-19	1.2E-18	9.26E-18
	Float32 Error	8.67E-25	6.3E-24	4.3E-23	2.94E-22	2.07E-21	1.53E-20	1.17E-19	9.07E-19
1E-08	Posit32 with quire Error	2.89E-22	1.72E-21	9.46E-21	5.13E-20	2.59E-19	1.25E-18	6.13E-18	3.12E-17
	Posit32 Error	2.89E-22	2.14E-21	1.56E-20	1.15E-19	8.31E-19	5.79E-18	3.95E-17	2.75E-16
	Float32 Error	9.07E-23	6.31E-22	4.13E-21	2.9E-20	2.04E-19	1.52E-18	1.16E-17	9.04E-17
1E-07	Posit32 with quire Error	9.12E-21	5E-20	2.61E-19	1.41E-18	7.73E-18	4.28E-17	2.27E-16	1.12E-15
	Posit32 Error	9.12E-21	6.42E-20	4.35E-19	3.16E-18	2.34E-17	1.81E-16	1.38E-15	1.01E-14
	Float32 Error	8.78E-21	6.03E-20	4.12E-19	2.84E-18	2.04E-17	1.51E-16	1.16E-15	9.03E-15
1E-06	Posit32 with quire Error	3.61E-19	1.89E-18	9.29E-18	4.52E-17	2.24E-16	1.17E-15	6.34E-15	3.5E-14
	Posit32 Error	3.61E-19	2.4E-18	1.62E-17	1.07E-16	7.24E-16	5.11E-15	3.78E-14	2.9E-13
	Float32 Error	8.44E-19	6.18E-18	4.27E-17	2.96E-16	2.08E-15	1.53E-14	1.17E-13	9.06E-13
1E-05	Posit32 with quire Error	8.97E-18	5.32E-17	2.96E-16	1.64E-15	8.37E-15	4.05E-14	1.99E-13	1.01E-12
	Posit32 Error	8.97E-18	6.62E-17	4.84E-16	3.64E-15	2.64E-14	1.85E-13	1.26E-12	8.73E-12
	Float32 Error	9.07E-17	6.01E-16	4.19E-15	2.91E-14	2.06E-13	1.52E-12	1.16E-11	9.04E-11
1E-04	Posit32 with quire Error	3.11E-16	1.59E-15	8.58E-15	4.56E-14	2.49E-13	1.39E-12	7.43E-12	3.74E-11
	Posit32 Error	3.11E-16	2.02E-15	1.41E-14	1E-13	7.41E-13	5.68E-12	4.36E-11	3.21E-10
	Float32 Error	8.37E-15	5.72E-14	4.01E-13	2.8E-12	2.03E-11	1.5E-10	1.15E-09	9.01E-09
1E-03	Posit32 with quire Error	1.17E-14	5.88E-14	3.04E-13	1.5E-12	7.57E-12	3.96E-11	2.15E-10	1.19E-09
	Posit32 Error	1.17E-14	7.36E-14	5.13E-13	3.43E-12	2.32E-11	1.62E-10	1.19E-09	9.11E-09
	Float32 Error	9.05E-13	6.36E-12	4.29E-11	2.95E-10	2.09E-09	1.53E-08	1.17E-07	9.07E-07
1E-02	Posit32 with quire Error	3.08E-13	1.82E-12	1.04E-11	5.76E-11	3.05E-10	1.52E-09	7.73E-09	4.06E-08
	Posit32 Error	3.08E-13	2.19E-12	1.58E-11	1.16E-10	8.49E-10	5.93E-09	4.04E-08	2.79E-07
	Float32 Error	8.92E-11	6.03E-10	4.15E-09	2.88E-08	2.06E-07	1.52E-06	1.16E-05	9.05E-05
1E-01	Posit32 with quire Error	1.13E-11	6.39E-11	3.55E-10	1.97E-09	1.11E-08	6.28E-08	3.47E-07	1.85E-06
	Posit32 Error	1.13E-11	7.46E-11	4.94E-10	3.38E-09	2.46E-08	1.84E-07	1.4E-06	1.03E-05
	Float32 Error	8.47E-09	5.75E-08	4.02E-07	2.78E-06	2E-05	0.00015	0.00115	0.008995
1E+00	Posit32 with quire Error	5.27E-10	3.34E-09	2.04E-08	1.18E-07	6.72E-07	3.83E-06	2.17E-05	0.000123
	Posit32 Error	5.27E-10	3.64E-09	2.53E-08	1.74E-07	1.25E-06	9.36E-06	7.19E-05	0.000562
	Float32 Error	8.71E-07	6.33E-06	4.29E-05	0.000292	0.002086	0.015375	0.116741	0.906595
1E+01	Posit32 with quire Error	7.74E-08	4.86E-07	3.06E-06	1.79E-05	0.000115	0.00075	0.004735	0.028511
	Posit32 Error	7.74E-08	5.57E-07	4.19E-06	3.1E-05	0.00026	0.002389	0.022216	0.197776
	Float32 Error	8.6E-05	0.000615	0.00419	0.029002	0.205107	1.526362	11.62299	90.49467
1E+02	Posit32 with quire Error	2.17E-05	0.000154	0.000964	0.005654	0.032412	0.186173	1.10801	7.277543
	Posit32 Error	2.17E-05	0.000184	0.001448	0.01128	0.088069	0.699523	5.730106	51.73045
	Float32 Error	0.008423	0.058803	0.40668	2.8119	20.04577	150.2123	1151.924	8997.98
1E+03	Posit32 with quire Error	0.005324	0.036883	0.255236	1.711123	10.74188	64.67037	377.6007	2163.619
	Posit32 Error	0.005324	0.044751	0.383679	3.475181	30.51793	261.2505	2154.053	17443.63
	Float32 Error	0.902423	6.441855	43.47824	292.5157	2078.753	15361.04	116529.9	906554
1E+04	Posit32 with quire Error	2.016035	12.28108	74.86674	434.6103	2807.748	18968.63	124887.8	774933.3
	Posit32 Error	2.016035	15.35545	121.4523	924.6816	7874.977	73209.96	689755.2	6230610

	Float32 Error	90.81149	621.3334	4233.879	28828.88	205697	1526756	11627969	90536518
1E+05	Posit32 with quire Error	644.5291	4278.61	27079.38	159093.4	921060.3	5317929	31480497	2.08E+08
	Posit32 Error	644.5291	5319.837	44016.11	349369.3	2782050	22185908	1.81E+08	1.61E+09
	Float32 Error	8921.833	59764.88	399793.3	2813942	20098968	1.5E+08	1.15E+09	9E+09
1E+06	Posit32 with quire Error	170060.9	1100338	7525947	50768734	3.24E+08	1.97E+09	1.16E+10	6.65E+10
	Posit32 Error	170060.9	1354985	11842655	1.07E+08	9.57E+08	8.26E+09	6.85E+10	5.57E+11
	Float32 Error	864805	6398273	42501650	2.9E+08	2.08E+09	1.53E+10	1.16E+11	9.07E+11
1E+07	Posit32 with quire Error	62679374	4E+08	2.32E+09	1.33E+10	8.44E+10	5.75E+11	3.83E+12	2.41E+13
	Posit32 Error	62679374	5.05E+08	3.81E+09	2.92E+10	2.45E+11	2.28E+12	2.16E+13	1.97E+14
1E+08	Float32 Error	85203149	6.31E+08	4.27E+09	2.9E+10	2.07E+11	1.53E+12	1.16E+13	9.06E+13
	Posit32 with quire Error	1.99E+10	1.33E+11	8.5E+11	5E+12	2.91E+13	1.68E+14	9.83E+14	6.43E+15
	Posit32 Error	1.99E+10	1.69E+11	1.4E+12	1.12E+13	8.91E+13	7.09E+14	5.74E+15	5.06E+16
	Float32 Error	9.7E+09	5.87E+10	4.07E+11	2.8E+12	2.03E+13	1.5E+14	1.15E+15	9.01E+15
1E+09	Posit32 with quire Error	5.66E+12	3.35E+13	2.35E+14	1.59E+15	1.02E+16	6.25E+16	3.68E+17	2.12E+18
	Posit32 Error	5.66E+12	4.18E+13	3.69E+14	3.35E+15	3.03E+16	2.62E+17	2.19E+18	1.78E+19
	Float32 Error	8.93E+11	6.18E+12	4.27E+13	2.93E+14	2.08E+15	1.53E+16	1.16E+17	9.06E+17
1E+10	Posit32 with quire Error	2.12E+15	1.25E+16	7.47E+16	4.22E+17	2.63E+18	1.79E+19	1.2E+20	7.62E+20
	Posit32 Error	2.12E+15	1.58E+16	1.23E+17	9.43E+17	7.73E+18	7.11E+19	6.75E+20	6.2E+21
	Float32 Error	9.63E+13	6.26E+14	4.26E+15	2.93E+16	2.08E+17	1.53E+18	1.16E+19	9.06E+19
1E+11	Posit32 with quire Error	6.96E+17	4.36E+18	2.68E+19	1.59E+20	9.29E+20	5.36E+21	3.11E+22	2E+23
	Posit32 Error	6.96E+17	5.39E+18	4.42E+19	3.56E+20	2.84E+21	2.27E+22	1.83E+23	1.58E+24
	Float32 Error	8.68E+15	6.22E+16	4.03E+17	2.81E+18	2.03E+19	1.51E+20	1.16E+21	9.02E+21
1E+12	Posit32 with quire Error	1.8E+20	1.12E+21	7.31E+21	4.96E+22	3.25E+23	1.98E+24	1.17E+25	6.78E+25
	Posit32 Error	1.8E+20	1.37E+21	1.16E+22	1.05E+23	9.53E+23	8.29E+24	6.94E+25	5.67E+26
	Float32 Error	9.27E+17	6.19E+18	4.2E+19	2.9E+20	2.07E+21	1.53E+22	1.16E+23	9.06E+23
1E+13	Posit32 with quire Error	6.76E+22	4.03E+23	2.34E+24	1.35E+25	8.28E+25	5.59E+26	3.77E+27	2.41E+28
	Posit32 Error	6.76E+22	5.1E+23	3.83E+24	3.02E+25	2.43E+26	2.2E+27	2.09E+28	1.93E+29
	Float32 Error	8.82E+19	6.36E+20	4.33E+21	2.91E+22	2.08E+23	1.53E+24	1.17E+25	9.06E+25
1E+14	Posit32 with quire Error	1.91E+25	1.39E+26	8.46E+26	5.07E+27	2.96E+28	1.71E+29	9.88E+29	6.25E+30
	Posit32 Error	1.91E+25	1.71E+26	1.39E+27	1.12E+28	9E+28	7.16E+29	5.71E+30	4.8E+31
	Float32 Error	8.75E+21	6.12E+22	4.12E+23	2.82E+24	2.04E+25	1.51E+26	1.16E+27	9.02E+27
1E+15	Posit32 with quire Error	5.58E+27	3.51E+28	2.23E+29	1.52E+30	1.01E+31	6.26E+31	3.72E+32	2.15E+33
	Posit32 Error	5.58E+27	4.38E+28	3.53E+29	3.15E+30	2.87E+31	2.48E+32	2.04E+33	1.63E+34
	Float32 Error	8.84E+23	6.04E+24	4.25E+25	2.86E+26	2.07E+27	1.52E+28	1.16E+29	9.06E+29
1E+16	Posit32 with quire Error	1.99E+30	1.23E+31	7.36E+31	4.2E+32	2.44E+33	1.59E+34	1.09E+35	7.08E+35
	Posit32 Error	1.99E+30	1.52E+31	1.18E+32	8.92E+32	6.78E+33	5.15E+34	4.13E+35	3.26E+36

TRITA-EECS-EX-2020: 345