The Analysis of Using Extended-Precision with Single-Precision Floats for Near Double-Precision Values

Kieran Warwick

MComp (hons) Computer Science
The University of Bath
May 2020

The Analysis of Using Extended-Precision with Single-Precision Floats for Near Double-Precision Values

Kieran Warwick
Department of Computer Science
University of Bath
kw510@bath.ac.uk

ABSTRACT
General Purpose Computation on Graphic Processing Units (GPGPU) is in use for High Performance Computing (HPC), however for the consumer lines of Graphic Processing Units (GPUs) this use case is neglected. This is because the focus of consumer cards is for graphics, not general computing, and as such, there exists a gap in performance between single precision and double precision floating point numbers. Is it possible to close this gap by making a near double precision floating point representation, which can then be used to provide faster double precision floating point numbers at the cost of accuracy? This can be performed through the use of a software technique known as Extended Precision (EP), which uses multiple floating point numbers to represent a higher precision floating point number. A library was built in order to test this, and the results showed that this software technique can produce speed-up, with some restrictions.

Author Keywords
Floating Point; Extended Precision; High Performance Computing; Graphics Processing Units; General Purpose Computation on a GPU; Floating Point Representation; Parallel Computing;

CCS Concepts
•Computing methodologies $\rightarrow$ Parallel computing methodologies; Representation of mathematical objects;

INTRODUCTION
The Graphics Processing Unit (GPU) is normally designed to display graphics, mostly ignoring the use case for General Purpose computation (GP) as it is unnecessary for displaying graphics. GPUs make use of 32-bit single precision floating point (FP32) numbers when performing these graphic operations, with the Fused-Multiply Add (FMAD) operation being one of the most used operations. As FP32 operations are the most used, it is a reasonable assumption that they are optimal in most GPUs, and are faster than 64-bit double precision floating point (FP64) operations. Can these FP32 operations be used to simulate FP64 arithmetic to a reasonable degree of accuracy, while maintaining the performance advantage of using the faster FP32 operations?

GP computation on GPUs (GPGPU) uses the GPU to perform computations that the Central Processing Unit (CPU) was initially designed to perform. The benefit of using a GPU over a CPU is that a GPU can make use of thousands of compute cores, allowing for highly parallel algorithms to be implemented, which in turn can outperform the CPU. Another benefit is that Floating Point (FP) arithmetic tends to be much faster on a GPU. This is because a GPU's main purpose is to process FP numbers rapidly, which means that for FP problems GPUs should be used.

When comparing the speed of FP32 and FP64 FP operations per second (FLOPS) for GPU, there is a noticeable difference for the GeForce line of Nvidia GPUs. For a mobile 20 series GeForce GPU, the processing power in terms of FLOPS of FP32 is exactly 32 times faster than that of FP64. This slug factor exists because the GeForce line is designed for games and FP32 tasks, so more focus is put into providing fast FP32 FLOPS over FP64. The Quadro line of GPUs also suffer from this as they are using the same chips as on GeForce GPUs, but they also provide more FLOPS and ECC memory. The Tesla line of GPUs, however only has a slug factor of 2, as the line is designed for High Performance Computing (HPC), with some of the cards not supporting graphics at all. In theory, if the slug factor can be improved the on the GeForce GPU, then the slug factor would also be improved on the Quadro line of GPUs.

It appears the hardware is limiting FP64 performance in GeForce cards, and therefore the question is, can this be improved through the use of software? Extended Precision (EP) is a software technique that can allow for a near FP64 representation and arithmetic, while using the faster FP32 numbers, which could provide speed-up at a cost of accuracy when compared to the native FP64 arithmetic. If speed-up can be achieved, it would allow for cheaper and more accessible precision operations, reducing the need for a Tesla GPU for fast FP64 performance.

A library of EP operations will be developed in order to test if speed-up can be achieved on the basic operations (+,-,$\div$,$\times$)

and how they differ from the native FP64 operations on the GPU. How EP operations perform in real world algorithms will also be developed by using these basic operations. Estimating π using Leibniz's formula, and comparing the results between each output of the FP64 and EP operations to verify that real algorithms, such as this, could be implemented using EP operations.

BACKGROUND

In this section different types of the software technique EP will be explored, along with the error-free operations required to perform the chosen EP arithmetic.

Extended Precision

EP is a software technique used to extend the available precision of a FP number through the use of lower precision FP numbers [4]. To represent an EP number, there exists two common approaches:

- **Multi-term:** Where an EP number stores a sequence of FP numbers, such that the unevaluated sum of the FP numbers represents a higher precision FP number.

- **Multi-digit:** Where an EP number stores a sequence of FP numbers and a single exponent, such that the unevaluated sum of the products of the FP numbers and the exponent represents a higher precision FP number.

Multi-term is the faster of the two, as it does not use a single exponent but instead uses the exponents of the FP terms. This restricts the range a multi-term EP number can represent, as the exponent is limited to the lower FP numbers' precision. The multi-digit representation does not have this problem, as it contains a single exponent that is used as a shared exponent to all the FP terms, which allows for increasing the range of the exponent but at a cost of increased computation [9].

For the representation of a near FP64, a two-term EP number will be used, using FP32 numbers for the terms of the EP number. This representation was chosen as it should provide the opportunity to achieve speed up when compared to a native FP64 number, and will provide a 49-bit significand, which is only 5 bits less than the effective significand in a FP64 number. The near FP64 number, however, will be restricted by the FP32 exponent, which reduces what the near FP64 number is able to represent.

This near FP64 will be defined as a new data-type within the library and will be referred to as an EP number throughout this paper. This EP number will be stored as a pair of FP32 numbers, (s, e), such that s is the closest approximation using native FP32 operations, and e is an error correcting term which negates the errors of the approximation. This means that the inequality

$$|s| \geq |e|,$$

must hold true for all EP numbers.

To implement algorithms for this new data-type, existing EP libraries and literature will be examined for the best algorithms in terms of performance and correctness.

Existing Libraries

The software technique EP is not new and as such, many libraries have implemented both multi-term and multi-digit EP number arithmetic, for both the CPU and GPU. The CPU libraries include the following:

- DoubleFloats.jl: Implemented in Julia, providing a multi-term double-float structure for FP64, FP32, and half precision FP types, with algebraic and transcendental functions [12].

- MPFUN2015: Provides multi-digit arbitrary EP computation for algebraic and transcendental functions, using C++ and Fortran-90 [1].

- QD: Provides multi-term "double-double" and "quad-double" arithmetic, with algebraic and transcendental functions, using C++ and Fortran-90 [2].

- ARPREC: An older version of MPFUN2015 which is not thread safe. It provides multi-digit arbitrary EP computation using 32 bit words [3].

Some of the CPU libraries above have been rewritten to new libraries which run on a GPU, such as GQD and GARPEC [9], which are libraries that have implemented QD and ARPEC for the GPU. CUMP [10] is also a GPU library that is based on the CPU library GMP [5], it was designed to be faster than the GARPREC library, through the use of 64 bit words and by storing the least significant bits in the word first.

Thall also created a library within his paper [13], which used the algorithms from Hila et. al [6] to develop a GPU library in Cg for both double-float and quad-float arithmetic. CAMPARY [8] was then developed to provide arbitrary precision using multi-term floating points, and could be considered an extension of Thall's library.

All of these libraries compare themselves to other EP libraries but they do not, however, measure the performance of their EP numbers to the native FP64 upon the GPU. But some do measure the correctness of the algorithms that were implemented.

Using this existing work, a new EP library will be developed with a focus on providing a two-term float arithmetic for the purpose of near-FP64 precision calculations. In order to do this, algorithms that were both fast and correct were implemented from these libraries.

CONSTRUCTED EXTENDED PRECISION LIBRARY

Using the literature and existing libraries, a GPU EP library was created. This was performed by first verifying that all functions needed for EP work on a CPU. If they do, then the functions will be transferred onto the GPU and verified that they calculate the same results as the CPU.

In the EP library, the error-free and arithmetic functions will only be accessible by the GPU, while the representation and conversion are accessible by both CPU and GPU. This section will show the functions that were implemented within the library.

Representation and Conversion

To convert a FP64 number into the new EP data-type, it requires that the FP64 number is split as accurately as possible. The approach taken to perform this is to first convert the FP64 number into its FP32 representation. Then using the difference between the FP32 representation and the FP64 number as the error term. Algorithm 1 demonstrates this.

Algorithm 1 . A FP64 number a can be converted into an EP number with two FP32 numbers, s and e, such that $|s| \geq |e|$, where

```
1: procedure FP64ToEP($a, b$)
2:     $s = \text{Float}(a)$
3:     $e = \text{Float}(a - s)$
4:     return $(s, e)$
5: end procedure
```

Then to convert back into a FP64 representation, the FP32 terms must be converted into FP64 and then summed together, which results in the FP64 representation of the EP number.

Error-Free FP32 Operations

When a FP32 operation is performed between two FP32 floats, there exists an error less than the precision of the FP32 numbers. This error can be negated through the use of an error correction term, which can make FP32 operations 'error-free' [7]. This requires a new set of operations which performs FP32 arithmetic and then calculates the error term.

Hila et al. [6] highlight algorithms which can be used to calculate error free FP32 operations for addition and multiplication, for which both return an EP number. Let $\oplus, \ominus, \otimes, \oslash$ be their respective FP32 operations and let $+, -, \times, \div$ be accurate real operations, then algorithms 2, 3 and 4 are considered error-free as they return an EP number, which contains the error correcting term. These algorithms can then be used as the basic building blocks of EP arithmetic, and were implemented within CUDA.

Algorithm 2 . [6, 11] To sum two FP32 numbers a and b error-free, first compute the first term $s = a \oplus b$ then calculate the error term $e = err(a \oplus b)$. This algorithm assumes that $|a| \geq |b|$

```
1: procedure QuickTwoSum($a, b$)
2:     $s = a \oplus b$
3:     $e = b \ominus (s \ominus a)$
4:     return $(s, e)$
5: end procedure
```

An additional error-free operation will also be used, for the difference between FP32 numbers. This is a modification of the two-sum algorithm, and can be seen in algorithm 5.

Extended Precision Arithmetic

Now that FP32 operations can be considered error-free, it is possible to implement basic EP arithmetic using the algorithms from literature and existing libraries.

Algorithm 3 . [6, 11] To sum two FP32 numbers a and b error-free, first compute the first term $s = a \oplus b$ then calculate the error term $e = err(a \oplus b)$. This algorithm does not assume that $|a| \geq |b|$

```
1: procedure TwoSum($a, b$)
2:     $s \leftarrow a \oplus b$
3:     $v \leftarrow s \ominus a$
4:     $e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
5:     return $(s, e)$
6: end procedure
```

Algorithm 4 . [6, 11] For two floating point numbers, a and b, $a \times b = (s, e)$ where,

```
1: procedure TwoProdFMA($a, b$)
2:     $s = a \otimes b$
3:     $e = \text{FMA}(a, b, -s)$
4:     return $(s, e)$
5: end procedure
```

Algorithm 5 . [6, 11] To find the difference between two FP32 numbers a and b error-free, first compute the first term $s = a \ominus b$ then calculate the error term $e = err(a \ominus b)$. This algorithm does not assume that $|a| \geq |b|$

```
1: procedure TwoDiff($a, b$)
2:     $s \leftarrow a \ominus b$
3:     $v \leftarrow s \ominus a$
4:     $e \leftarrow (a \ominus (s \ominus v)) \ominus (b \oplus v)$
5:     return $(s, e)$
6: end procedure
```

Addition

To add two EP numbers, the library will use the function 'IEEE addition' used in the QD library [2] as it is recommended as it performs addition very accurately [7]. However, it is about two times slower than the other addition function 'sloppy addition' found within the QD library, which requires that both operands have the same sign.

Algorithm 6 . IEEE addition of two EP numbers as implemented in QD

```
1: procedure ADDITION($(a_{hi}, a_{lo}), (b_{hi}, b_{lo})$)
2:     $(s_{hi}, s_{lo}) \leftarrow$ TWOSUM($a_{hi}, b_{hi}$)
3:     $(t_{hi}, t_{lo}) \leftarrow$ TWOSUM($a_{lo}, b_{lo}$)
4:     $s_{lo} \leftarrow s_{lo} \oplus t_{hi}$
5:     $(s_{hi}, s_{lo}) \leftarrow$ QUICKTWOSUM($s_{hi}, s_{lo}$)
6:     $s_{lo} \leftarrow s_{lo} \oplus t_{lo}$
7:     $(s_{hi}, s_{lo}) \leftarrow$ QUICKTWOSUM($s_{hi}, s_{lo}$)
8:     return $(s_{hi}, s_{lo})$
9: end procedure
```

Subtraction

To subtract an EP number *b* from another EP number *a*, it can be as simple as inverting the sign of the terms of *b* and performing addition. For this library, the same structure of algorithm 6 was used, and used TwoDiff instead of TwoSum. This can be decomposed to the same operations as inverting the sign of *b* and adding, however the algorithm 7 is used as it makes it clear how subtraction works.

Algorithm 7 . IEEE subtraction of two EP numbers as implemented in QD

```
1: procedure SUBTRACTION($(a_{hi}, a_{lo}), (b_{hi}, b_{lo})$)
2:     $(s_{hi}, s_{lo}) \leftarrow$ TWODIFF($a_{hi}, b_{hi}$)
3:     $(t_{hi}, t_{lo}) \leftarrow$ TWODIFF($a_{lo}, b_{lo}$)
4:     $s_{lo} \leftarrow s_{lo} \oplus t_{hi}$
5:     $(s_{hi}, s_{lo}) \leftarrow$ QUICKTWOSUM($s_{hi}, s_{lo}$)
6:     $s_{lo} \leftarrow s_{lo} \oplus t_{lo}$
7:     $(s_{hi}, s_{lo}) \leftarrow$ QUICKTWOSUM($s_{hi}, s_{lo}$)
8:     return $(s_{hi}, s_{lo})$
9: end procedure
```

Multiplication

For the multiplication between two EP numbers, the algorithm 'DWTimesDW3' was chosen to be implemented from the paper by Joldes, Muller, and Popescu [7]. This algorithm was also implemented within doubleFloats.jl [12] and works effectively within Julia. This algorithm was then modified to run on CUDA.

While implementing the algorithm 'DWTimesDW3', it was noted that a FLOP could be saved from the algorithm! It is possible to include an extra FMA operation and as such, saves a FLOP from the original. This also has the bonus benefit of increasing the accuracy of the algorithm, as removing a FLOP can possibly remove a potential rounding error. The result was Algorithm 8, and was thus implemented within CUDA.

Algorithm 8 . Multiplication of two EP numbers

```
1: procedure MULTIPLICATION($(a_{hi}, a_{lo}), (b_{hi}, b_{lo})$)
2:     $(s_{hi}, s_{lo}) \leftarrow$ TWOPRODFMA($a_{hi}, b_{hi}$)
3:     $s_{lo} \leftarrow$ FMA($a_{lo}, b_{lo}, s_{lo}$)
4:     $s_{lo} \leftarrow$ FMA($a_{hi}, b_{lo}, s_{lo}$)
5:     $s_{lo} \leftarrow$ FMA($a_{lo}, b_{hi}, s_{lo}$)
6:     $(s_{hi}, s_{lo}) \leftarrow$ QUICKTWOSUM($s_{hi}, s_{lo}$)
7:     return $(s_{hi}, s_{lo})$
8: end procedure
```

Division

For division, the algorithm 'dvi_dddd_dd_fast' from DoubleFloats.jl [12] was chosen, which can be seen to be a slight modification to the 'DWDivDW2' algorithm within the paper by Joldes, Muller, and Popescu [7].

From the original algorithm, an FMA was added to the DoubleFloats implementation, which saves a FLOP in total and should also reduce errors. Algorithm 9 shows the algorithm as it is implemented within the constructed library.

Algorithm 9 . Division of two EP numbers

```
1: procedure DIVISION($(a_{hi}, a_{lo}), (b_{hi}, b_{lo})$)
2:     $s_{hi} \leftarrow a_{hi} \oslash b_{hi}$
3:     $t_{hi}, t_{lo} \leftarrow$ TWOPRODFMA($s_{hi}, b_{hi}$)
4:     $v \leftarrow (a_{hi} \ominus t_{hi}) \oplus (a_{lo} \ominus t_{lo})$
5:     $s_{lo} \leftarrow$ FMA($-hi, b_{lo}, v$) $\oslash b_{hi}$
6:     $(s_{hi}, s_{lo}) \leftarrow$ QUICKTWOSUM($s_{hi}, s_{lo}$)
7:     return $(s_{hi}, s_{lo})$
8: end procedure
```

Mixed Precision

Algorithms were also implemented for mixed precision operations between an EP number and a float, however they are not tested in this paper, thus considered further work that could be performed.

TESTING AND RESULTS

To investigate if this software technique is indeed useful, the basic operations will be tested using 1000 blocks with 1024 threads, each performing one operation per thread, upon various Nvidia GPUs from GeForce Mobile to HPC Tesla. For the tests, the maximum number of threads per block for the GPUs were tested, and the number of blocks will ensure that the dataset to work on is substantial enough to be able to measure speed-up effectively.

To perform the testing, two one-dimensional vectors of size 1000*1024 (1.024 million) are required for the input datasets, which were allocated random FP64 values from -1,000,000 to 1,000,000 by using a uniform real distribution. When testing is performed, the dataset was converted into the correct datatype for the operation, which ensured that mixed precision operations were not performed.

For the mobile and desktop GPUs, the test suite was compiled using both CUDA run-times 10.2 and 9.0 within Windows, and distributed to trusted friends to run on their Windows device.

	$+$	$-$	$\times$	$\div$
Mean	27.1	31.4	7.847	10.29
Median	4	4	6	7
Max	2359296	3145728	81	208
Min	0	0	0	0
sd	2711	4223	7.344	10.880

Table 1. Correctness testing summary in terms of ULP

They were told to first update their drivers to be able to run the CUDA 10.2 runtime, except in one case where driver support for the GPU had ended. They were then given the respective executable, which then was run with laptops plugged in, and on high performance. They were then asked to send the results back, and it is trusted that these results were not tampered with.

For the Tesla GPUs, the test suite was compiled on the balena HPC cluster using the CUDA 9.0 compiler for Linux, and then run on the cluster for certain Tesla GPUs.

Correctness Testing
When tested, each operation used the same two datasets for the parameters of the operation. For correctness testing, first the calculation was performed upon the CPU using a FP64 operation. This was done to ensure that a vector of control values were created into order to perform comparisons to the results from the GPU. Once this control vector was created, it was possible to perform correctness testing.

For each operation performed on the GPU, it returned a vector of the computed results in the data-type of the operation. The computed results were then converted into FP64 numbers to produce a vector that was then compared to the control vector. This was necessary as to measure the correctness of the computed results, the spacing between two FP64 numbers needed to be calculated. This was done by using the difference in the ULP (Unit in the Last Place) for each term of the converted results and the control vectors.

This produced a correctness vector, comprised of ULP distances. This was analysed and determined if the operation was to be considered correct. A summary of the correctness vectors can be seen in table 1, which shows information about the distribution of the correctness vector for EP operations. For the FP64 operations on the GPU, the ULP was zero in all cases, while FP32 operations have a mean ULP in the hundred of millions.

Performance Testing
The performance of the EP operations was measured on each GPU by calculating the speed-up of the EP operation when compared to a FP64 operation. In order to do this, the time taken for the GPU to compute the results for both FP64 and EP operations was measured. To ensure that these times were reliable, the operation was repeated 100,000 times, with the timer starting after the memory was transferred to the GPU. The timer started here because the test was measuring the performance of the operation, not how fast the data transferred to the card.

The speed-up of the EP operation when compared to a FP64 can then be calculated by the following formula:

$$\text{Speed-up} = \frac{\text{Total time taken for FP64}}{\text{Total time taken for EP}},$$

where the total time was the time taken for the operation to run 100,000 times.

This speed-up was calculated for all basic EP operations, and the results of the testing can be seen in table 2. This also includes the slug factor of the GPUs, which is how many times faster FP32 operations are compared to FP64 operations.

Real-world Application Testing
A test was also performed to investigate how speed-up and correctness performs in the case of a more typical operation. The typical operation chosen for this test was the estimation of π, which used Leibniz's formula to estimate, which states that the infinite series:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Using this formula, π can thus be estimated using n terms by summing each term and multiplying by 4. This was implemented in CUDA through the use of an accumulator, summing each term of the series with the accumulator on each iteration, then returning the value multiplied by 4.

The test was performed on a single thread with a single block to calculate 2000 terms of the series. The time taken to perform this operation 100,000 times was recorded, and was done for both EP and FP64 numbers. The results for the performance of each card tested can be seen in table 3, and for each card, it returned the same result for both values, with the EP number having an ULP error of 5 from the FP64 estimation.

ANALYSIS

Addition and Subtraction

Correctness
The correctness for both EP addition and subtraction operations have similar results, which is expected from how EP subtraction works. The graphs of the distribution of the correctness vectors can be seen in figures 1 and 2, and clearly show that there are some errors coming up. These errors occurred when the value was subtracted by a number very close to the value itself.

GPU	+	−	×	÷	Slug Factor
GeForce GT 750M (Mobile)	0.62209	0.59221	0.59800	0.80469	24
GeForce GTX 960M (Mobile)	1.00247	1.00334	1.00256	1.02301	32
GeForce GTX 1050 Ti (Mobile)	1.00216	1.00214	1.00196	1.01108	32
GeForce GTX 1070 (Desktop)	1.00772	1.00779	1.00753	1.01393	32
GeForce RTX 2060 (Mobile)	0.99981	0.99334	0.99680	1.20577	32
GeForce RTX 2060 (Desktop)	0.99712	1.00435	0.99584	1.03800	32
Tesla K20x (Linux)	0.93858	0.93662	0.94728	0.88510	3
Tesla P100 (Linux)	0.87725	0.87734	0.87427	0.87526	2

Table 2. The performance of each EP operation upon GPUs in terms of speed-up against FP64 operations

GPU	Speed-up
GeForce GT 750M (Mobile)	0.87928
GeForce GTX 960M (Mobile)	1.25012
GeForce GTX 1050 Ti (Mobile)	1.02547
GeForce GTX 1070 (Desktop)	0.96786
GeForce RTX 2060 (Mobile)	1.12984
GeForce RTX 2060 (Desktop)	1.13049
Tesla K20x (Linux)	0.27422
Tesla P100 (Linux)	0.24488

Table 3. The speed-up achieved for the calculation of π with 2000 terms

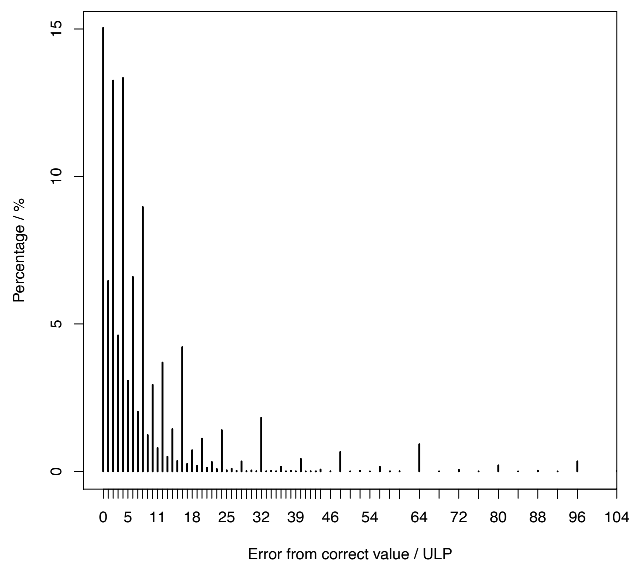

Figure 1. Distribution of the correctness vector for EP addition

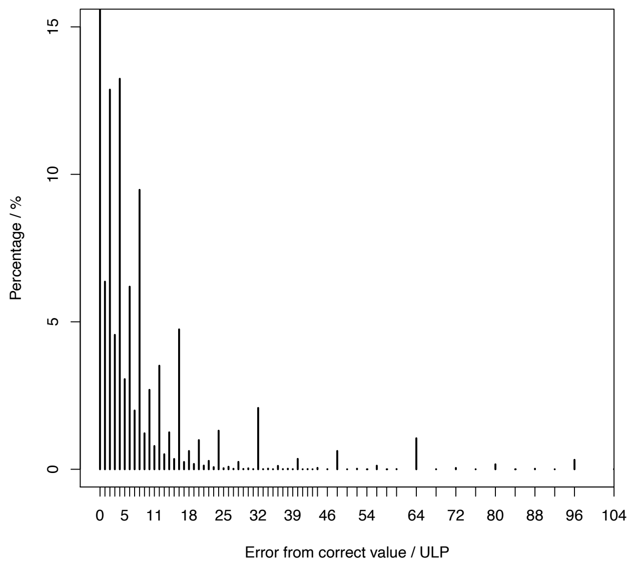

Figure 2. Distribution of the correctness vector for EP subtraction

This implies that the algorithm as implemented either contains an error, or the algorithm is not effective in this specific case.

The mean is high for these operations, however due to errors occurring within the algorithm which caused a maximum value error of 3,145,728, thus skewing the mean. Otherwise, the correctness of the algorithms were acceptable in most cases, with 75% of all errors being 10 ULP, or under for both operations.

Performance

For the performance of the algorithms, speed-up was achieved on the GeForce GTX GPUs, which tend to have a 32 slug factor. For the GeForce RTX GPUs tested, only the desktop GeForce 2070 RTX GPU produced speed-up for subtraction, and no other speed-up was achieved. As subtraction uses a similar algorithm to addition, this provides conflicting results for this GPU. Perhaps the FP64 operation for subtraction is more expensive on this GPU, which would result in the conflict.

The GeForce GT GPU tested performed the worst. This was the oldest card tested, and as such, its slug factor was 24. This

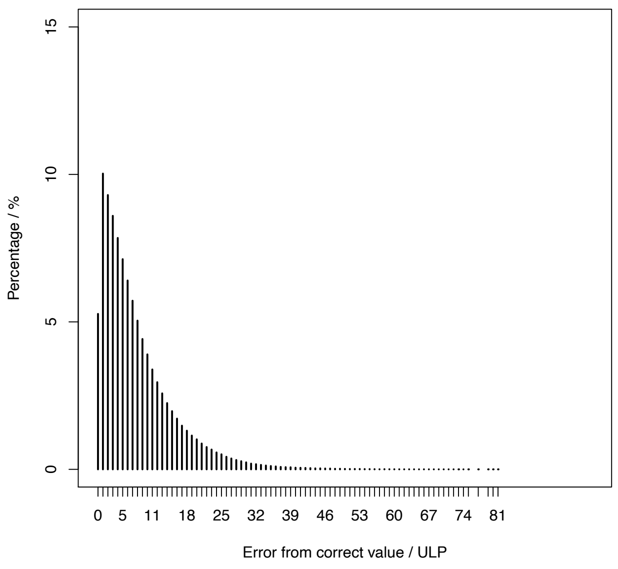

Figure 3. Distribution of the correctness vector for EP multiplication

means speed-up was not expected on this GPU, as it provides more FLOP performance for FP64 than expected, which is unlike the GTX or RTX GPUs, which have a slug factor of 32. This GT GPU also ran the 9.0 version of the executable to run the tests, however it is likely that it did not affect performance.

The Tesla HPC GPU cards did not provide speed-up for either operation. This was due to the fact the cards have been designed in hardware to perform very well at FP64 operations, which is evident in the slug factor being a single digit.

To conclude, addition and subtraction provide a small speed-up on GTX cards only, with the correctness of the EP operations providing acceptable correctness for the representation of a near FP64 number. The only restriction, however is to not subtract a value close to itself, as this will result in a less precise EP number.

Multiplication

Correctness

The correctness for EP multiplication can be argued as more reliable than the operations for addition and subtraction, as they have a lower overall mean. This can be attributed to the fact that for EP multiplication, the maximum recorded ULP error was 81, which was the lowest for all EP operations. The graph of the correctness vector distribution can be seen in figure 3.

The figure shows that multiplication however is slightly worse than addition and subtraction, with 75% of all operations being 11 ULP or under. With there being a less steep curve down compared to addition and subtraction in the figure, it can be determined to be less accurate than these operations. However, for the purpose of near FP64 arithmetic, this accuracy is acceptable.

Performance

Speed-up was achieved upon the GeForce GTX GPUs, with the desktop GeForce GTX 1070 card performing the best. For the RTX cards however, speed-up was not achieved, which is much like the performance for EP addition and subtraction operations.

The GeForce GT GPU once again performed the worst due to the previous reasons explained for addition and subtraction. The Tesla HPC GPU cards did not provide speed-up for either operation, performing just about the same as for the addition and subtraction operations. In conclusion, the multiplication operation performed much the same as the addition and subtraction operations, but produced less accuracy but more reliable results.

Division

Correctness

The correctness for EP division can also be argued as more reliable than the operations for addition and subtraction, but not as reliable as the EP multiplication operation as the maximum recorded ULP error is 208.

Comparing figure 3 to figure 4, it is possible to see that division has a wider distribution, with 75% of operations being 14 ULP or under. This shows that division is less accurate than multiplication, and thus also less accurate than addition and subtraction.

These errors should be noted, as it has the worst correctness of the all basic operations.

Performance

Speed-up was achieved in both GeForce GTX and GeForce RTX GPUs, with the GeForce RTX GPUs performing the best. The mobile GeForce RTX 2070 achieved the best speed-up of 1.20577, which was much higher than the desktop counterpart, only achieving 1.03800. This could however, be due to hardware optimisation within the CUDA compiler, as the mobile GeForce RTX 2070 was on the machine which compiled the executable.

The GeForce GT GPU also showed faster results for this operation, however, it could not provide speed-up as the 24 slug factor was too large to overcome.

The Tesla HPC GPUs performed about the same as all of the other operations. This suggests that for consumer cards, FP64 division is slower than all other operations and can be sped up through the use of EP.

Real-World Calculations

Correctness

The correctness of this calculation demonstrates that EP can be used to find a close estimate to the FP64 estimate for π while using same the number of iterations. As the ULP for 2000 interactions was 5, it essentially produced a very slightly inaccurate result while using the same algorithm as FP64. This supports the use of EP numbers in more applications, where slight inaccuracies do not effect the result.

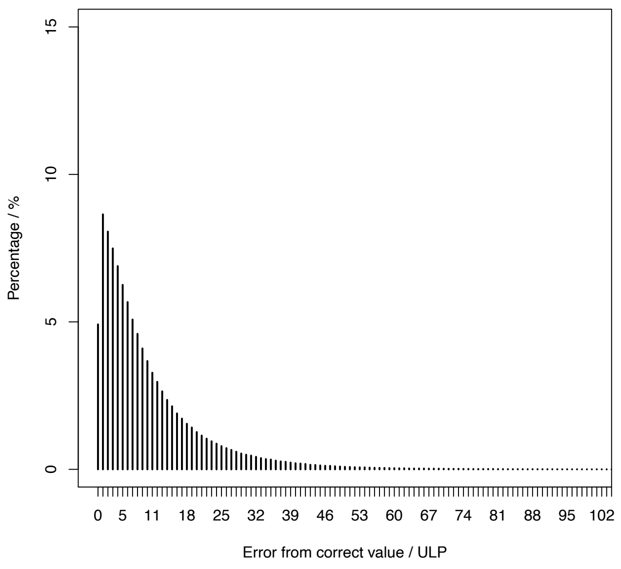

Figure 4. Distribution of the correctness vector for EP division

Performance
For the real-world calculation, the estimation of π was used as a best case scenario, as division saw speed-up on both GeForce GTX and GeForce RTX GPUs. This is because the Leibniz formula uses a ratio of 2 divisions and 1 add and 1 subtract for 2 iterations. Thus a GPU which is good at these EP operations should perform a lot better than GPUs that are not. While this statement is true, there were exceptions, as seen in table 3, the desktop GeForce GTX 1070 does not achieve speed-up, however achieved speed-up for the operations used within the estimation of π.

This could be explained by the algorithm implemented to estimate pi, which converts each integer term to the relevant data-type before performing division in each iteration. This could have perhaps induced unseen overhead. To improve this algorithm, the conversion to the data-type should be removed, and should not convert between data-types during the computation.

All other GeForce GTX and GeForce RTX GPUs did, however, obtain speed-up for this computation, with the mobile GeForce GTX 960M performing exceptionally well, which could be attributed to having fast EP operations for addition, subtraction and division. The GeForce RTX GPUs, which have the best EP division and bad EP addition and subtraction still performed well, achieving speed-up for the calculation of π.

The GeForce GT and Tesla HPC GPUs did not see such speed-up, with the Tesla GPUs performance being 4 times slower of that of the FP64 operations. This is due once again to the fact these cards have a slug factor smaller than 32, and as such, EP operations do not provide speed-up at all to these GPUs.

GPU	EP	FP64
GeForce GT 750M (Mobile)	27.295	24
GeForce GTX 960M (Mobile)	25.598	32
GeForce GTX 1050 Ti (Mobile)	31.205	32
GeForce GTX 1070 (Desktop)	33.063	32
GeForce RTX 2060 (Mobile)	28.323	32
GeForce RTX 2060 (Desktop)	28.306	32
Tesla K20x (Linux)	10.940	3
Tesla P100 (Linux)	8.1672	2

Table 4. The effective slug factor for the calculation of π with 2000 terms

The summary of this analysis for this test can be seen in Table 4. It shows the effective slug rate against FP32 operations for this calculation assuming that the slug factor for FP64 remains at its specified performance.

CONCLUSION

The Speed-up v.s. Precision Trade-off
With the small increase in speed for basic operations, EP operations could provide notable increases in performance of many real algorithms. The trade off being however, that the result would be very slightly wrong, and in some edge cases, extremely wrong. If the precision matters more than the speed of the operations, one should still use FP64 operations. However, if a little bit of error does not matter, and the numbers are within the FP32 exponent range, EP operations are recommended. They are only recommended, however, if the hardware can provide speed-up for these basic operations, i.e. if the slug factor is 32 for FP64 operations compared to FP32. The constructed library provided will leave in the code used to test the operations, such that users of this library can test if EP operations will be suitable.

The answer to the initial question, 'Can these FP32 operations be used to simulate FP64 arithmetic to a reasonable degree of accuracy, while maintaining the performance advantage of using the faster FP32 operations?', is a yes, but with exceptions. These exceptions are:

- Operations are limited to the range of an FP32 exponent.
- There can be cases where the error is extreme, as can be seen in the addition and subtraction operations.
- The programmer should not expect perfect accuracy with this library, as this assumption may lead to incorrect results.
- That the basic EP operations do indeed provide a speed-up on the GPU that the programmer is deploying the application to.
- Conversion to the EP data-type should be minimised, as it could induce overhead within the application, which would lead to slowdown.

Two-Term EP Viability as a Near FP64 Number
The two-term EP number representation for a near FP64 number performed surprising well in terms of accuracy, with all operations achieving a reasonable level of accuracy with most

of the errors in the computations being under 100 ULP for all operations. For the case of a two-term EP representing a near FP64 number, it can be deemed acceptable, if the exceptions as described above are followed. With this result, a near FP64 number could provide the programmer with a fast FP64 number, where the programmer is aware that it does not have the exact accuracy of the native FP64 number, but will perform faster if the GPU the application to be deployed to, has a slug factor of 32.

Achievements and Reflection

The constructed library provides a fast EP division operation, with little accuracy drawbacks with a mean ULP of 10.29. This means that for division heavy calculations, EP operations could be used as a replacement to FP64 operations, as the benefit the speed-up provides outweighs the drawbacks accuracy would bring. This depends if the overall algorithm needs either exact or approximate precision.

Speed-up over FP64 with EP operations was achieved on some of the GeForce GTX line of GPUs, with most of them performing well in the real-world application test. More testing should be performed on the Quadro line of GPU, as these GPUs were not tested upon during this paper. If the Quadro GPUs do use the same chips of the GeForce line however, then it can be expected the speed-up could be achieved on these GPUs.

The error-free algorithms as implemented in this library are naturally sequential, as error correcting terms relies upon calculating the error of the first term being computed. This is a trend in all the algorithms that exist within this library, so this exposes an area of research: can error-free algorithms be developed in such a way that that they support parallelism?. If this could be achieved, then this would provide a substantial speed-up to the EP software technique.

This paper provides a library of some of the basic operations for EP arithmetic and will be released alongside this paper, allowing for programmers to test if their FP64 applications can be sped up through the use of EP. This would only require the programmer to change the data-type within their application to the two-term EP number, and then convert back to FP64 at the end of the calculations.

Future work

To improve on this library, more functions should be implemented to provide the programmer with a range of the essential operations needed for GPGPU. These should be both algebraic and transcendental functions.

The performance of mixed precision operations should be tested and compared. That is, can EP numbers perform operations with FP32 numbers such that speed-up can be obtained? And then how does this effect the correctness of the result when compared to a native mixed precision operation between a FP64 and FP32 numbers?

As this library demonstrates the capability of EP multi-term representation, further work should be performed in developing a library which uses the multi-digit form instead. If the multi-digit form does provide speed-up, then the full range of a FP64 number could be represented, allowing for EP operations in more applications.

If the terms of the EP multi-term representation could be calculated independently, this would allow for fast arbitrary precision, by calculating each term of the multi-term using the parallelism of the GPU. However, there is no known methods to perform these calculations. If this problem is solved then these calculations would not only provide speed-up for the library discussed in this paper, but for all EP libraries that have been developed.

REFERENCES

[1] David H Bailey. 2016. *A thread-safe arbitrary precision computation package (full documentation)*. Technical Report. Tech. rep., Mar. 2017. http://www. davidhbailey. com/dhbsoftware.

[2] David H. Bailey, Xiaoye S. Li, and Yozo Hida. 2003. QD : A Double-Double/ Quad-Double Package, Version 00. (6 2003). https://www.davidhbailey.com/dhbsoftware/

[3] David H. Bailey, Hida Yozo, Xiaoye S. Li, and Brandon Thompson. 2002. ARPREC: An arbitrary precision computation package. (9 2002). DOI: http://dx.doi.org/10.2172/817634

[4] T. J. Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (1971), 224–242. DOI: http://dx.doi.org/10.1007/BF01397083

[5] Torbjörn Granlund. 2004. GNU MP: The GNU multiple precision arithmetic library. *http://gmplib. org/* (2004).

[6] Y. Hida, X. S. Li, and D. H. Bailey. 2001. Algorithms for quad-double precision floating point arithmetic. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*. 155–162. DOI: http://dx.doi.org/10.1109/ARITH.2001.930115

[7] Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. 2017. Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. *ACM Trans. Math. Softw.* 44, 2, Article Article 15res (Oct. 2017), 27 pages. DOI: http://dx.doi.org/10.1145/3121432

[8] Mioara Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. 2016. CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications. In *Mathematical Software – ICMS 2016*, Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese (Eds.). Springer International Publishing, Cham, 232–240.

[9] Mian Lu, Bingsheng He, and Qiong Luo. 2010. Supporting Extended Precision on Graphics Processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN '10)*. Association for Computing Machinery, New York, NY, USA, 19–26. DOI: http://dx.doi.org/10.1145/1869389.1869392

[10] Takatoshi Nakayama and Daisuke Takahashi. 2011. Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing. (12 2011). DOI: http://dx.doi.org/10.2316/P.2011.757-041

[11] Jonathan Richard Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry* 18, 3 (1997), 305–363. DOI: http://dx.doi.org/10.1007/PL00009321

[12] Jeffrey Sarnoff. 2020. DoubleFloats.jl. https://github.com/JuliaMath/DoubleFloats.jl/. (2020).

[13] A. Thall. 2006. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH 2006 Research posters*. ACM, 52.