School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Parallel Arbitrary Precision Floating-Point Arithmetic with AVX Compiler Intrinsics

Rejus Bulevicius

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2023/2024

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (08/05/24)
Link to online code repository	URL	Sent to supervisor and assessor (08/05/24)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) [illegible]

Summary

The arbitrary precision library GNU MPFR does not currently utilise a parallel approach to its arithmetic calculations. Advanced Vector Extensions (AVX) are a set of parallel compiler intrinsics that can be used both within MPFR and as a basis for its own arbitrary precision library, to increase performance in simple arithmetic.

This project has demonstrated that with careful consideration of code, it is possible to emulate an arbitrary precision library such as MPFR, with a positive comparison in speed.

The methods used here can be applied to MPFR itself in future additions or to an independent arbitrary precision library that wishes to utilise the potential of the AVX instruction set.

Acknowledgements

I would like to thank Mantas Mikaitis, my supervisor, for the consistent meetings and help with both understanding the nature of such a project and the concepts used within the report.

Additionally, I want to make clear my appreciation to Nishant Ravikumar for the mid-term feedback as my assessor.

Finally, I want to extend my gratitude to Massimiliano Fasi for the original project proposal and with the understanding of AVX intrinsics.

Contents

Chapter 1

Introduction and surrounding research

1.1 Introduction

The IEEE754 standard is the de facto standard that many microprocessors adhere to when dealing with floating point number arithmetic, this means that often a regular computer may only support three distinct floating point number formats, those being the single, double and double-extended [1]. Even among these cases, the double-extended format is optional, with its implementation left to the microprocessor designer, with singles specified as binary32, or 32 bits in total and doubles specified as binary64, or 64 bits, meaning that these computers don't have native support to operate on numbers that require even more bits to be represented exactly.

Elements of a floating point have the form $m \times \beta^{e-p+1}$, with $0 \leq m \leq \beta^p - 1$ (an integer significand or mantissa), and $e_1 \leq e \leq e_2$ (an integer exponent) [2] where:

- β represents the base of the number, in our case 2
- A precision $p \geq 2$
- Extremal exponents e_{min} and e_{max}
- Usually $e_{min} < e_{max}$ and $e_{min} = 1 - e_{max}$

The binary64/32 formats can be represented (*Figure 1.1*) as the sum of its bits, with both formats having a single sign bit and then 11 or 8 bits for a biased exponent and finally 52 or 23 bits for the mantissa/significand, which is the feature our arbitrary precision focuses on expanding. Since the mantissa controls the precision, and is in itself already limited, the precision of our floats are very limited without further enhancement [1].

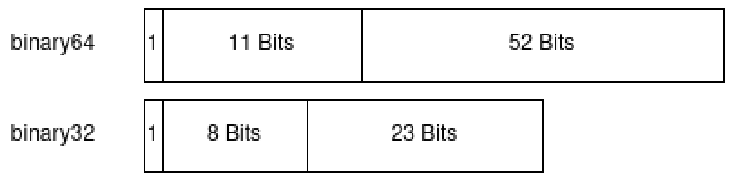
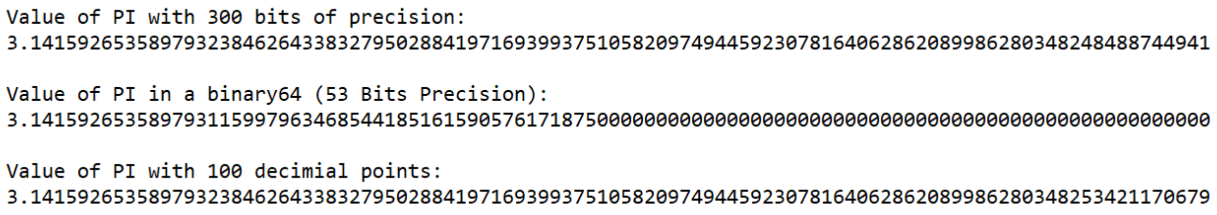


Figure 1.1: Bit representation of binary64 and binary32.

This architectural limit makes storing and manipulating numbers that surpass these constraints a challenging task for programmers. The library in question, GNU MPFR, addresses this problem by using sophisticated algorithms for storing large number's mantissas as sets of 32-bit or 64-bit integers [3, p. 11], outside of the boundaries of the IEEE754 standard and then performing correct rounding to minimise any error and enforce reproducibility.

This limited precision can be best seen when comparing the constant of PI with MPFR at 300 bits of precision against the double precision representation of PI from math.h in C and against the actual value of PI for 100 decimal places:



```
Value of PI with 300 bits of precision:
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348248488744941

Value of PI in a binary64 (53 Bits Precision):
3.1415926535897931159979634685441851615905761718750000000000000000000000000000000000000000000000000000

Value of PI with 100 decimial points:
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679
```

Figure 1.2: PI with different precision.

In *figure 1.2* you can see how the use of MPFR can represent both a much more precise, and larger value than the limited double precision format. This shows that there is an avenue for exploration into making MPFR more efficient as it is a more useful tool over the standard floats and so having it be as efficient as possible is critical.

Further, standard math libraries in the C programming language do not actually ensure correct rounding [4, Section 19.7] and as such MPFR's conformity to the IEEE754 standard [3] makes it a more portable and reproducible library, fit for scientific computing, and as such making it more efficient would be useful in scientific spheres.

Additionally, since MPFR's numbers of arbitrary precision can be broken down into sets of integer numbers, they are suited to be processed in parallel during arithmetic operations. Fittingly, AVX instructions are a set of specialised arithmetic registers or vectors that works by using a single instruction on multiple data (SIMD), being limited to groups of single or double numbers [5]. The benefit of these instructions is that since they affect lots of data simultaneously, they are a parallel architectural design in supported processors, which we will try to integrate with MPFR and test if an increase in computational efficiency can be observed.

1.2 Surrounding work and research

1.2.1 MPFR and GMP

The MPFR library itself is an extension of the GNU MP or GMP library [3], and so for us to talk about MPFR we cannot ignore GMP. GMP fundamentally stores each number's mantissa as an array of machine word size integers (32 or 64 bits depending on the processor), known as 'limbs' by the library [3, 6]. These limbs are stored from least to most significant, with the most significant limb representing the 64 most significant bits (MSBs) of the number on a 64-bit processor, and the exponent stored separately as a single machine word; we will assume this processor for the rest of this report.

MPFR follows the same principle, additionally with an enforced normalisation process to conform to IEEE754 standards [1]. This leads to unused padding bits at the least significant limb's least significant bits after normalisation, as MPFR only generates limbs at fixed sizes. *Figure 1.2.1* shows the 3 least significant bits being unused in the representation of a number with exactly 9 bits of precision, or $(1.01101101)_2$ as a floating point on a 4-bit processor [3, p. 4].

$$\begin{array}{ccc} \boxed{1011} & \boxed{0110} & \boxed{1\mathit{000}} \\ \text{Limb 3} & \text{Limb 2} & \text{Limb 1} \end{array}$$

Figure 1.3: Diagram of limbs.

The diagrammatic view of these limbs helps show how they can be viewed as separate blocks, which could be added in a parallel approach as this paper will try to demonstrate the efficacy of. With [7, p. 19] informing us when two MPFR numbers with the same exponent are added there will be a carry over bit remaining outside of the limb; a remainder would not matter in a serial approach but may affect results in a parallel approach that cannot as easily propagate these results between limbs. As Lefèvre, et al. [7, p. 19] say about propagation "which is more complex to express the borrow propagation in C" when referring to limb-wise subtraction.

$$\begin{array}{ccc} \boxed{1001} & \boxed{0010} & \boxed{1\mathit{000}} \\ \boxed{1011} & \boxed{0111} & \boxed{1\mathit{000}} \\ \hline \end{array}$$

$$\begin{array}{ccccc} 1 & \boxed{0100} & \boxed{1001} & 1 & \boxed{0\mathit{000}} \\ \text{Limb 3 Carry} & \text{Limb 3} & \text{Limb 2} & \text{Limb 1 Carry} & \text{Limb 1} \end{array}$$

Figure 1.4: A potential parallel addition carry problem.

This situation has been partially accounted for by an experimental GMP feature known as nails. These were designed to help carry handling on some processors by leaving a single zero or padding bit at the MSB of each limb [6]. This nail bit could be used to capture the additional carry bits seen in *figure 1.2.1*, however, it was not ported over to MPFR from GMP.

The main difference between GMP and MPFR is MPFR's focus on floating point numbers where as GMP is orientated more towards large integer manipulation. This can be seen in MPFR's usage of rounding modes specified in IEEE754 such as, rounding to nearest, rounding towards zero, rounding towards and away from infinity. Where as GMP only assumes a truncating round to zero [6, p. 124], which inherently lacks complete accuracy in any number system. We will explore the available rounding modes and how to apply them further in *section 2.5*.

1.2.2 Importance of arbitrary precision

1.2.2.1 High-precision

Arbitrary precision is something we take for granted in natural mathematics, being able to represent any result accurately, but when transitioning to a computer system with limited precision to represent floating point numbers, this can quickly lead to errors in calculations. In such cases accumulated rounding errors, bad numerical representations of functions and other unrepresentable values can easily shift the outcome of a computation [8] into a completely different result.

Libraries such as MPFR and other arbitrary precision libraries exist to help overcome these issues, as new discoveries would not have the computational accuracy to be of any value, with fields such as climate modelling or electromagnetic scattering [9] stagnating without the availability of more precision.

It is stated by Ghazi et al. [8, p. 3] that GCC compiler versions 4.3.2 and older fail to always correctly represent floating point numbers due to the use of the GNU C or libc library, which itself works with double precision at most. GCC 4.3.2 and later help tackle this by using the MPFR library in the compiler optimisations. This helps to demonstrate that there is a fundamental need for arbitrary precision, especially in fields of scientific interest, with MPFR bringing consistency across different hardware [10].

While there are other arbitrary precision libraries, i.e. Number Theory Library (NTL) [11] and DecNumber [12], as Ghazi et al. [8] outlined, "GCC uses MPFR to perform constant folding of intrinsic (or builtin) mathematical functions". Due to MPFRs presence in the popular GCC compiler, it has become the primary and best choice to focus on for the inclusion of AVX intrinsics.

1.2.2.2 Low-precision

While we have primarily discussed the use of high-precision, there is also a use of low-precision in scientific computing, with machine learning, especially deep neural networks being a primary constituent of such practices. The use of low precision found only a 1-3% loss of model accuracy at 2-bits of precision when used in classification and model detection, but in turn seeing faster computation [13].

This trend can be observed in GPUs such as the NVIDIA H100 GPU [14, p. 22] which focused on artificial intelligence and high-performance computing. Built upon previous models such as the NVIDIA Tesla 100 V100 GPU it now supports among others, 8-bit (FP8) and 16-bit (FP16) floating point number tensor core architecture.

Further, the NVIDIA H100 GPU FP8 tensor core has been designed with two inputs, E4M3 featuring a 4 bit exponent and 3 bit mantissa, allowing more precision but less range than the E3M4, 3 bit exponent, 4 bit mantissa FP8 input. This showcases the inherit requirement for more precision in software than those outlined by the IEEE754 standard.

Using low precision can be regarded as hardware problem, and its improvement in software limited, however it can be achieved equally such with an arbitrary precision library, an additional one being the CPFloat library [15].

1.2.3 Compiler intrinsics, AVX2 and AVX512

Compiler intrinsics serve as a link between high-level programming languages and low-level processor specific instructions, enabling direct integration of optimised code into different programs.

Intrinsics, specific to both compiler and processor architecture, facilitate efficient SIMD operations, enhancing performance over scalar operations, where single instructions operate on single data. While tailored to a particular compiler, they offer a degree of portability, allowing code adaptation across architectures. For instance the Advanced Vector Extensions (AVX) are a set of SIMD instructions for the x86 instruction set, designed for x86-64 bit processors. They serve as an extension of the previous SSE instruction sets used for SIMD, opening the door for designing parallel algorithms specialised for data parallelism with the processor at hand.

The use of SIMD and parallelisation in general can be critical to creating time-dependent algorithms. Indeed Kouya [16, p. 4,6] found that on average across different arbitrary precision libraries, integrating AVX2 demonstrated a notable decrease in computational time, especially when the algorithm is ran in a multi-threaded context. The AVX2 instructions themselves were designed as 256-bit YMM registers, able to operate on either four 64-bit numbers or eight 32-bit numbers simultaneously called lanes [5, p. 2]. In previous SSE SIMD instructions, data that was not aligned to the 256-bit boundary would not be able to operate, and while the new AVX2 instructions allow some some misalignment, a significant speed down would be expected in these cases. A general outline of the AVX2 lanes is outlined in *table 1.1*.

Lane	Bit Range
Lane 3	0–63
Lane 2	64–127
Lane 1	128–191
Lane 0	192–255

Table 1.1: AVX2 Lanes and bit Ranges when using 4 limbs/lanes.

Recently a new set of 512-bit registers were released that formed the AVX512 instruction set, allowing us to operate on twice as much data. The increase in register size lends itself to a decrease in computational time, with some benchmarks showing AVX512 in a threaded context being faster than scalar by nearly a factor of 10 [17, p. 7, figure 4]. On the other hand, it can be seen in the same results that AVX512 without threading sported no notable difference compared to AVX2 or scalar operations. Cebrian et al. [17, p. 15] concluded that "Speedup is limited by several factors, including algorithmic limitations, data structure organization, application input or the under lying memory hierarchy".

Overall, the use of AVX compiler intrinsics can present a potential speedup, but for this to happen, we must use specialised algorithms designed to work in parallel. From the research on AVX2/512 it clear that we cannot just expect the compiler to vectorise MPFR efficiently itself and we must create a program suitable for the task of doing limb arithmetic in parallel.

Chapter 2

Methodology and code design

2.1 Transforming the problem from linear to parallel

The original implementation as seen in MPFR is done entirely linearly with no aim for a parallel solution, while this makes it useful especially as a generalised approach for differing processor micro-architectures, this can be refined for specific applications.

As we saw in *section 1.2.1*, the design of high precision numbers as sub-units allows us to simultaneously operate on multiple limbs.

Algorithm 1 Parallel Add

```
Input: Two MPFR numbers $a$ and $b$
Output: $c \leftarrow$ Addition of $a$ and $b$
// Assuming an equal amount of limbs in $a$ and $b$
Extract limbs[0...n] from numbers $a$ and $b$

for $i = 0$ to $n$ do
    // Add all limbs $a_i$ and $b_i$ in parallel
    if Addition does not overflow then
        $c_i \leftarrow a_i + b_i$
    else
        $c_i \leftarrow a_i + b_i$
        $c_{i+1} \leftarrow c_{i+1} + 1$
return $c$
```

When looking at *algorithm 1*, we can establish that given two inputs of equal limb size with no overflow and no other transformations to the limbs, the execution time would logically follow to be the same as the execution time of adding a single limb-pair when parallel overheads are disregarded.

However, we cannot always assume that the addition of a limb-pair will produce no overflow bits. Unlike a serial algorithm which can directly propagate the overflow bit to the next output limb, a parallel approach may have overflow bits calculated separately from the initial limb-pair addition, and not as part of said addition. This would leave the result of overflow in a discarded limbo state where it is in neither the next limb or the previous one.

A naive approach to solving this issue can be seen if an overflow is detected, you simply add 1 to the next limb. In a high level parallelisation library such as OpenMP, we could store the results in its own variable to propagate, but as we will explore in *section 2.2*, detecting and adding overflow bits in parallel at the register level is not as simple if we want to minimise redundant AVX register operations.

Additionally, while using a library such as OpenMP would allow for this more naive algorithm,

$$\underset{\text{Limb 3}}{\boxed{\underline{0}101}}\quad\underset{\text{Limb 2}}{\boxed{\underline{0}101}}\quad\underset{\text{Limb 1}}{\boxed{\underline{0}101}}$$

Figure 2.1: Limbs with nail bits.

it would also introduce its own parallel overheads such as thread creation that we cannot control and race conditions between adding limbs and adding their carry bits, all of which would produce an inconsistent result to our calculations which we want to prove can or cannot be sped up with parallelism. This is why we are primarily looking into AVX which works on the register level and allows us to have complete control over how the algorithm operates.

2.2 Creating an AVX addition algorithm

Assembly languages such as x86 assembly have an instruction for add with carry, or adding two numbers and setting a carry flag if there is a carry after addition, this proves useful when you would want to detect carries such as in our parallel addition concept. Without the carry flag we would require a different approach to detect and store the resultant state of an addition operation.

Now unlike x86 assembly, neither AVX2 nor AVX512 instruction sets contain an add with carry instruction and so any attempt to capture this information must be user implemented since there would have to be a carry flag for each subsection of the register. If such information about carries would be omitted we could not accurately compute any value that handles a carry over.

2.2.1 Storing carries

One approach to being able to store a potential carry is to have a separate array buffer for the two input limbs and an output buffer that is a single bit larger. The addition would be calculated per bit of each limb alongside each bit-wise carry, with any remaining carry being stored in the additional memory of the array.

While the above approach may be valid for calculating carry addition in parallel, it would require creating a lot of additional buffers for our AVX registers and writing to them. This approach may be fine and in fact easier to scale for different limb sizes than the next approach, however, it has the drawback of having to create a lot of additional buffers and then mapping these back into the appropriate AVX lanes constantly.

Furthermore, while scalability would be easier with a buffered approach, the fact that limb sizes remain consistent at 64 bits, this becomes a redundant benefit, alongside having to keep AVX lanes within an equal partitioned state of 4 limbs in AVX2 or 8 limbs in AVX512.

Another approach which would require less memory and fewer calls to intrinsics would be that of using a system akin to the nail bits discussed within *section 1.2.1* in regards to GMP, where the MSB of each limb is reserved as a bit to store the carry within itself.

In *figure 2.2.1* we can see how this would look when fitted to the limbs of *figure 1.2.1*. A set of key things to take notice of are:

- Each nail bit is set to 0.
- There has been a reduction in padding or unused bits.

The nail bits are foremost set to 0 so that any carry over would be successfully stored in the MSB of each limb, allowing you to store the state of a carry add, and thus know when to propagate a carry. This will also prove useful over a buffered approach to overflow when we discuss how to deal with propagation.

The reduction/removal of unnecessary right limb padding has a more subtle affect on the outcome of the number. The addition of the nail bit produces n additional bits for n limbs, and since in our AVX2 implementation we can store 256_{bits} / 64_{bits}, or 4 *limbs*, this requires us to use 4 additional bits.

The use of these bits can be seen as having no impact, especially in *figure 2.2.1* where no useful bits were lost and thus all information retained. On the other hand, when there are less than n padding bits, there is bound to be some precision lost in the final MPFR number, as not all of it can be stored. We will discuss more the methods for inserting/removing nail bits and its effects in *section 2.3*.

While discarding some information does prove a downside to this implementation, by offsetting the least significant bits first, we more often utilise the redundant space taken up by padding bits and at the same time minimise our precision lost as this would at most lose 4 bits of precision in a 256 bit precision MPFR number. Nonetheless, for this reason the implementation of the code will use the final precision stated in *table 2.1* so that when we benchmark against the equivalent MPFR solution, we are working with the same amount of precision.

AVX Set	Padding Bits	Final Precision
AVX2	4	252
AVX512	8	504

Table 2.1: AVX Set, Padding Bits, and Final Precision.

2.2.2 Propagating carries

Now that we have successfully transformed our MPFR numbers to be able to safely store the result of a carry bit, we must figure out what the best method for propagating the carry across limbs would be.

First however, we must elaborate on the structure of an AVX register displayed in *table 1.1*. Since a 256-bit YMM register which we will call an AVX register, uses four sets of 64-bit lanes we can fit four limbs each with a nail bit, but we cannot actually move bits between these lanes as they are treated separately. Further, contrary to common byte ordering in code, the lanes are indexed from highest to lowest.

Because we cannot move bits between lanes of a register and have decided to use nail bits, a different solution could be to extract the nail bits and then use AVX intrinsics to add them at different lanes.

To further clarify the nature of an AVX register in the following pseudocode, since it follows a SIMD nature, all lanes are affected simultaneously by a single instruction. For display purposes the AVX instructions may be indiciated parallel via the comments or in for loops when indexing is relevant.

Algorithm 2 AVX2 Carry Add

```
Input: Two MPFR numbers $a$ and $b$ with nail bits
Output: $c \leftarrow$ Addition of $a$ and $b$
// Assuming an equal amount of limbs in $a$ and $b$
Extract limbs[0...n] from numbers $a$ and $b$
// First assign the limbs to the register lanes in parallel
for $i = 0$ to $n$ do
    $AVX\_a_i \leftarrow limbs\_a_{n-i}$
    $AVX\_b_i \leftarrow limbs\_b_{n-i}$
// Add all AVX lanes $a_i$ and $b_i$ in parallel using AVX instructions
$AVX\_result \leftarrow AVX\_a_i + AVX\_b_i$
while Nail bits != 0 do
    // Extract the nail bits in parallel using AVX instructions
    $Carry\_Bits[0...2] \leftarrow AVX\_Result\ RightShift\ 63\ Bits$
    // Add lanes 3...1 carries in parallel to lanes 2...0
    $AVX\_Result[2...0] \leftarrow AVX\_Result[2...0] + Carry\_Bits[0...2]$
return $c$
```

In *algorithm 2*, a high level overview is shown of how you can take an MPFR number, transform it to work with AVX registers and then how to move a carry across if it is detected. For simplicity of the pseudocode, all AVX instructions have been reduced to a human readable definition, however, it should be clear that all instructions work in SIMD fashion on sets of limbs simultaneously.

The pseudocode is broken down into three steps:

1. Assigning limbs to the registers
2. Adding the registers
3. Carrying overflows

The first step is the simplest, using the *_mm256_set_epi64x* intrinsic you can assign the contents of each AVX lane, right to left based on any given memory address.

The second step requires using *_mm256_add_epi64* to add two registers, at this stage you can also use *_mm256_set1_epi64x(0x7FFFFFFFFFFFFFFF)* to mask only the non-nail bits (63 lower bits) of an MPFR number to ensure consistent formatting.

The third and most critical step is a combination of *_mm256_and_si256(...)* to mask away the carry bit from the result AVX_Result register followed by using *_mm256_srl_epi64(carry,* *_mm_cvtsi32_si128(63))* to shift over the carry bit into the LSB of its register. Now using a combination of the intrinsic from step 1 to set the carry bits over a lane and the addition intrinsic from step 2 to add the new carry register to the result register.

When working with the AVX512 instruction set, there are correspondent intrinsics designed for 512 bits, allowing a similar approach to be used as the above demonstrated using intrinsics from the AVX2 instruction set.

2.3 Inserting and removing nail bits

In *section 2.2.1* we briefly discussed the need for nail bits in storing carries and its affect on the precision of our MPFR number, but now we will expand upon how we can achieve this and how to revert our limbs to a structure that MPFR can read as a valid floating point number.

Firstly, let us reiterate that adding nail bits does mean we lose a bit for each limb when storing the mantissa of an MPFR float, leading to a reduced precision in the program. This could be bypassed with methods such as linked AVX registers, but for the scope of this project, as an initial goal we have resorted to instead losing some precision for the sake of code brevity.

2.3.1 Inserting nail bits

Given that the padding bits found in the LSBs of the least significant limb store no actual information, these can be overwritten entirely. A quick approach would be to shift each limb across $1 + n$ times where n is 'total limbs $-$ 1 $-$ current limb index' as this would shift out the appropriate amount of bits to leave a single nail bit in each limb while losing at most ≈ 0.0157 precision when all bits are used with no exponent in the float compared to a 256 bit float. The effect of exponents will be explored more in *section 2.4*.

While the above approach is what we have chosen to pursue, it does not inherently account for the fact that all these limbs are still independent unsigned 64 bit integer blocks and cannot be bit shifted between each other.

Fortunately, GMP exposes a low level function that is designed in assembly, leveraging maximum efficiency for our purposes called *mpn_rshift*, letting us shift across limbs, storing any truncation in a buffer that we use to mask into the high bits of the lower limb.

2.3.2 Removing nail bits

Once the AVX addition code has finished running, we must return the limbs to the original state that MPFR uses, otherwise MPFR will interpret the limbs as a completely different number than intended. However, the difference compared to inserting nail bits now is that to put the padding back to the LSB of the least significant limb we must preserve the state of bits as we shift.

Whereas beforehand during insertion we could arbitrarily right shift limbs or sets off, now we must utilise *GMP_lshift* storing truncated data in a buffer to shift individual limbs. The buffer stores the MSBs being shifted out which we can mask into the LSBs of the next limb to the left, essentially shifting the contents of a limb left-wise between limbs and removing nail bits.

Algorithm 3 AVX2 Remove Nail Bits

```
Input: An MPFR number $a$ with nail bits
Output: $a$ without nail bits

Extract limbs[0...n] from numbers $a$

// First left shift the left most limb (limbs[3] to remove the nail bit
$mpn\_lshift\ limbs_2\ by\ 1\ bit$

// Mask the non-nail MSB of limbs[2] to shift into limbs[3] and then shift
   limbs[2] over twice leaving two 0 LSBs
$mask \leftarrow mpn\_lshift\ limbs_2\ by\ 2\ bits$
$limbs_3 \leftarrow limbs_3\ OR\ mask$

// Repeat this with every limb until you reach the last limb, shifting over n
   times for the current limb index
return $a$
```

2.4 Allowing for generalised inputs

2.4.1 Dealing with exponents

So far we have made our algorithms with the assumption that the exponent of any given MPFR numbers are negligible and would not change the outcome of a calculation. However, this assumption does not hold true when working with any floating point number that follows the IEEE754 standard which allows for a range of exponents and by default has a biased exponent [1], already breaking our current assumptions.

In regular binary or decimal arithmetic, when you want to add two numbers if they have a differing exponent, you must first align the exponents. We must follow the same principle in our addition of limbs, offsetting the limbs and changing the exponent until they are the same.

While in natural mathematics you are also able to align the exponents an infinite amount with no information loss, with our limited mantissa size even with limbs, there is an inherit potential for information loss during this operation. When two numbers have an exponent difference of more than 256, to make these exponents congruent, you would have to offset the mantissa by 256 bits, flushing out any information it previously held when dealing with a 256 precision MPFR float.

To partially combat this we raise the lower exponent to the higher exponent, this results in us having to reduce the mantissa, or in code, right-shifting the limbs. By right-shifting we have some leeway given by the nail bits which have not yet been shifted into the MSBs, and then any meaningful bits that we do lose will be of the lowest increment first, building up gradually.

2.4.2 Negative numbers

When dealing with the addition of negative numbers there are two cases we must consider; the first case being that off the addition of two negative numbers, another case being that off adding a negative number with a positive number.

In the first case we can simply take the addition of both numbers magnitude, as the sign and magnitude representation determines $|a| + |b|$ to be correct regardless of prior sign bits. The result would have the negative sign bit as both inputs were negative.

The second case has the caveat of what happens to the sign bit. When we add the numbers we must pass it along to a subtraction operator and use the sign bit of the larger magnitude. However, since we are not implementing a subtraction operator we could use MPFR's subtraction operator, but for the purposes of testing speedup this would go against avoid MPFR's arithmetic and therefore will not be included.

2.5 Normalisation and rounding

2.5.1 Normalisation

The process of normalisation helps to ensure that all numbers are in a unified format and that as much of the mantissa can be used to store a float with as little precision loss as possible. To normalise a number in IEEE754 format, you must adjust the exponent such that the leading bit is a *1*.

Since the process of normalisation in MPFR is IEEE754 compliant, we can expect the prior behaviour when working with MPFR numbers. The difficulty in our implementation is that since we are using SIMD AVX instructions in our addition, we cannot perform a linear shift of the limbs when normalising as any change affects all the limbs at once.

In the implementation of normalisation for an addition algorithm, we can make useful assumptions that the exponent will only increase due to the most significant nail bit, and that normalisation will only have to shift by a single bit if at least one of the input numbers is normalised prior and both are of the same exponent.

In *algorithm 4* there is a breakdown of the process as it would be done in AVX using parallel instructions, a combination of extracting the LSB of limbs and then masking these into the MSB of the next limb. We can also at this stage perform rounding on a number since if it has been normalised after addition, it has exceeded the 252 bits of guaranteed precision.

Algorithm 4 AVX Normalisation

```
Input: MPFR number $a$ with nail bits
Output: $a$ normalised
// First extract the last bit of each limb in parallel
last_bits[n...0] ← Extract LSB of each limbs[0...n]
// Left shift every last bit in parallel to the bit prior the nail bit
top_bits[n...0] ← Left shift last_bits (62 bits)
// Set the top bits a lane over to the right in AVX and set last lane's top bit
   at the same time
top_bits[n...1] ← top_bits[n-1...0]
top_bits[0] ← $0x4000000000000000$ // This is the most significant limb
// Shift the original limbs over by 1 bit
limbs ← Right shift limbs (1 Bit)
// Insert top bits into the original limbs in parallel
limbs ← limbs OR top_bits
// Increase exponent
$a$_exponent ← $a$_exponent + 1
return $a$
```

2.5.2 Rounding

MPFR has 6 different rounding modes [3], 4 of which are based on IEEE754 rounding modes:

- MPFR_RNDN: round to nearest, with the even rounding rule (roundTiesToEven in IEEE 754).
- MPFR_RNDD: round toward negative infinity (roundTowardNegative in IEEE 754).
- MPFR_RNDU: round toward positive infinity (roundTowardPositive in IEEE 754).
- MPFR_RNDZ: round toward zero (roundTowardZero in IEEE 754).
- MPFR_RNDA: round away from zero.
- MPFR_RNDF: faithful rounding.

The round to nearest mode works as we would naturally round, i.e. if rounding to the nearest integer in base$_{10}$ you would round down numbers *0.1, ..., 0.4* and round up *0.6, ..., 0.9.* The special case of *0.5* is a midpoint between representable values and as such according to IEEE754 roundTiesToEven, it would get rounded down towards the even number *0* as opposed to the odd number *1*.

The directional rounding modes of *RNDD, RNDU, RNDZ, RNDA* all operate in the direction specified, i.e. RNDD will always round down.

Faithful rounding, which is exclusive to MPFR produces a number that if not exactly representable, is either the result of rounding down or rounding up. This helps to ensure that the error of faithful rounding is less than the smallest possible difference in adjacent representable values of the given base [18].

2.6 Version control

Throughout my project git has been utilised, specifically GitHub as a version control system. This allowed clear tracking and distribution of work across different branches, also allowing us to compartmentalise different algorithms and sprints.

GitHub issues were also utilised to help navigate and keep track of workflow, creating clear checklists for required features and whether requirements for implementation have been met.

2.7 Project scope

With this project the aim is to achieve an independent comparison of the arithmetic calculation done by MPFR and an emulation of the process being done with AVX instructions. The original proposal for the project included creating a whole library of arbitrary precision arithmetic using AVX instructions. However, as this would be too large a project to be completed given the time, it was chosen to instead first investigate if there is any merit on a smaller scale to the efficacy of vectorising the calculations as well as forming a partial foundation for a vectorised arbitrary precision library.

Originally, there were aims to additionally complete other arithmetic operations aside from addition, however, it seemed better to concentrate efforts on a single operation. This would still allow us to investigate the core concepts that are required to make parallelisation work in a new or existing arbitrary precision library and if this is a worthy venture for future research.

To help organise the coding aspect of the project and to ensure it is completed it to a suitable standard we broke the requirements of an addition algorithm into its constituent parts and then implemented them in the order of importance, working on a single algorithm at a time akin to a waterfall methodology, but with more flexibility to change as in agile methodology.

1. Learn how AVX2 instructions work, producing rudimentary addition.
   Design a system for how to store and transfer overflows in AVX2.
2. Create the necessary code for debugging the data in limbs and in AVX registers.
   Implement the code to add AVX2 registers with overflow.
   Implement code to transform limbs to store overflow.
3. Allowing for inputs of different exponents.
4. Implement benchmarking to compare AVX2 and MPFR.
   Repeat previous steps for AVX512.
   Perform benchmark analysis of the sub-algorithms.

You can see in *figure 2.2* a flow chart of how the overall algorithm would work and the key steps it would undertake when emulating the work required for creating an addition operation in arbitrary precision.

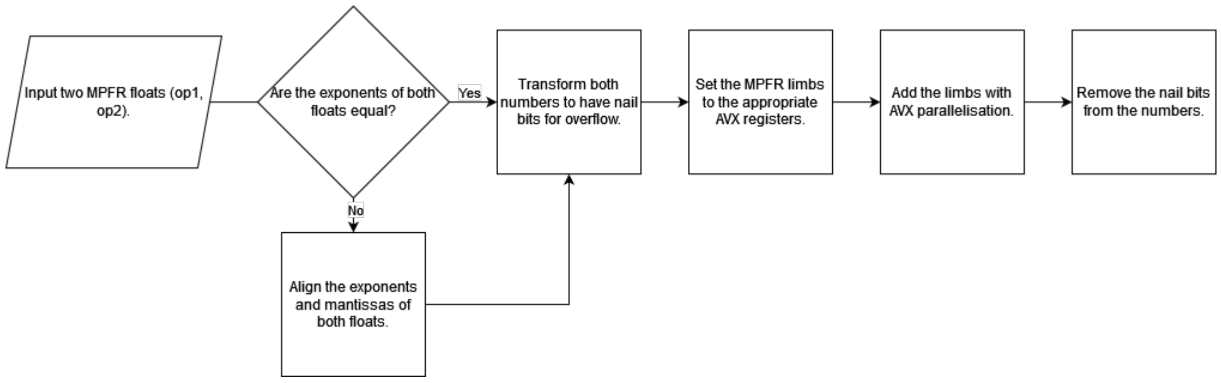

Input two MPFR floats (op1, op2).
Are the exponents of both floats equal?
Yes
No
Transform both numbers to have nail bits for overflow.
Set the MPFR limbs to the appropriate AVX registers.
Add the limbs with AVX parallelisation.
Remove the nail bits from the numbers.
Align the exponents and mantissas of both floats.

Figure 2.2: Flow chart of the algorithm for AVX addition.

Chapter 3

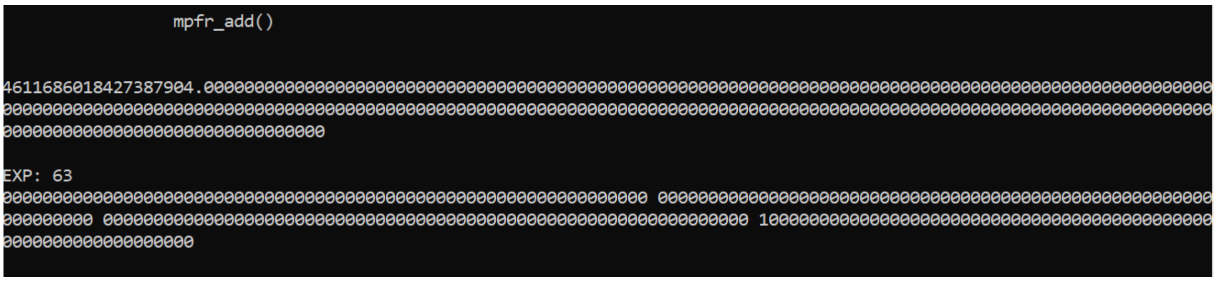
Results

3.1 Software implementation

3.1.1 Initial implementation and testing

When first working with MPFR it was important to create a set of debugging tools to help understand how the memory layout of limbs works and if they can be manipulated as expected. At first we experimented with using *mpfr_ sprintf* to take the limbs directly out as a string buffer which could then be manipulated for transformation into AVX lanes. However, the process of using a buffered output to get the value of limbs would be more inefficient than necessary, especially when trying to optimise for speed up. To accommodate for this, we next looked into trying to access the memory of MPFR limbs directly.

Since each limb is stored as a separate unsigned 64 bit integer, we could not access this through our own interface without the overhead of finding all limbs in memory. Fortunately, MPFR exposes a C structure that allows us to have a pointer for the limbs, which after calculating the number of limbs per precision, allowed us to access all limbs. Having access to the limbs directly helped in debugging by allowing us to create a limb display (3.1) and more importantly allowed us to extract and operate on limb information with no more overhead than MPFR would use, helping us to match its access speed.



```
                mpfr_add()


4611686018427387904.0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000

EXP: 63
0000000000000000000000000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000000000000
000000000 0000000000000000000000000000000000000000000000000000000000000000 100000000000000000000000000000000000000000000
0000000000000000000
```

Figure 3.1: Limb display of the number 4611686018427387904.00 in MPFR 252 precision.

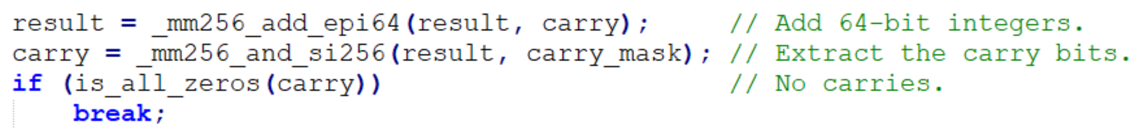
Next was learning how AVX2 instructions operate and can be used in the expansion of a parallel solution. After consulting the Intel AVX documentation [19] and following examples from [16] there was a better understanding of how AVX intrinsics can be used, especially in the context of optimising arithmetic operations.

With this I had enough resources and knowledge to begin working on the main algorithms surrounding the addition operation pipeline. With convenient debugging methods to help streamline the production and full testing of individual algorithms before integrating them into a single addition pipeline.

3.1.2 Creating the algorithms

When approaching the algorithms we started off with implementation of the core addition. This algorithm is designed to circumvent the process of addition in a linear fashion and as such the first problem we had to solve was how to detect the presence any carries in the nail-bit MSBs.

In *figure 3.2* where result and carry are 256 bit inputs of *__m256i_u*, i.e. all four limbs in AVX2 using a C union to access the lanes as arrays. You can see addition in isolation is a single intrinsic command for adding packed integers of 64 bits. We can use this same principle to apply a MSB mask to each packed integer lane simultaneously and return the result in a register called *carry*, which if empty means the addition has completed without further requirements.



```
result = _mm256_add_epi64(result, carry);      // Add 64-bit integers.
carry = _mm256_and_si256(result, carry_mask); // Extract the carry bits.
if (is_all_zeros(carry))                      // No carries.
    break;
```

Figure 3.2: AVX Add and check carry.

Given the case that each limb has no carry over we can expect AVX2 to perform this addition $4x$ faster than the linear approach implemented by MPFR, but we will explore these effects later in *section 3.3*.

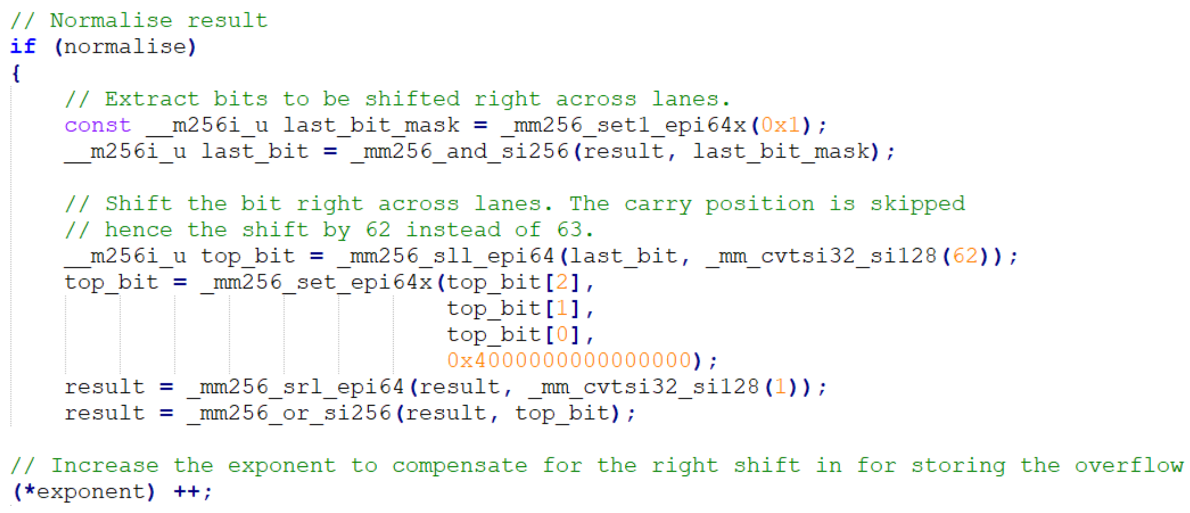
Then next natural step was to solve for normalisation. In the addition of two normalised numbers of the same exponent, normalisation will always occur, and since MPFR automatically normalises any input numbers we had to account for this step to ensure all cases of our program have replicability with MPFR.

In *figure 3.3* you can see how shifting was implemented as a set of shifts between the limbs. This process can be seen akin to a linear approach, which would have have shifted to the next limb over regardless. As such it can be extrapolated that using AVX instructions for normalisation may not be necessary.

During and after the normalisation process, rounding may be implemented. However, in this custom program, after initial tests trying to implement the IEEE754 RoundToNearest TiesToEven rounding method, we found issues getting it to be bitwise accurate with MPFR. This would be further compounded by the additional overhead to isolate the operations to a single limb, and as such would be no more optimal than a linear approach.

To compensate, we chose to initially focus on the MPFR rounding mode called round toward zero, as when working with positive values, this would be the same operation as truncation. This would prove fast to implement in hardware [20] and software, making it an efficient rounding method to compare our program against MPFR with reproducibility of calculations.

However, it should be noted that using MPFR's round to faithful had also passed these tests due to the property of faithful rounding that implies if a value is exactly representable, it will be returned as such. Since we rely on MPFR to initialise the limbs with a number, this would always be exact and so MPFR faithful would require no further rounding.



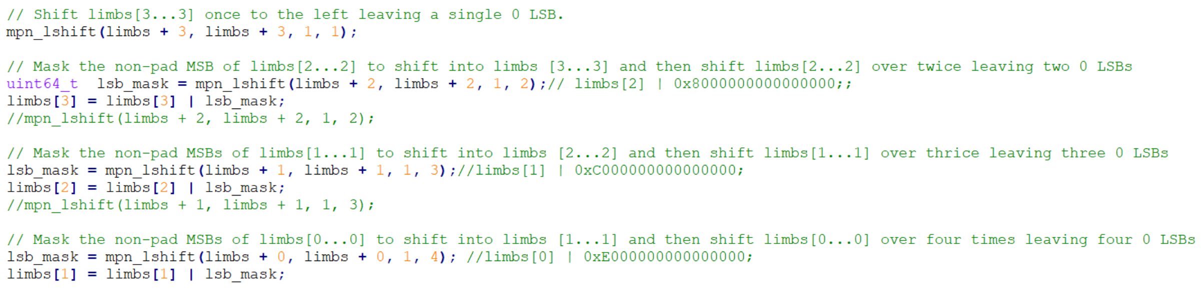
```
// Normalise result
if (normalise)
{
    // Extract bits to be shifted right across lanes.
    const __m256i_u last_bit_mask = _mm256_set1_epi64x(0x1);
    __m256i_u last_bit = _mm256_and_si256(result, last_bit_mask);

    // Shift the bit right across lanes. The carry position is skipped
    // hence the shift by 62 instead of 63.
    __m256i_u top_bit = _mm256_sll_epi64(last_bit, _mm_cvtsi32_si128(62));
    top_bit = _mm256_set_epi64x(top_bit[2],
                                top_bit[1],
                                top_bit[0],
                                0x4000000000000000);
    result = _mm256_srl_epi64(result, _mm_cvtsi32_si128(1));
    result = _mm256_or_si256(result, top_bit);

// Increase the exponent to compensate for the right shift in for storing the overflow
(*exponent) ++;
```

Figure 3.3: AVX Normalisation.

When creating the algorithms for preparing our limbs to have nail bits it was as indicated in *section 2.3*. Using *gmp_rshifts* to sequentially shift right each limb once more than the previous so the 4 unutilised padding bits from MPFR were transferred into nail bits.

Conversely, after the addition had been completed, the nail bits had to be removed. To achieve this in code in the first implementation we created a mask for each limb that would store the MSBs of the previous limb. This mask was used to retain the information of the bits as left shifting with GMP did not inherently perform a circular left shift between limbs. After a shift we would then mask the MSB into the LSB of the new limb.

However, after implementing it, we found that it was not as efficient as required and that the *gmp_lshift* stored any values it shifted off into its own secondary variable. We could then utilise this variable as our mask into the LSB of the next limb to the left, filtering out the nail bits. This is the method depicted in *algorithm 3* and *figure 3.4*.



```
// Shift limbs[3...3] once to the left leaving a single 0 LSB.
mpn_lshift(limbs + 3, limbs + 3, 1, 1);

// Mask the non-pad MSB of limbs[2...2] to shift into limbs [3...3] and then shift limbs[2...2] over twice leaving two 0 LSBs
uint64_t  lsb_mask = mpn_lshift(limbs + 2, limbs + 2, 1, 2);// limbs[2] | 0x8000000000000000;;
limbs[3] = limbs[3] | lsb_mask;
//mpn_lshift(limbs + 2, limbs + 2, 1, 2);

// Mask the non-pad MSBs of limbs[1...1] to shift into limbs [2...2] and then shift limbs[1...1] over thrice leaving three 0 LSBs
lsb_mask = mpn_lshift(limbs + 1, limbs + 1, 1, 3);//limbs[1] | 0xC000000000000000;
limbs[2] = limbs[2] | lsb_mask;
//mpn_lshift(limbs + 1, limbs + 1, 1, 3);

// Mask the non-pad MSBs of limbs[0...0] to shift into limbs [1...1] and then shift limbs[0...0] over four times leaving four 0 LSBs
lsb_mask = mpn_lshift(limbs + 0, limbs + 0, 1, 4); //limbs[0] | 0xE000000000000000;
limbs[1] = limbs[1] | lsb_mask;
```

Figure 3.4: AVX2 Un-padding code snippet.

3.1.3 Relevant hardware/software

As this project aims to capture relevant timing information to compare the usage of MPFR and a custom AVX implementation, it is important to specify the hardware and software used for this project (3.1). To replicate with different hardware would require the user to have a processor that supports the modern AVX512 instruction set.

Category	**Information**
Processor	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
Ubuntu Version (WSL 1.0)	Ubuntu 20.04.6 LTS
MPFR Version	GNU MPFR-4.2.1
GMP Version	GMP 6.3.0
GCC Compiler Version	GCC Version 9.4.0

Table 3.1: System information.

3.2 Performance analysis

3.2.1 Creating a test suite

At first, we looked if there were any pre-existing test suites that we could use. However, with the specialist software required for being able to simulate arbitrary floating points, there was no standardised approach to testing as only a few programs supported correct arbitrary rounding outside of MPFR [3, p. 9].

3.2.1.1 Initial testing method

Since we are using MPFR and GMP's architecture for limbs, any value it produces is properly rounded and thus if our output is the same, it is safe to say we produced the correct value. With this in mind, we started by creating an initial set of randomised test cases and moving onto specialised test cases further into development.

As highlighted, at first we wanted to create a large volume of randomised numbers that would be added in both MPFR and our algorithm to compare timing. To achieve this, a function was created that took an array, adding a random decimal point and filling every other index with either 0s or 1s. This array would then be passed to MPFR so it could initialise the relevant structure for storing the number. To utilise this properly, we setup a designated program for testing.

This program would make 2^{20} random large numbers or 1048576 iterations of the appropriate precision and then time the execution of the full addition pipeline. The choice to use 2^{20} iterations was done so that we could account for outliers in the time taken and that any overheads associated with our timing function or other CPU usage would not largely affect our output time. Furthermore, using multiple randomised values offers the benefit of testing for bitwise congruence across multiple values, giving us confidence in the robustness of our implementation.

3.2.1.2 Timing method

When analysing the performance of our algorithm, it was vital to establish precise timing methods. We utilised *time.h* and specifically benefited from the *clock_ monotonic* feature for consistent time measurements in POSIX compliant systems, independent of system clock variations or external time changes. This choice ensured the reliability and repeatability of our timing tests, especially at the nanosecond scale which is imperative for benchmarking small and fast execution programs.

Using these tools, we isolated the execution function of the respective addition function and wrapped it with a call to the current wall clock time before and after execution. This information was then used to calculate the difference in seconds using the elapsed time equation below.

$$\text{elapsed\_time} = (\text{end} \to \text{tv\_sec} - \text{start} \to \text{tv\_sec}) + \frac{\text{double}(\text{end} \to \text{tv\_nsec} - \text{start} \to \text{tv\_nsec})}{1e9}$$

3.3 Timing results and improvements

The first tests compared AVX2 against MPFR at 252 bits of precision, and AVX512 against MPFR with 504 bits can be seen in *figure 3.5* with average execution time scaled to the nanosecond.

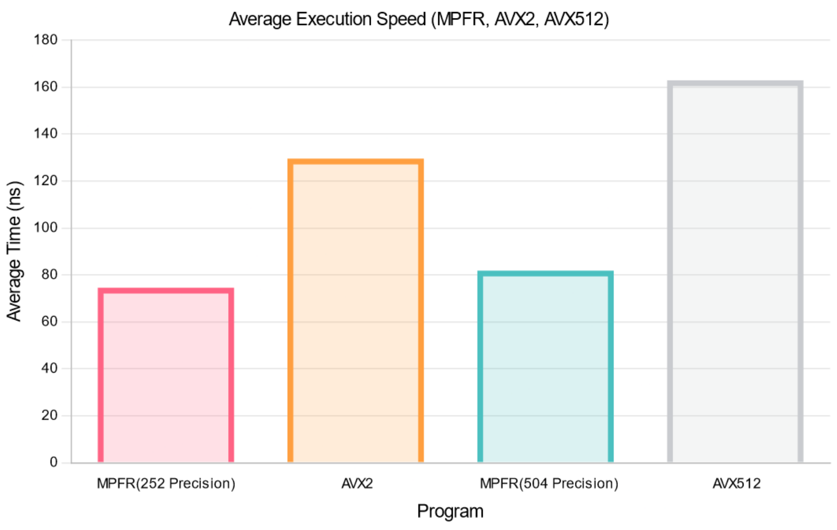


Figure 3.5: Random input tests.

It is clear from a glance that MPFR performed better, but to calculate the exact parallel speedup we used the formula: $S = \frac{T_s}{T_p}$ where T_s is the serial execution time, or MPFR in our case and T_p being the parallel AVX approach.

In *table 3.2*, we see that when using using 252 bits of precision our program is slower by a factor of 1.74 and that when using 504 bits of precision, our implementation is slower by a factor of 2.66.

This is counter intuitive to our understanding of parallelism, especially with AVX2 operating on their respective limbs simultaneously.

This may be attributed to underlying conditions such as the constituent algorithms that are required to prepare the numbers for the parallel addition, or it may be in the addition code itself. For this reason we will try a different set of tests given the following parameters:

1. Random tests without compiler optimisation.
2. Largest representable number test.
3. Smallest representable number test.

The first new test was aimed at comparing the impact that compiler optimisation had on the individual inputs. This was done as a large discrepancy between the original optimised tests and a non-optimised test would indicate that the code itself was not efficient. Hence, the speed-up of the algorithms becomes more obscure when measured.

The second test would test having all overflows and having to add full sized limbs, testing the upper limit edge case. Conversely, a test was done for the smallest value that would not feature any overflows. Exploring these would help direct us to whether the ability to detect and propagate overflow simultaneously had an effect or if the register level parallelisation can be ignored for more traditional optimisation techniques foremost.

In *figure 3.6*, you can see the result for these tests on AVX2 and in *figure 3.7*, the results for AVX512. With the speedup calculations shown in *table 3.3* for AVX2 and *table 3.4* for AVX512.

When analysing the results from AVX2, it can be seen that it performs best in the case that you add the largest values and as such, have overflow in every limb. Conversely, the inverse was the worst test case, the smallest number possible possible. This would suggest that potentially the AVX2 addition algorithm is better suited at dealing with overflows than MPFR, but the other requirements make it slower, which can be seen clearly when no overflows have a significant speed decrease over MPFR.

Similar results can be seen in the case of AVX512, with the distinctive factor being that it is on average, slower than AVX2 no matter the parameters tested. Given the fact that both AVX programs use a parallel approach to the addition, we can infer that the discrepancy is in the other algorithms used.

Program	**Execution Time** (S)	**Speedup**
MPFR (252)	$7.46002979278564 \times 10^{-8}$	
AVX2	$1.29608410835266 \times 10^{-7}$	$\frac{7.46002979278564 \times 10^{-8}}{1.29608410835266 \times 10^{-7}} \approx 0.5763$
MPFR (504)	$6.12750911712646 \times 10^{-8}$	
AVX512	$1.62954329490661 \times 10^{-7}$	$\frac{6.12750911712646 \times 10^{-8}}{1.62954329490661 \times 10^{-7}} \approx 0.3764$

Table 3.2: Parallel speedup on initial tests.

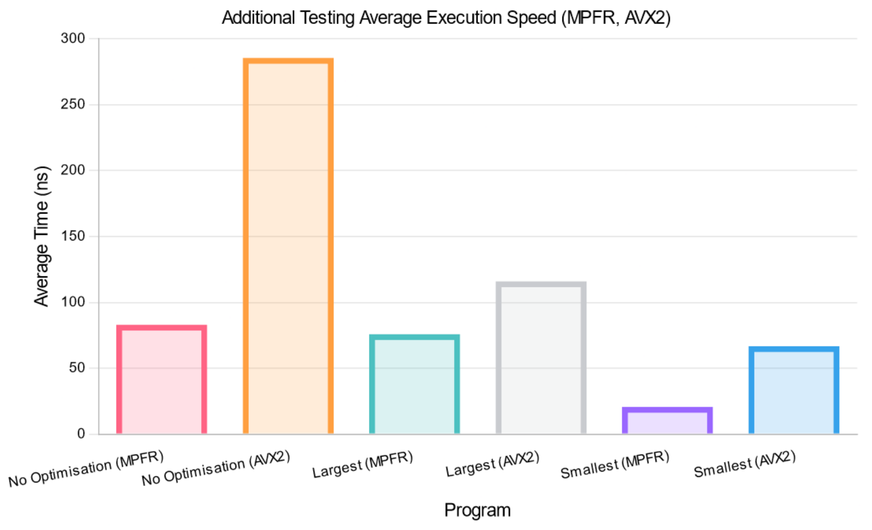


Figure 3.6: AVX2 Parallel speedup on additional tests.

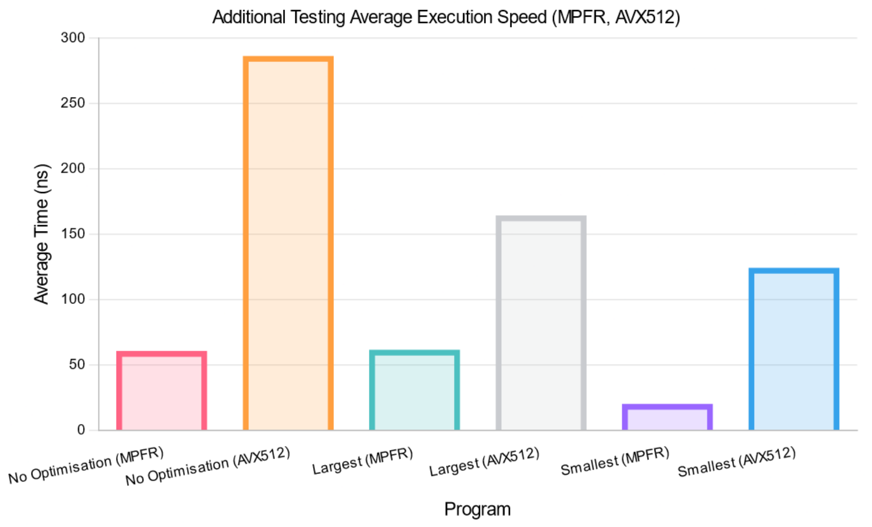


Figure 3.7: AVX512 Parallel speedup on additional tests.

Program	**Execution Time** (S)	**Speedup**
No op. (MPFR)	$8.31369447708129 \times 10^{-8}$	$\frac{8.31369447708129 \times 10^{-8}}{2.85535889625549 \times 10^{-7}} \approx 0.2911$
No op. (AVX2)	$2.85535889625549 \times 10^{-7}$	
Largest (MPFR)	$7.59575691223144 \times 10^{-8}$	$\frac{7.59575691223144 \times 10^{-8}}{1.16068011283874 \times 10^{-7}} \approx 0.6542$
Largest (AVX2)	$1.16068011283874 \times 10^{-7}$	
Smallest (MPFR)	$2.07799921035766 \times 10^{-8}$	$\frac{2.07799921035766 \times 10^{-8}}{6.68241834640502 \times 10^{-8}} \approx 0.3107$
Smallest (AVX2)	$6.68241834640502 \times 10^{-8}$	

Table 3.3: AVX2 Parallel speedup on additional tests.

Program	**Execution Time** (S)	**Speedup**
No op. (MPFR)	$6.08145313262939 \times 10^{-8}$	
No op. (AVX512)	$2.86530055999755 \times 10^{-7}$	$\frac{6.08145313262939 \times 10^{-8}}{2.86530055999755 \times 10^{-7}} \approx 0.2123$
Largest (MPFR)	$6.18299922943115 \times 10^{-8}$	
Largest (AVX512)	$1.64497594833374 \times 10^{-7}$	$\frac{6.18299922943115 \times 10^{-8}}{1.64497594833374 \times 10^{-7}} \approx 0.3758$
Smallest (MPFR)	$2.03425703048706 \times 10^{-8}$	
Smallest (AVX512)	$1.24486479759216 \times 10^{-7}$	$\frac{2.03425703048706 \times 10^{-8}}{1.24486479759216 \times 10^{-7}} \approx 0.1635$

Table 3.4: AVX512 Parallel speedup on additional tests.

To confirm that the slow down of the overall algorithm could be attributed to the culmination of the surrounding algorithms and not necessarily the addition, we decided to test for this in *table 3.5*. The test involved isolating the specific limb addition code in MPFR, ignoring other factors such as exponent alignment. The same procedure was followed for AVX2.

Algorithm	Average Time (s)	Parallel Speedup
MPFR	9.355×10^{-7}	
AVX2	1.78667×10^{-7}	$\frac{\text{MPFR Time}}{\text{AVX2 Time}} = \frac{9.355 \times 10^{-7}}{1.78667 \times 10^{-7}} \approx 5.24$

Table 3.5: Average time for AVX2 and MPFR addition in isolation, and parallel speedup.

What can be seen in the case of AVX2 (3.5) is that the parallel addition in isolation is 5.24 times faster than MPFR's addition at 252 bits of precision. We expected a value near 4 as the AVX2 addition works on 4 limbs at once, the greater value suggests that MPFR's limb addition is not at its theoretical best speed. This speed gain could be due to implementation and external factors such as CPU scheduling.

When investigating the same procedure in AVX512 (*Table 3.6*), it can be seen that even though it deals with twice as many limbs as AVX2, it does not directly correlate to a double speedup. This reinforces the findings of Cebrian et al. [17, p. 7, figure 4] when they showed AVX512 sported little speed up over AVX2 without multi-threading.

We have now established that the parallel AVX approach is more efficient and so when looking at *figures 3.6 and 3.7*, we know that it is preparing a number for use in the parallel addition that is the limiting factor of our program. As this is the case we chose to time the sub-algorithms of aligning exponents, adding nail bits (pad) and removing carry bits (unpad), with results seen in *figure 3.8*

Algorithm	Average Time (s)	Parallel Speedup
MPFR	1.346815×10^{-6}	
AVX512	2.1825×10^{-7}	$\frac{\text{MPFR Time}}{\text{AVX512 Time}} = \frac{1.346815 \times 10^{-6}}{2.1825 \times 10^{-7}} \approx 6.17$

Table 3.6: Average time for AVX512 and MPFR addition in isolation, and parallel speedup.

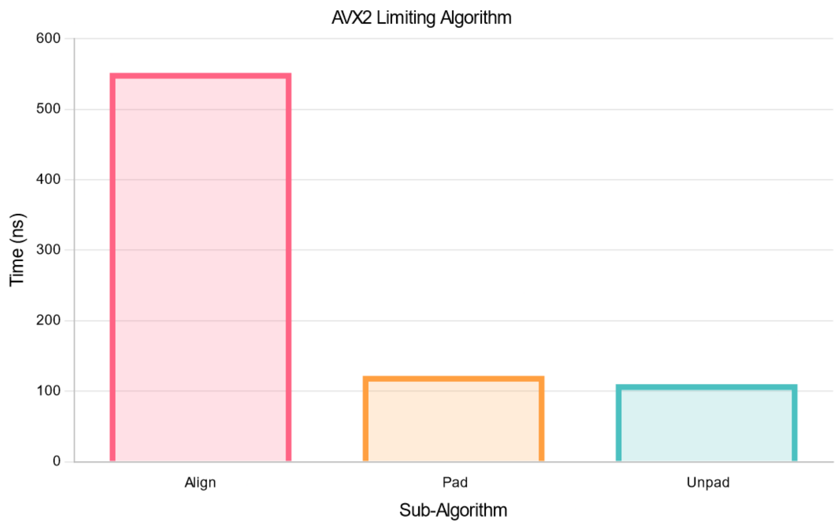


Figure 3.8: AVX2 Addition sub-algorithms limiting factors.

It is clear from comparing the three primary constituent algorithms (3.8), that it is aligning the exponents that takes the most amount of time and as such could be investigated for potential code and algorithm optimisations.

When looking at the code for aligning exponents (3.9) further, an adjustment that could be made is the reduction of costly division operations [21]. The intial code was designed to allow a single function for both AVX2 and AVX512, but by separating it and reducing the call to *'(PRECISION + GMP_NUMB_BITS -1) / GMP_NUMB_BITS'* we can expect minor improvements. The mathematical expression can be replaced by the value 4 i.e. four limbs in AVX2 and 8 in AVX512 as a static variable instead of computing known values repeatedly, as we can enforce requirements for the CPU micro architecture used.

Implementing the established change saw a speed increase of 83.73% from AVX2 exponent alignment code before and after the replacement of the division operation for static values. Similar small modifications across the code can help bring our program in line with MPFR and potentially even faster.

It could also be argued that in exponent alignment, shifting across limbs could be a vectorised operation, as a single shift has to affect all lanes sequentially, so by using AVX to mask and shift all lanes, a similar solution in parallel can be investigated in *algorithm 5*.

This approach would help make the limiting factor parallel and therefore would increase the speed up of our the overall program. However, implementing it (*figure 3.10*) requires more AVX instructions to shift out the top limbs with the correct bit wise correspondence to MPFR which may implement some overhead. On the other hand, seeing as it would largely follow a similar parallel structure to *algorithm 4* when right shifting, we know that this would produce a significant increase in speed.

```
// If a difference of greater than 64
else if (expDifference > 64)
{
    // Keep track of total times a whole limb has shifted
    int limbShiftCount = 0;

while (expDifference > 64)
{
    mpn_rshift((firstNum)->_mpfr_d, (firstNum)->_mpfr_d, (PRECISION + GMP_NUMB_BITS - 1) / GMP_NUMB_BITS, 64);
    expDifference -= 64;
    limbShiftCount++;
}

// The difference should now be less than or equal to 64
mpn_rshift((firstNum)->_mpfr_d, (firstNum)->_mpfr_d, (PRECISION + GMP_NUMB_BITS - 1) / GMP_NUMB_BITS, expDifference);

// Now set the exponent to the shifted value
(firstNum)->_mpfr_exp += GMP_NUMB_BITS * limbShiftCount + expDifference;
}
```

Figure 3.9: Exponent alignment snippet.

Algorithm 5 AVX Limb shift right

```
Input: MPFR Number $a$. Shift value $s$
Output: $a$ shifted right by $s$
// First extract the last $s$ LSBs of every limb in parallel
last_bits[n...0] ← Mask $s$ LSBs of each limbs[0...n]
// Right shift every limb by $s$ bits in parallel
limbs[n...0] ← Right shift limbs ($s$ bits)
// Set $s$ MSBs of the right lane in AVX to the extracted last_bits in parallel
limbs[n-1...0] ← limbs[n-1...0] OR top_bits[n...1]
return $a$
```

```
if (expDifference <= 64)
{
    // First mask out the low bits
    __m256i_u last_bit_mask = _mm256_set1_epi64x(expDifference);
    __m256i_u last_bit = _mm256_and_si256(avx_first_num, last_bit_mask);


    // Shift the bit right across lanes.
    __m256i_u top_bit = _mm256_sll_epi64(last_bit, _mm_cvtsi32_si128(64-expDifference));
    top_bit = _mm256_set_epi64x(top_bit[2],
                                top_bit[1],
                                top_bit[0],
                                top_bit[0] - expDifference);    // First lane is right shifted by expDifference

    avx_first_num = _mm256_srl_epi64(avx_first_num, _mm_cvtsi32_si128(expDifference)); // Shift each lane right by expDiff
    avx_first_num = _mm256_or_si256(avx_first_num, top_bit);

    firstNum->_mpfr_d[0] = avx_first_num[3];
    firstNum->_mpfr_d[1] = avx_first_num[2];
    firstNum->_mpfr_d[2] = avx_first_num[1];
    firstNum->_mpfr_d[3] = avx_first_num[0];

    // Now set the exponent to the shifted value
    (firstNum)->_mpfr_exp += expDifference;
```

Figure 3.10: AVX2 Exponent alignment base code.

Chapter 4

Discussion

4.1 Conclusions

Overall, the aim of the project has not been met to the fullest extent, especially after the initial findings indicating that perhaps utilising AVX was not the best approach. However, after further research revealed that the AVX approach can be more efficient than MPFR we can say that as a pilot study to the use of AVX in arbitrary precision this objective has been achieved.

On the other hand, the objective to create a competitive partial arbitrary precision library has not been met with as much success. While it can be utilised and works in its own right, it is not a complete product and does not itself display significant increase over the standards used currently.

At its core, this project has achieved the primary goal of comparing MPFR and AVX implementations, utilising different AVX instruction sets to see what produces the most efficient arbitrary precision arithmetic for floating points. The research garnered from this project will help direct the efforts of future research into vectorising arbitrary precision operations.

When benchmarking the programs we tried to keep a controlled environment throughout testing, but nonetheless external factors played into the recorded time. With all benchmarking being done at the scale of nanoseconds, factors such as CPU scheduling and memory hierarchy had a relevant effect on the final results. To help mitigate this throughout the report, we have tried to compare results as ratios instead of any direct calculation, as such we were able to compare relative timing and form conclusions that are more impervious to the aforementioned external factors.

4.2 Ideas for future work

In the future now that we have seen the potential of AVX, it would be interesting to see this taken further with changes implemented directly into MPFR. So that users of x86-64 bit architecture CPUs would be able to utilise the potential speed up.

Different instructions sets may also be explored and their potential use cases and impact, such as ARM's Neon [22]. Neon does not boast the same capability to operate on 8 or even 4 limbs simultaneously, but nonetheless can be exploited for vectorisng the operations.

If we want to expand the current program we could return to the idea of implementing subtraction, expanding this further to include the primary arithmetic operations. Further enhancement to the base program could include having it combine AVX additions to allow any multiple of 256 or 512 precision.

We could also bring focus to the rounding modes, not all of which were implemented, so that the program would have greater IEEE 754 compliance in this regard.

References

[1] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[2] Mantas Mikaitis. Week 2: Floating-point arithmetic. Lecture Notes, 2020. University of Manchester COMP36212, Mathematical Systems and Computation, 2020/21 presentation.

[3] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, jun 2007.

[4] GNU C Library. *Errors in Math Functions.* GNU Project, 2022. Accessed: May 3, 2024.

[5] Chris Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23:1–21, 2011.

[6] Torbjörn Granlund. Gnu mp. *The GNU Multiple Precision Arithmetic Library*, 2(2), 1996.

[7] Vincent Lefèvre and Paul Zimmermann. Optimized binary64 and binary128 arithmetic with gnu mpfr. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 18–26. IEEE, 2017.

[8] Kaveh R Ghazi, Vincent Lefèvre, Philippe Théveny, and Paul Zimmermann. Why and how to use arbitrary precision. *Computing in Science & Engineering*, 12(1-3):5–5, 2010.

[9] D.H. Bailey. High-precision floating-point arithmetic in scientific computation. *Computing in Science Engineering*, 7(3):54–61, 2005.

[10] Paul Zimmermann. Reliable computing with gnu mpfr. In *Mathematical Software–ICMS 2010: Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings 3*, pages 42–45. Springer, 2010.

[11] Victor Shoup and others. NTL: A library for doing number theory, 2024. Accessed: April 16, 2024.

[12] Mike Cowlishaw. DecNumber: Decimal arithmetic package, 2024. Accessed: April 17, 2024.

[13] Mohammadhossein Askarihemmat, Sean Wagner, Olexa Bilaniuk, Yassine Hariri, Yvon Savaria, and Jean-Pierre David. Barvinn: Arbitrary precision dnn accelerator controlled by a risc-v cpu. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 483–489, 2023.

[14] NVIDIA. NVIDIA Tensor Core resources, 2024. Accessed: May 3, 2024.

[15] Massimiliano Fasi and Mantas Mikaitis. Cpfloat: Ac library for simulating low-precision arithmetic. *ACM Transactions on Mathematical Software*, 49(2):1–32, 2023.

[16] Tomonori Kouya. Optimization of mixed-precision iterative refinement using parallelized direct methods. In *2022 International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6. IEEE, 2022.

[17] Juan M. Cebrian, Lasse Natvig, and Magnus Jahre. Scalability analysis of avx-512 extensions. *The Journal of Supercomputing*, 76:2082 – 2097, 2019.

[18] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: Multiple Precision Floating-Point Reliable Library, 2024.

[19] Intel Corporation. Intel Intrinsics Guide, 2024.

[20] Clive Maxfield. An introduction to different rounding algorithms. *Programmable Logic Design Line*, pages 1–15, 2006.

[21] Vasilios Kelefouras and Georgios Keramidas. Design and implementation of 2d convolution on x86/x64 processors. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3800–3815, 2022.

[22] Venu Gopal Reddy. Neon technology introduction. *ARM Corporation*, 4(1):1–33, 2008.

Appendix A

Self-appraisal

A.1 Critical self-evaluation

When I first chose to undertake this project I had no idea what AVX intrinsics were, or how they could be applied. With little parallel knowledge in general and little knowledge in numerical systems, I was not well suited for this project beforehand.

This lack of prior understanding definitely had an affect on the direction of my project and its execution. While my project was heavily reduced in scope from the gargantuan task of making a whole AVX arbitrary precision library, it struggled to maintain a clear identity of what the end goal was supposed to be. What ended up happening was a hybrid between a theoretical project of comparing how MPFR could be improved by AVX and the process of making arithmetic operations in AVX.

When preliminary tests showed unfavourable results for the use of AVX this again threw into question the value of this report. While it could be used as a base for future improvement, the negative outcome made it harder to justify the work to build the project, especially given that on the whole, similar papers showed AVX can be useful. However, instead this was converted into a further analysis of why my implementation might have been slower, and proved useful in identifying faults that would make it even better for future work to base of.

At the end of it, while I did not concisely meet the aims of the project I set out to achieve, it still provided insight into how AVX could be used in MPFR for the future. Or even how it could be used to construct the original project, an independent AVX arbitrary precision library. Successfully identifying the pit holes in such an undertaking and providing an overall comparison for a similar approach against the standard.

A.2 Personal reflection and lessons learned

As stated prior, the ideas of parallel programming and numerical systems were mostly foreign to me before starting this project. It is for those stated reasons that I was drawn to this project, succeeding in it would teach me new aspects about computer science that I would not have otherwise learnt, and additionally let me contribute to a field that was not commonly explored at an undergraduate level in the University of Leeds.

This project has helped me solidify my understanding of how the computer systems we work with, themselves, work. While I was not able to implement features such as multiple rounding modes, it helped me understand how I would approach these in the future and the effects they can have in our computers.

Personally, I found researching for this project a particular challenge as I struggled finding relevant information for such a task. With only a handful of complete implementations of similar work, this proved even harder to find information on how to approach making it from new scratch. This caused me to use a lot of intuition from the sparse findings on implementation, and learning to become better at reading and following the documentation / source code of other projects.

Overall, I quite enjoyed working with AVX as it provided a non-conventional way to look at what would otherwise be a simple operation. With this new found knowledge I would be interested in pursuing such a project further, but perhaps first starting in an arbitrary integer library which does not have all the same constraints as an arbitrary float library.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

This project was done using the 3^{rd} party software library GNU MPFR and GMP. One should consider the legalities of using an external library. However, MPFR and GMP are distributed under the GNU Lesser General Public License (LGPL) and as such are free to be modified and redistributed as long as there is no attempt at breaching the terms of the GNU LGPL which I have not.

A.3.2 Social issues

Since this project was a purely theoretical venture into the possible improvements to an already pre-existing software library, the social impact is minimal. Only being impactful to members of society who are already interested in high-performance computing.

However, the project does require access to compatable hardware, which may not be easy to obtain, this could create a divide between users with better access to newer technologies if they wanted to utilise the findings of the report.

A.3.3 Ethical issues

Since my project does not include any humans, animals or otherwise living beings nor does it aim to include any, it can be considered of low ethical risk. However, for good practice it should be made clear to users how they could use this new software if they so wish.

A.3.4 Professional issues

The implementation of this project in the standard MPFR library would require careful cooperation between the GNU MPFR team and relevant stakeholders. Especially since an even more efficient use of MPFR may bring it to new light in areas of interest such as trading.

Maintaining the parallel code should also be considered, proper maintenance and exploitation of the parallel approach would require knowledgeable users in the field.

Appendix B

External Material

The MPFR library was used to compare against and for its limb structure. Version: GNU MPFR-4.2.1.

The GMP library was also used as MPFR is built upon it. Version: GMP-6.3.0.

Initial AVX2 code based off the discussed algorithms provided by Massimiliano Fasi.