University of Dublin, Trinity College

Masters Thesis

Scientific computing with non-standard floating point types

Author:
Vlăduţ Mădălin Druţa

Supervisor:
Dr. David Gregg

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

in the

Department of Computer Science and Statistics

May 2015

Trinity College
The University of Dublin

Declaration of Authorship

I, Vlăduţ Mădălin Druţa, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signed: ______________________________

Vlăduţ Mădălin Druţa
May 21, 2015

"The purpose of computing is insight, not numbers."

Richard Hamming

UNIVERSITY OF DUBLIN, TRINITY COLLEGE

Abstract

Faculty of Engineering, Mathematics and Science
Department of Computer Science and Statistics

Master of Science in Computer Science

Scientific computing with non-standard floating point types

May 2015

Author: Vlăduţ Mădălin Druţa

Supervisor: Dr. David Gregg

This study examined the possible use of non-standard floating point types for scientific computing. The question of this thesis is: "Is there anything to be gained by supporting non-standard floating point data types?".

There are several gaps in the literature that this thesis will aim to address. There could exist potential in the use of non-standard floating point types. This thesis investigates in particular the non-standard floating point type of 48-bit size. As long as there is no need for the full precision of floating point standard size of 64, the 48-bit non-standard type requires less memory, reduces the amount of data movement and might be faster than the standard size of 64-bit.

The initial findings showed that the non-standard (f48-bit) without the use of Streaming SIMD (Single Instruction Multiple Data) Extensions (SSE) is slower than using the standard 64 bit floating point. However, using SSE intrinsics the non-standard 48-bit floating point is competitive with the standard 64-bit. The results shown are good for a floating-point type that is not supported in hardware.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. David Gregg for allowing me to pursue this research under his supervision, and his guidance and continuous help in completing this dissertation.

Besides my supervisor, I want to thank Andrew Anderson for his support and guidance throughout this research. I also want to thank Mark Whelan and John Lennon for their help and support throughout the past five years at Trinity.

Another huge thank you to my girlfriend Loredana, who has had to endure my constant reading, writing and reflecting over the past year. For all of the lonely nights and her unwavering support and encouragement, I will be forever grateful.

Last but not the least, I am very grateful to my parents who have accompanied me through every difficult deadline over the past five years at Trinity.

Contents

List of Figures

List of Tables

Abbreviations

SSE	**S**treaming **S**IMD* **E**xtensions
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
BLAS	**B**asic **L**inear **A**lgebra **S**ubroutines
F64	**F**loating point type **64**-bits
F48	**F**loating point type **48**-bits
FPU	**F**loating-**P**oint execution **U**nit
DVFS	**D**ynamic **V**oltage and **F**requency **S**caling
IPC	**I**nstructions **P**er **C**ycle
PCM	**P**erformance **C**ounter **M**onitor
MIL STD	**Mil**itary **St**an**d**ard
TI	**T**exas **I**nstruments
CPU	**C**entral **P**rocessing **U**nit
IA32	**I**ntel **32**-bit **A**rchitecture
IA64	**I**ntel **64**-bit **A**rchitecture
GCC	**G**NU* **C**ompiler **C**ollection
GNU	**G**NU **N**ot **U**nix
RTDSC	Read Time-Stamp Counter and Processor ID IA assembly instruction

Chapter 1

Introduction

This research explores the idea of using a non-standard floating point type in scientific computing. This dissertation presents a design for such a non-standard floating point type, along with the conversions to the standard 64-bit type (double). The dissertation also provides implementation of selected basic linear algebra subroutines (BLAS). The functions selected provide an answer to the efficiency question. Additionally, using a non-standard type of floating point reduces the memory bandwidth and could possibly have a diversity of applications.

1.1 Motivation

One of the motivations for this project was to find if there is the possibility of defining and emulating the use of non-standard floating point types on current existing hardware.

Another interesting motivation point for this research was to discover if newly defined non-standard floating point types can be competitive with current standard ones, in terms of computation time and performance. This brings us to the main research question outlined below.

The concept of the Internet of Things is becoming ever more widespread. We see conventional objects combined with electronics and embedded systems to make our lives easier in many ways. With such a boom in this area, the need for embedded system has risen. Embedded systems are the core motivation towards the use of non-standard floating point types, in order to save on memory usage with trade-off being precision.

Floating point types have in general a diversity of applications. The motivation for using a non-standard type of floating point relies on the memory savings achieved by using smaller non-standard floating point types. A scenario where 32-bit representation does not provide the accuracy required, but the 64-bit representation is an overkill in terms of the memory usage leads to a big-data problem. The use of non-standard floating point types could offer a solution to this problem.

From another perspective, the use of multiple cores convey cheap computation. However, the bottleneck is caused by data movement which is expensive. Memory bandwidth plays a huge role in the performance of such computations. The use of non-standard floating point types that suffice the accuracy of the intended system would require less memory bandwidth. Saving memory bandwidth has implications in the energy efficiency of the system. Less memory bandwidth required — less the energy consumption.

Saving memory, energy, and data movement are very important factors in a wide range of domains where the extra precision and accuracy of the results is required.

1.2 Research Question

According to the motivation and small introduction of this project, its aim focuses on the exploration of non-standard floating point types and their usage. The research conducted aims to answer the following question:

> Is there anything to be gained by supporting non-standard floating point data types?

Having this question in mind and as long as there is no need for the full precision of the standard double (F64) type, then the non-standard might be faster, require less memory and reduce the amount of data movement.

1.3 Thesis Outline

This report presents the research that has been conducted towards the final answer to the research question. The report is structured in six chapters including this Introduction (Chapter 1).

Chapter 2 presents a solid state of the art research being done towards gathering information as part of the background or related work to the scope of this research.

Chapter 3 presents the design of the new non-standard type along with the structure that is going to be used in computation using this newly defined type.

Chapter 4 goes into the details of the implementation, from the conversions required to each function selected from the different levels of the Basic Linear Algebra Subroutines.

Chapter 5 consolidates the experiments. It goes into the details required for setting up the environment to run the experiments. A description of the results and their evaluation is provided in chapter 5.

Finally, Chapter 6 presents the conclusions drawn from the investigations and results gathered and answers the initial research question along with some future work mentions.

Chapter 2

State of the Art

This chapter consolidates the background of this dissertation research project. It also gives an overview of the existing related work that has been explored towards achieving the aim of the dissertation.

2.1 Background

As part of the background research towards achieving a concise response to the research question (in Chapter 1), I have investigated the formal specifications of the floating point types. These are defined in the IEEE-754 standard [1] and have been widely adopted since its first version back in the 1980s. A later version of the IEEE-754 [1] defines the structure of two floating point types: 32-bit and 64-bit. Both of these floating point types are of main interest of my work in this dissertation. The standard specifies the number representations and the structure: sign, exponent and mantissa location in terms of bit placement and their interpretation.

IEEE-754 [1] also defines five different rounding modes. These rounding modes are:

a) To nearest, ties to even

b) To nearest, ties away from zero

c) Toward 0

d) Toward $+\infty$

e) Toward $-\infty$

The most commonly used rounding method is (a) To nearest, ties to even. I have decided to use this mode in the work presented in this dissertation. Revy in chapter 1 of his thesis (Implementation of binary floating-point arithmetic on embedded integer processors) [2] outlines and further explains the structure of floating point types based on the standard IEEE-754 [1]. Revy, also explains the implementation of the rounding methods in Chapter 2 of his thesis [2]. This has been the main sustain point of the rounding mode implementation in my dissertation.

Operations with floating point types require the use of a floating point unit (FPU). One or more FPUs may be integrated with the central processing unit (CPU). The usage of the FPUs can vary as exemplified by Deschamps et al. in 'Floating Point Unit' [3]. This shows the possible usage of the FPU for different situations and different type of data other than the standard floating point types.

In order to examine the performance of a newly defined floating point type this dissertation addresses the most common functions of the basic linear algebra subroutines. These functions defined for Fortan use by Lawson et al. [4] and extended by Hammarlin et al. [5] along with the addition of level 3 by Dongarra et al. [6] represent a solid baseline of functions that can be used to test the performance of the non-standard floating point type. BLAS is defined on three levels:

a) vector functions — dot product,

b) matrix-vector functions — matrix-vector multiplication, and

c) matrix-matrix functions — matrix-matrix multiplication.

These three different levels offer analysis of the performance of the non-standard type compared to the standard type across a variety of functions and different levels of complexity of the computations performed.

In order to gain and analyse performance for both of the types (standard and non-standard) a solution comes by adopting and implementing the BLAS functions with the use of Intel's streaming SIMD (single instruction multiple data) extensions (SSE) [7]. The SSE intrinsics provide access to a large set of Intel instructions without the need to write assembly code. These functions offer the ability to perform the same single computation (operation) on multiple data lanes. Along with the knowledge of the latest version of SSE (SSEv4), the Intel intrinsics guide [8] allowed the search for specific functions and their description and usage.

2.2 Related work

The related work section of this project is quite slim. There is a gap in the literature of this particular topic.

Telemetry standard RCC document [9] appendix O shows a range of different floating point representations including the IEEE-754 [1]. However, interesting definitions that are related to the interest of this research are the military standard (MIL STD) 1750A and the Texas instruments (TI) which both define interesting non-standard floating point types.

The Texas instruments defines a 40-bit extended precision floating point type. Its structure is defined as 8 bits for the exponent, followed by the sign bit (one bit) followed by 32bits representing the fraction. This structure is related to TI's structure of a 32-bit floating point which is represented and structured in similar fashion, except it is using only 23 bits for the mantissa.

A more interesting definition in the Telemetry standard RCC document [9] is the MIL STD 1750A. This standard was initially introduced in 1980 with the aim towards airborne computers. It provides the formal definition of a 16-bit computer instruction set architecture including both required and optional components. Within the optional components the structure of the two types of floating point is defined.

MIL STD 1750A defined a 32-bit single precision floating point type and a 48-bit double precision floating point type. It defines the 32-bit single precision floating point type as 1 bit for the sign, 23 bits for the fraction followed by the remaining 8 bits for the exponent.

The design approach for a MIL STD 1750A microprocessor([10]) explores in more detail the structure of the double precision floating point type of the MIL STD 1750A. Its structure is very similar to the single precision floating point type. It requires the extra 16 bits of fraction to be padded after the exponent, with the mention that the most significant fraction is the one following the sign.

With the acceptance of the IEEE754 initially in 1985 and it's more recent version from 2008, the MIL STD and the TI definition of floating point types are not as common in the industry.

A more interesting and more recent analysis of using a half precision float (16-bits) type is presented in Intel's article 'Performance Benefits of Half Precision Floats' [11]. Patrick Konsor, defines the F16 as a storing type and the focus is on improving the locality and data transfer. The trade-off between using the typical 32-bit floating point type and the half precision float defined is the precision and range against half the storage and memory bandwidth. This article is addressing alignment issues and performance of the half floats defined.

Chapter 3

Design

For the purpose of analysing the question of efficiency, a new non-standard floating point type is chosen. The non-standard floating point picked as representative for non-standard floating point types is on 48-bits (F48).

This chapter will put emphasis on the structure of the newly defined floating point type on 48-bits (F48) in comparison with the standard double (F64). It will also focus on the computation structure that is required for the mathematical functions defined in the Basic Linear Algebra Subroutines (BLAS).

3.1 F48 structure

The aim of this section is to describe in detail the structure design of the newly created floating point type on 48-bits (F48). Along with the structure description this section is aimed to discuss the differences in precision, range and memory use that arise from comparing the standard double (F64) type against the defined non-standard F48.

The structure of the typical double (F64) specified in IEEE-754(2008) [1] is as shown in the Figure 3.1 below. The structure is defined as one single bit for the sign of the floating point number value. The sign bit set (equal to one) representing negative number and the sign bit equal to zero representing positive number value. The sign bit being followed by 11 bits representing the exponent, and the remaining 52 bits representing the mantissa.

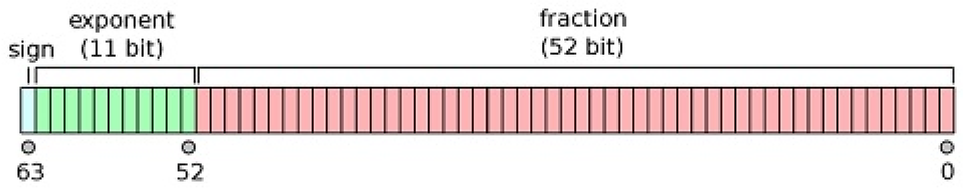

FIGURE 3.1: Bit structure of the standard double (F64).

This leads to the design decision of structuring the non-standard floating point type F48 in a similar manner. Its structure is defined as a single bit for the sign of the number value, 11 bits for the exponent and the remaining 36 bits in this case for the mantissa. The structure described is shown in the Figure 3.2 below.

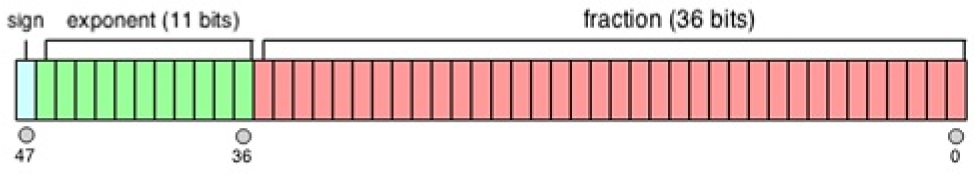

FIGURE 3.2: Bit structure of the non-standard F48.

The definition of double (F64) [1] shows a precision of $\approx$ 15.9 decimal digits. From the design structure of the the non-standard F48 type its precision is of $\approx$ 10.8 decimal digits.

The precision of both types can be computed using the formula $m \times \log_{10}(2)$ where m is the number of mantissa bits. With the difference of 16 bits from the mantissa, the precision of F48 type is reduced by $\approx$ 5.1 compared to the double (F64) standard [1].

An obvious difference between the F64 (double) defined in the IEEE754 standard [1] and the newly defined floating point type on 48-bits is the memory required. A difference of 16 bits that allows the store of four F48s in the same space required to store only three F64s (doubles). In other words, in 192 bits required to store three F64s (doubles) could be stored four F48s.

3.2 Computation structure

As part of the design decisions regarding the structure and format of the chosen non-standard floating-point type, computational structure is required. This section defines the design decisions made towards the computation structure.

The research is inclined towards finding an answer to the efficiency question. This requires a set of mathematical functions to be tested using the F48 type and compared to the standard F64 (double) type. Computation for this is required such as addition, multiplication, subtraction etc.

In a typical situation, the processor contains one or more floating point execution units (FPU). The FPU execution units perform typical operations such as addition, subtraction, multiplication, division etc. These units are designed to be used with the standard sizes defined in IEEE-754 [1]. Therefore the use of the FPU requires the input to be in a supported format.

Floating point type on 48-bits (F48) is not supported by a standard FPU. However operations as described above are required to be performed on F48. Therefore my design solution involves the use of a standard FPU execution unit to perform the standard operations using standard types.

Below, Figure 3.3 shows the designed flow of computation. This involves the need of converters (to and from doubles - F64). Performing operations on floating point in the FPU but using the non-standard F48 type defined, requires the conversion of the operands to the standard type (double - F64) perform the operation using the FPU and the resultant to be converted back to the same non-standard type (F48).

This flow of computation allows to support non-standard floating point types to be emulated in software. The decision in using this computation flow, forces the library that defines the F48 floating type to consist of conversion methods. The conversions dictated by the computation flow are *a*) to double (F64); and *b*) to non-standard F48;

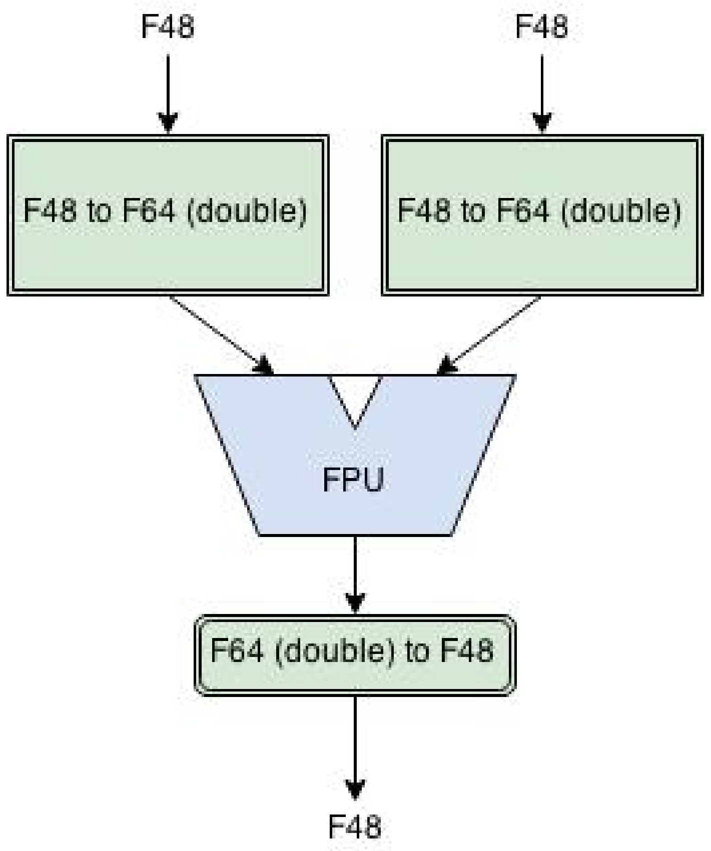

FIGURE 3.3: Graphical representation of the flow of computation.

A small description on the requirements of each of the conversion can be found below. Conversions required as enumerated above are described in details in Chapter 4 (Implementation). These requirements are derived from the designed computation structure.

a) To double (F64) - takes as input a single F48 type number and converts to a standard double (F64) type.

b) To non-standard F48 - taking a single standard F64 (double) type and correctly converting it to a non-standard F48.

The computation structure I picked for the non-standard F48 type makes use of the existing standard F64 (double) unit. The reason for this choice is that computation is cheap. However, data movement is expensive and this brings a strong reason for doing the computation on the existing hardware.

Chapter 4

Implementation

This chapter describes a general approach to the implementation of the library. The following section will extend towards the technical details of the conversions required, and finally the last section of this chapter will focus on the implementation details of the Basic Linear Algebra Subroutines (BLAS) that will be used in order to evaluate the performance and provide an answer to the research question. The research question stated in chapter 1 is:

> Is there anything to be gained by supporting non-standard floating point data types?

The language of choice for the implementation of the dissertation is **C++**. This allows the definition of the new type as a class and allows the implementation of specific operators that act on the new type.

4.1 Approach

The implementation has two main sections (Conversions and BLAS) which are described in detail in the following sections of this chapter. The implementation approach is to implement *a*) conversions described in Chapter 3 (Design); and *b*) a significant function from each of the three levels of the BLAS. The conversions are implemented as part of the library defining the F48 type. These library functions are then used to implement standard functions from BLAS. Functions of BLAS are implemented using both the standard double (F64) and the newly defined F48 (defined in Chapter 3 - Design).

The newly defined F48 type is represented in the implementation as a class. The F48 class contains a single private attribute of type *unsigned long long* on 48 bits. The class forces the use of only 48 bits as shown in Listing 4.1 by applying the GNU compiler collection (GCC) specific attribute packed to the class.

```
class f48 {
private:
  unsigned long long num:48;
public:
  f48() { };
  f48(double value);
  operator double();
} __attribute__((packed));
```

Listing 4.1: F48 type class definition

The approach in implementing the functions of BLAS is to implement at least one representative function from each of the three levels using the Intel's Streaming SIMD (single instruction multiple data) Extensions (SSE) [7].

Lomont's Introduction to Intel advanced vector extensions(2011) [12] offers a more detailed overview of both the legacy SSE4.2 (used in this project) and the new Advanced Vector Extensions (AVX — which rely on the SSE versions). This offered assistance with more advanced SSE intrinsics that are required for the implementation of the different functions of the BLAS.

The non-SSE function implemented for initial/partial results is the representative function matrix-vector multiplication of level 2 in the BLAS. This function (matrix-vector multiplication) showed that the non-SSE versions are slow even when using the standard floating point type. Results of the non-SSE version of the F48 type and the F64 type are present in chapter 5. Therefore, in order to keep the performance of the standard type as high as possible I have decided that the focus of the project should change. This change means that the implementation and measurement of performance of the BLAS functions in both of the types is done using the Intel's SSE.

4.2 Conversions

This section of the implementation aims to explain in detail the conversions required and implemented in this research. The conversions are *a*) to standard F64 - double (from non-standard F48); and *b*) to the non-standard F48 (from the standard F64 - double).

These conversions are required for the flow of computation as described in subsection 'Computation structure' of chapter 3 (Design). The computation is implemented using the standard type (double - F64). Two types of conversions are required as described in chapter 3, but also an SSE version of the conversion to non-standard F48 type is required. The reason for this conversion in SSE being required is that the BLAS functions implemented are using SSE. Keeping the SSE level of parallelism through the computation flow is important from a performance point of view.

4.2.1 To standard F64 (double)

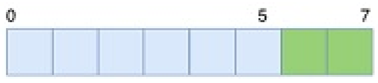

Figure 4.1: Graphical representation of the conversion of the non-standard F48 to the standard F64 (double).

Conversion of F48 to the standard F64 (double) requires the missing two bytes to be padded to the existing data. As shown in Figure 4.1 the existing six bytes (48-bits) of the f48 require the addition of extra two bytes with the value of 0 to convey to the size of 64-bits required for the standard double.

To achieve a correct conversion without using SSE, the 48 bits from the class defined earlier in this chapter, have been shifted 16 times to the left. This offers the extra 16 non-set(0) bits. Reinterpreting the bits from an *unsigned long long* type to the *double* type assures that the alignment (little endian) is correct. Listing 4.2 shows the union that allows the interpretation of an *unsigned long long* as *double*. The union is used for the conversion without using SSE as shown in listing 4.3.

```
union conversion_union {
  unsigned long long l;
  double d;
} convert;
```

Listing 4.2: Union definition

```
f48::operator double() {
  convert.l = (this->num)<<16;
  return convert.d;
}
```

Listing 4.3: Conversion from non-standard F48 to double

Conversion to double is mainly required for the ability to display the result on screen. However, there is a need for an SSE implementation of the conversion. This is mainly required for being able to load the F48 values as doubles in the *__m128* type defined by SSE. Listing 4.4 shows the conversion being implemented in SSE at load time. The snippet of code shown in the listing is hiding the use of the values, showing only the load and conversion happening. The details of the use of the values will be detailed in the following section of this chapter. The first two values of the array *a* are loaded in the *__m128i* vector. Then each element of the vector is being shuffled based on the mask defined. The mask defines the extra two bytes to be set as 0 at the end of the F48 value. This is represented by the appended 255 value at the end of the *epi-8* mask. The other numbers represent the counts of the bytes to be loaded from the original load.

```
f48 * a;
__m128i mask = _mm_set_epi8(11, 10, 9, 8, 7, 6, 255, 255,
                            5,  4, 3, 2, 1, 0, 255, 255);
for ( int i = 0; i < SIZE; i+=2 ) {
__m128i a01 = _mm_loadu_si128((__m128i*)(&a[i]));
a01 = _mm_shuffle_epi8(a01, mask);
/* ... */
}
```

Listing 4.4: Conversion from non-standard F48 to double using SSE

4.2.2 To non-standard F48

Converting the standard F64 (double) to the non-standard F48 requires correct rounding as truncation of the extra 16 bits is not enough. Removing the extra 16 bits erases data therefore removing precision. To calibrate the result to the correct value requires the implementation of a rounding method. The default rounding mode in **C++** and other programming languages such as **C**, **Python** etc... is *round to nearest, where ties round to the nearest even.* This mode of rounding is by far the most common mode, therefore I have selected it as the method of rounding for this conversion.

IEEE-754 [1] defines rounding to nearest, where ties round to the nearest even. This method does the expected for most of the values. For example rounding 1.132 down to to digits yields 1.13 and rounding 1.159 yields 1.16. However, rounding values that are exactly at halfway are rounded to the nearest even digit. For example 0.125 rounded down to two digits rounds to 0.12 similarly when rounding 0.675 down to two digits rounds up to 0.68.

This rounding method is also described by Revy in chapter 2, section two of his thesis (Implementation of binary floating-point arithmetic on embedded integer processors) [2]. Rounding to nearest, where ties round to even is a method that relies on three bits. *Guard (g)* bit, *Sticky (s)* bit and the second-last bit of the final result *(m).*

The *Guard (g)* bit is the last bit of the value after truncation (elimination of the extra 16 bits in our case). The *Sticky (s)* bit is the logical or of the bits removed in the truncation process. The second-last bit of the final result *(m)*, can be seen as the bit before g. Revy [2] defines the rounding procedure by an addition of the *rounding bit (b)* to the truncated value. Bit b is defined based on the mode of rounding. For the chosen method (to nearest, ties to even), *rounding bit (b)* is defined in Revy's thesis [2] as follows:

$$b = \begin{cases} 0 & \text{if } g = 0; \\ 1 & \text{if } g = 1 \text{ and } s = 1; \\ m & \text{if } g = 1 \text{ and } s = 0. \end{cases}$$

The extraction of the sticky bit, guard bit, and the bit before the guard bit have been implemented as shown in listing 4.5. The listing also shows the computation of the rounding bit (b) [2] (which is dependant on the bits g, m and s) on line 7. The rounding bit is then used to be added to the truncated result. Overflow of the mantissa would increase the exponent by the structure of the newly defined type, therefore the conversion is being performed correctly. The final result is the correct approximation of the original standard value expressed in double type.

```
  f48::f48(double value) {
2   convert.d = value;
    unsigned long long tmp = convert.l;
4   bool S = (tmp & 65535)>0 ? 1:0; // sticky bit
    bool G = (tmp >> 16) & 1; // guard bit
6   bool M = (tmp >> 17) & 1; // bit before guard bit
    bool b = G & (M | S); // rounding bit
8   /* ... */
  }
```

LISTING 4.5: Conversion from standard F64(double) to non-standard F48

The above conversion is not using SSE and conversions using SSE are required for the computation flow to stay as using SSE. The version using SSE has to apply the same rounding method to two values (items) at a time. Extraction of the *Sticky* bit (s) is done by using a mask with all the bits set corresponding to the bits being removed from the standard double type. Figure 4.2 shows the __m128 variable used as a mask to extract the *Sticky* bit. Firstly the mask is used to extract the bits that are being removed by applying logical and between the mask and the original double value. This results in clearing all the other bits before the bits to be removed, and keeping the original 16 bits to be removed. However, adding (arithmetic add) this temporary extraction to the same mask would result in an overflow at the position of G if any of the bits in S is set, creating a logical or of them. As Revy [2] has defined the *Sticky* bit (s) to be the logical or of all the bits being removed, this addition computes this logical or by addition and inspecting the overflow.

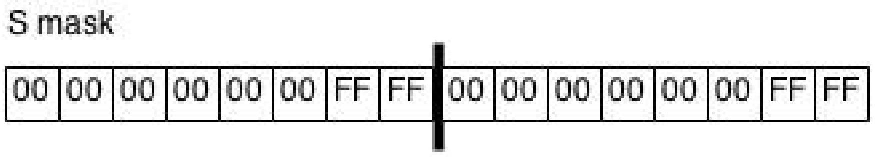

FIGURE 4.2: SSE mask required for the extraction of the sticky bit (s)

The extraction of the *guard* bit (g) is done in a simpler fashion. Logical 'and' between the existing value and the mask shown in Figure 4.3 would result with the value of the *guard* bit (g) at the same position as original. This leads to the extraction of the bit before the guard bit (m). For this a similar mask is required to mask out all the other bits, by having a single bit set as 1 in position of the bit before G.

Figure 4.4 shows the mask required for the extraction of m. This mask is used by applying logical 'and' between itself and the original value. The result of the operation is the m bit to the left of the guard bit. Having the *sticky* bit in position of the *guard* bit, forces a single logical shift to the right of the m bit for it to be in line with the other extracted bits. Logically shifting the m bit to the right once, aligns the extracted bits for the operations required to compute the *rounding bit (b).*

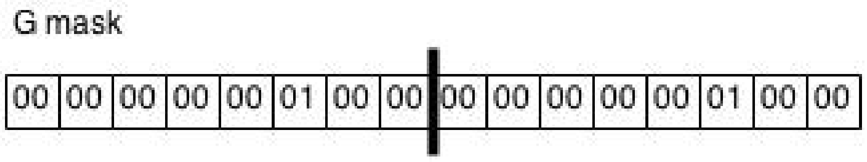

FIGURE 4.3: SSE mask required for the extraction of the guard bit (g)

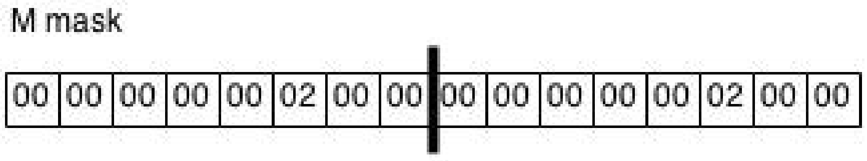

FIGURE 4.4: SSE mask required for the extraction of the bit before the guard bit (m)

In order to compute the *rounding* bit (b), a series of operations are required. All the operations are possible as the bits are aligned in the same bit position. The bit position chosen is the position of the *guard* bit. Firstly, apply logic or between the extracted m and the extracted s. This results in the m or s value. This value is then logically anded with the *guard* bit (g) extracted earlier using its mask. The result of this logical and is the value of the *rounding* bit (b). The operations described for the computation of the *rounding* bit (b) do not affect the positioning of the bit, therefore the resultant bit b is in the same position as the original *guard* bit. The *guard* bit (g) being the last bit of the final result, then the arithmetic addition between the initial value and the resultant bit b is possible. This final addition results in a correct rounded value for both items of the __m128 vector.

Listing 4.6 shows the function described above. The fixed values for each of the masks are derived from the re-interpretation(conversion) of the hexadecimal representation shown in the figures of the masks to their corresponding double type value. The function applies round to nearest, ties to even rounding mode for both of the times in the __m128 vector, keeping the same level of parallelism as the SSE intrinsics.

```
__m128i convert_double_to_f48_SSE (__m128i a) {
  __m128i mask = _mm_set_epi8(15,14,13,12,11,10,255, 255,
        7, 6, 5, 4, 3, 2, 255, 255);
  // preparing final result
  __m128i unrounded_result = _mm_shuffle_epi8(a, mask);

  // extracting sticky bit S
  __m128d s_mask = _mm_set_pd(3.2378592100206092272611435840 6E
    -319,3.23785921002060922726114358406E-319);
  __m128i s = _mm_and_si128(a, (__m128i)s_mask);
  s = (__m128i)_mm_add_pd((__m128d)s, (__m128d)s_mask);

  // extracting bit before guard M
  __m128d m_mask = _mm_set_pd(6.47581723317038670383112248188E
    -319,6.47581723317038670383112248188E-319);
  __m128i m = _mm_and_si128(a, (__m128i)m_mask);
  __m128d shift_count = _mm_set1_pd(1.0);
  m = _mm_srl_epi64(m, (__m128i)shift_count);

  // extracting the guard bit G
```

```
    __m128d g_mask = _mm_set_pd(3.237908616585193351915561240 94E
      -319,3.2379086165851933519155612409 4E-319);
    __m128i g = _mm_and_si128(a, (__m128i)g_mask);

    // computing M|S
    __m128i m_or_s = _mm_or_si128(s,m);
    // computing B
    __m128i b = _mm_and_si128(g,m_or_s);
    // apply rounding bit by addition
    __m128d result = _mm_add_pd((__m128d)unrounded_result, (__m128d)b);
    // adding final padding
    __m128i permute_mask = _mm_set_epi8(15, 14, 13, 12, 11, 10, 255, 255,
                            7,  6,  5,  4,  3,  2, 255, 255);
    return (__m128i)_mm_shuffle_epi8((__m128i)result, permute_mask);
}
```

Listing 4.6: Conversion from standard F64(double) to non-standard F48 using SSE

The above mentioned conversions, both using and not using SSE intrinsics, are being used in the implementation of the representative functions from each level of the BLAS. Both of the different versions of conversions are required for keeping the computation flow as designed and detailed in chapter 3 (Design) section 2.

4.3 Basic Linear Algebra Subroutines

This section will describe in detail the implementation that has been done for each of the selected functions from the levels of the BLAS. The original BLAS [4] along with all the extensions and updates (level 3 addition [6], extensions [5] and sparse extensions [13]) addresses the functionality for each of the levels in BLAS. Each level can be categorised by the type of operations it addressees. Level 1 can be described as addressing scalar and vector(array) operations. Level 2 considers operations on matrix(2D-array) and vector(array). Level 3 addresses matrix-matrix(2D-arrays) functions.

BLAS functions have been chosen in order to evaluate the performance of the non-standard type defined (F48) against the standard double (F64) type. Testing the different level functions allows to answer the efficiency question of the research. BLAS functions offers concrete and complex functions in terms of the operations performed.

The representative functions that have been implemented from each of the BLAS levels are enumerated below. Each of the functions have been implemented using both F48 and the standard double types.

- **Level 1:** Dot product, Magnitude, Scale, Absolute maximum, Absolute minimum (vector functions)
- **Level 2:** Matrix-Vector multiplication
- **Level 3:** Matrix-Matrix multiplication

The implementation of all the representative functions enumerated above is described in details in the following subsections of this section. A subsection is dedicated to the details of the implementation of the F48 loop-unroll which is used in a couple of functions in level 1 and level 2. The loop-unroll is required to minimise the data being written back to memory after the computation. This technique is used where a vector(array) of the newly defined F48 type is required to be stored back to memory.

4.3.1 Loop un-roll

The loop un-roll technique refers to unrolling a loop when using the non-standard F48 type for specific functions in the BLAS library that require the store back to the memory of a vector(array) with elements of the non-standard type. Figure 4.5 below, shows the typical store of two F48 elements from the SSE vector represented on top in the figure, to the memory. The figure describes the overwrite of the two byte padding for each F48 item stored. In the case of storing to the initial array, this would also alter the next values that have not yet been used. To eliminate this side-effect a solution is required as extracting the bits that will be overwritten, saving them and after the overwrite write the bits back to the memory. Another solution to this issue is to extract the bits that will be overwritten and or them with the padding, causing the padding to contain the original data. This approach would still require two data transfers from memory. One for getting the required data and another one to write it back.

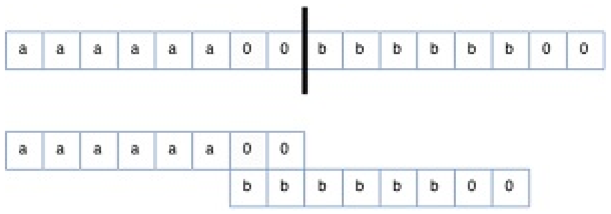

Figure 4.5: Graphical representation of storing to memory an SSE vector with F48 elements.

To counteract these issues, the loop-unroll technique uses eight F48 values and structures them as shown in Figure 4.6 (byte representation of the three SSE vectors required to store eight F48 values). This allows the write to memory to be smoother as the items are aligned to the actual memory address space eliminating the side effects caused by overwriting. This is achieved by permuting values across the SSE vector items boundary, and allowing the cross of values across two items. As it can be seen in the figure below, the two byes of the F48 value ‘b‘ are on the first item of the first SSE vector, and the other four bytes in the second item of the SSE vector. Similarly, the values are spread between two SSE vectors as the ‘c‘ value which is split between the last item of the first SSE vector (4 bytes) and the first item of the second SSE vector (remaining 2 bytes).

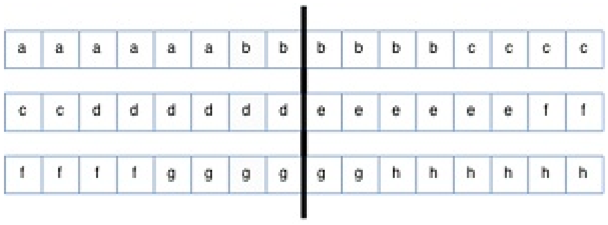
a a a a a a b b b b b b c c c c
c c d d d d d d e e e e e e f f
f f f f g g g g g g h h h h h h

FIGURE 4.6: Graphical representation of the structure of items when using the loop-unrolling technique.

Loop unroll technique offers the possibility to the BLAS function implementations to make less data movement when storing F48 values to the memory in a vector(array).

4.3.2 Level 1

All the functions selected for this level, have been implemented using Intel's SSE intrinsics [7].

Starting with the **dot product** function. This function takes as input two vectors(arrays), computes and returns a single value of the initial type. The formula below describes the operation of the dot product.

$$a \cdot b = \sum_{i=0}^{n} a_i \times b_i$$

By letting A and B to be defined as:

$$A = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, B = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Then the dot product formula expands to the following:

$$A \cdot B = a_0 \times b_0 + a_1 \times b_1 + \ldots + a_n \times b_n$$

The central approach to this function is the same using both types (standard F64 and non-standard F48). In achieving this operation using SSE, a running sum that is initially

0 is required. Each iteration of the loop requires the load of two items from both of the arrays(A and B) in two SSE vectors (for example: in one SSE vector at first time around the loop a_0 and a_1 would be loaded, similarly b_0 and b_1 would be loaded in the other vector). Applying a multiplication operation between the loaded SSE vectors, results in two item multiplied (computing $a_0 \times b_0$ and $a_1 \times b_1$ for example). This temporary result has to be added to the running sum (initially $0 + a_0 \times b_0$ and $0 + a_1 \times b_1$ following the example). Saving this running sum and adding each multiplication result to it would result in an SSE vector that would contain the first item as: $(0 + a_0 \times b_0 + a_2 \times b_2 + \ldots + a_{n-1} \times b_{n-1}$, and the second item as: $0 + a_1 \times b_1 + a_3 \times b_3 + \ldots + a_n \times b_n)$. Final result is then obtained by horizontally adding the items in the running sum vector. This implies the swap of the values between them and addition. The final result being the addition of the first value containing the odd indexed items and the second value of the vector containing the even indexed items.

In addition to the above described approach for computing the dot product using SSE, when implementing this function on the non-standard type there is a need to convert the input at load time as described in the conversions section of this chapter. Having the values converted from the non-standard F48 to the standard double type, the computation described above can be executed, and the final result can be stored to memory after it's conversion back to F48 is completed.

Another function implemented from BLAS level 1 is the **magnitude** of a vector(array). This function requires a single vector(array) as input, and produces a single value of the same type as output. The formula for computing the magnitude of an array is:

$$|A| = \sqrt{\sum_{i=0}^{n} (a_i)^2}$$

Similar to the dot product, the implementation of the magnitude function is the same for both types (standard and non-standard), as the computation required is processed in the same mode. A running sum is required for this case as well as in the dot product described above. Having the running sum starting initially at 0, and continuously adding the squared values of the odd indexed items of the array into the left item of the SSE vector, and the squared values of the even indexed values into the right item of the SSE vector. The running sum after execution of the loop around each element of the initial array, should contain the first item as: $0 + {a_0}^2 + {a_2}^2 + \ldots + {a_{n-1}}^2$ and the second item as $0 + {a_1}^2 + {a_3}^2 + \ldots + {a_n}^2$. Similar to the dot product, a final horizontal add is required to obtain the final value of the running sum (adding the odd indexed and the even indexed square values together). The running sum is then: $0 + {a_0}^2 + {a_1}^2 + {a_2}^2 + \ldots + {a_n}^2$.

In order to get the final value of the magnitude, square root is applied to the final sum resulting in the final result of the function (obtaining $\sqrt{{a_0}^2 + {a_1}^2 + \ldots + {a_n}^2}$). As the dot product, the function implemented to work on the non-standard type, requires the conversion of the values to the standard (doubles) before doing any computation, but also requires the conversion from the final double (F64) result to the non-standard (F48) type.

The **scale** function requires a vector(array) and a scalar as parameters. The result of the function is a vector(array) of the same size as the input, but with each of the elements multiplied (scaled) by the scalar parameter. This operation can be seen as follows.

$$\text{Let: } A = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \text{ and the scalar } S$$

The result of the scale function would be the following vector(array).

$$S \times A = \begin{bmatrix} a_0 \times S \\ a_1 \times S \\ \vdots \\ a_n \times S \end{bmatrix}$$

The implementation of this function for the standard type is a straight-forward multiplication of each two loaded elements of the array in the SSE vector, by the preloaded scalar. Then storing the new values back into the memory at the address of the initial loaded items. However, the non-standard F48 implementation of this function makes use of the loop-unroll technique. This requires the load of eight items of the array at each loop iteration. Converting each of the item to the standard double type at load time, and computing the multiplication for each in turn. Then each of the 4 SSE vectors (4 vectors containing 2 items each) are converted back to the non-standard f48 type by applying the rounding method described in the Conversions section of this chapter. Having the 8 elements in 4 SSE vectors and storing straight to the memory, will overwrite 16 bytes. For each item stored back two bytes will be written twice. For this reason, the loop-unroll technique described in section 4.3.1 of this chapter. Applying this technique removes the overwrite, and compacts the 4 SSE vectors into using only 3 SSE vectors to store the same amount of data (24 bytes - 8 F48 values).

Two functions named **absolute maximum** and **absolute minimum** have been implemented using both the standard type F64 (double) and the newly defined non-standard F48 type. These functions are similar in their structure. The only difference between the absolute maximum and the absolute minimum being the comparison (greater/less) that is being performed on the items. Both of the functions have a vector(array) as input, and one single value is returned as result. The implementation of these functions using SSE requires an initial load of the first two elements of the array. These are to be considered as the maximum / minimum respectively. At every iteration of the loop (loop starting from the third element and incrementing by two items at a time) the next two elements are loaded and the comparison is done between the initial assumed maximum/minimum. By using the SSE intrinsic '_mm_max_pd'/'_mm_min_pd' the maximum/minimum is selected for both of the items in the SSE vector. At the end of the loop, a final comparison is required to detect the final maximum/minimum. Similar to the final horizontal add implemented for the magnitude and dot product, swapping the two values of the SSE vector, and applying the comparison would result in the final result that requires to be returned by the function (or saved back to the memory). As before, the implementation of the F48 version of this function requires the conversion of each loaded value to double, and the final result to be converted back to the non-standard F48 type.

4.3.3 Level 2

The representative function chosen for the second level of BLAS is the **matrix-vector multiplication**. In general this multiplication is defined as taking a matrix and a vector(array) input and resulting a vector return.

$$\text{Let matrix } A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,0} & a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \text{ and the vector } B = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$

$$A \times B = \begin{bmatrix} (a_{0,0} \times b_0) + (a_{0,1} \times b_1) + \ldots + (a_{0,n} \times b_n) \\ (a_{1,0} \times b_0) + (a_{1,1} \times b_1) + \ldots + (a_{1,n} \times b_n) \\ \vdots \\ (a_{m,0} \times b_0) + (a_{m,1} \times b_1) + \ldots + (a_{m,n} \times b_n) \end{bmatrix}$$

For the implementation of this function, I have experimented with different approach. The first approach considered was without using SSE. This has been a naive implementation of the function, computing a running sum for each of the rows in matrix A. The running sum comprised of computing the multiplication between each of the elements and adding to the existing running sum (which was initially set to 0). Finally, once a row of the matrix has been evaluated store the final running sum to the index of the vector(array) corresponding to the row of the matrix.

Two other implementations approaches have been considered, both of which are making use of the SSE intrinsics. The first version, is an SSE version of the above described approach. This involves the load of two values from a row of the matrix at each iteration. Also at each iteration to corresponding two values from the vector(array) are loaded. Loading these values into two separate SSE vectors allows the multiplication to be performed. As an example, loading $a_{0,0}, a_{0,1}$ into one SSE vector and b_0, b_1 in another SSE vector, then in turn computing $a_{0,0} \times b_0, a_{0,1} \times b_1$. Computing a running sum of these values and at the end of traversing a row of the matrix using the technique described in the level 1 functions of horizontal add by swapping the values to compute the final sum for that row. Storing this value as being the first computed value of the result, and repeating the process for each of the rows of the matrix. The size of the vector resultant is the same as the 'm' size of the input matrix. The F48 implementation is similar to the described approach in this version with the addition of the conversions from F48 to double at load time. Computation happening in the standard double type. Before storing the values back to the memory, each of the double type sums are converted in turn back to F48.

The second approach, applies a similar approach as the first version described above. Version two makes use of loading four items from the matrix and two from the vector, improving locality and also computing four partial results at a time. Using the notations above for example. This approach loads the following items from the matrix: $a_{0,0}, a_{0,1}$ (the same as the first approach), and $a_{1,0}, a_{1,1}$ (from the second row of the matrix). These values are loaded in SSE vectors similarly to the first approach. The values loaded from the vector are the same as in the first approach b_0, b_1. Using these values, this approach can compute the following: $a_{0,0} \times b_0, a_{0,1} \times b_1$ (the same as the first approach) but also $a_{1,0} \times b_0, a_{1,1} \times b_1$. The procedure of computing the running sum and finally the sum that is the final result for the respective row is the same as in version one. However, in this version there are two different running sums (one for each of the rows loaded initially). Once both of the final sums are computed, this approach makes use of the permutation abilities provided by SSE intrinsics and merges the two final results into one SSE vector. This allows the store of the SSE vector into memory, storing the two final results in one store operation.

The above version two describes the approach implemented for the double type use of the function. However, the non-standard F48 type use of the function exploits the loop unroll technique defined in section 4.3.1 of this chapter. As explained in detail in section 4.3.1, loop unroll technique requires eight elements to work with. For this, version two of the implementation using the non-standard F48 type eight pairs of elements are loaded from the matrix (for example from the first eight rows of the matrix load the first two values). However, from the vector there is no other required values to be loaded other than the ones loaded in the version two working on the standard F64 (double) type. This means that only two values are loaded from the vector and used in conjunction with all the eight values loaded from the matrix. The computation is similar to the version two for use on the standard types. The main difference being that in this case, there are eight running sums being computed, therefore at each completed run across the rows of the matrix there are eight final results computed. These results are then being converted back to the non-standard F48 type, aligned according to the loop unroll technique and stored in the final vector result.

4.3.4 Level 3

Level 3 of the BLAS provides details of functions that operate on matrices. The representative function selected for this level is the **matrix-matrix multiplication**. Matrix-Matrix multiplication requires two matrices as input and produces another matrix as the result. This function has been implemented in two ways. Firstly without using the Intel's SSE intrinsics and the second version makes use of the SSE vectors and intrinsics.

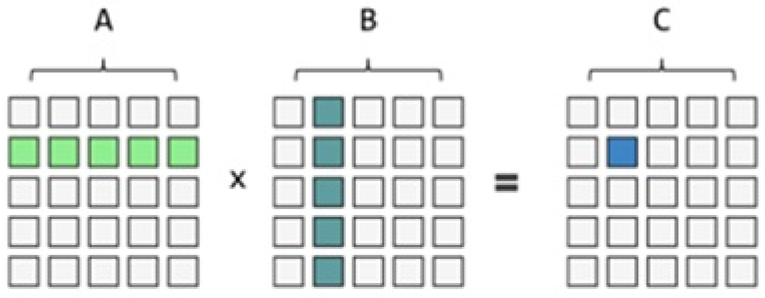

FIGURE 4.7: Graphical representation of the computation of an item in the result of a matrix-matrix multiplication.

Both non-standard (F48) and standard F64 (double) types of implementation are naive implementations of the function, without making use of any locality improvement. Therefore both SSE implementations (F48 type and the standard double type) are based on the same principle. The implementation creates a running sum (similar to the ones used in the implementation of matrix-vector multiplication - level 2) for each element of the result matrix. Each running sum is stored in a temporary result matrix at the position where the final sum has to be placed in the final matrix. At each pass the multiplication of the items is computed and added to the corresponding running sum (initially 0).

A final pass is required to finalise the addition of the running sums. This applies the same approach as the horizontal addition described in the level two, and passes through all the saved SSE vectors as partial sums for the elements where the values have been computed. Once the final addition is done, the resultant value is converted to F48 (if required) and stored back to memory accordingly.

Chapter 5

Experiments and Evaluation

In order to be able to answer the thesis question, I have conducted a number of experiments. These experiments mainly consist in evaluating the time required to compute the implemented BLAS functions using both types (the standard F64 double and the non-standard F48). A setup was required for the experiments which is described in details in the section below. This allowed to time accurately the run of the functions on different vector/matrix sizes. This chapter focuses on the setup that allows the running of experiments and also with the analysis of the results and interpretation of the timings.

All the experiments described in this section have been run on the Intel Core i5-3450 processor (ivy bridge architecture). The processor has a base frequency of 3.1GHz with the maximum Turbo frequency reaching 3.5GHz.

5.1 Experiment setup

The experiments that are to be ran, in order to investigate towards the answer to the question of performance when using the non-standard F48 type compared to the standard F64 (double) type, require a precise setup. The setup that has been put in place is based on Paoloni's publication on How to benchmark code execution times (2010) [14].

As the BLAS functions implemented are considered as being small programs (subroutines) that would not take a long time to execute, then the measurement of time required for the computation needs to be a lot more accurate and precise. In order to obtain most stable results the processor has to be locked to it's base frequency (3.1GHz) by disabling turbo and also the Dynamic Voltage and Frequency Scaling (DVFS). All the processes that are running on the machine have to be killed including the graphical interfaces so that the processor is clear to use a core dedicated to the running of the benchmarks.

Having this setup in place, in order to run the benchmarks and record the timings of each of the functions, I have made use of the Intel's Read Time-Stamp Counter and Processor ID IA assembly instruction RDTSC [15] to get the time-stamp of the processor before the function (that has to be timed) call and getting the same RDTSC time-stamp at the end of the execution of the function. This allows the computation of the time needed for execution of the function in the number of cycles by subtracting the timestamps. Listing 5.1 shows RDTSC usage for the calculation of the timings for each of the function.

For further investigation that was pursued in order to find a correlation between the slow-down of some functions, the Intel Performance Counter Monitor (PCM) tool [16] has been used to gather information about the number of instructions executed per cycle, number of cache hits etc.

```
#include <clocks.h>
uint64_t start, stop, diff;
for (int i = 1; i < runs+1; i++) {
  start = clocks();
    benchmark(a, b);
    stop = clocks();
    diff = stop - start;
}
```

Listing 5.1: RDTSC usage example

5.2 Experiment results and analysis

Each of the functions described in Chapter 4 (implementation) a series of tests have been run in order to measure the number of cycles that it takes to execute. Each of the tests has been run 100 times to be able to get a more stable result and confirm that the result is stable enough to be considered.

In the following subsections, all the results of the tests grouped by the level of BLAS are analysed and interpreted.

5.2.1 BLAS level 1

The functions implemented in this level of BLAS are *a*) dot product, *b*) magnitude, *c*) scale, and *d*) absolute maximum/minimum.

The most representative function for this level is the '**dot product**'. Looking at figure 5.1 can be seen that the timings over 100 runs are stable and provide an accurate measurement of the number of cycles required for each of the readings. For each size of the input vector histograms have been generated across all the functions implemented in BLAS level 1.

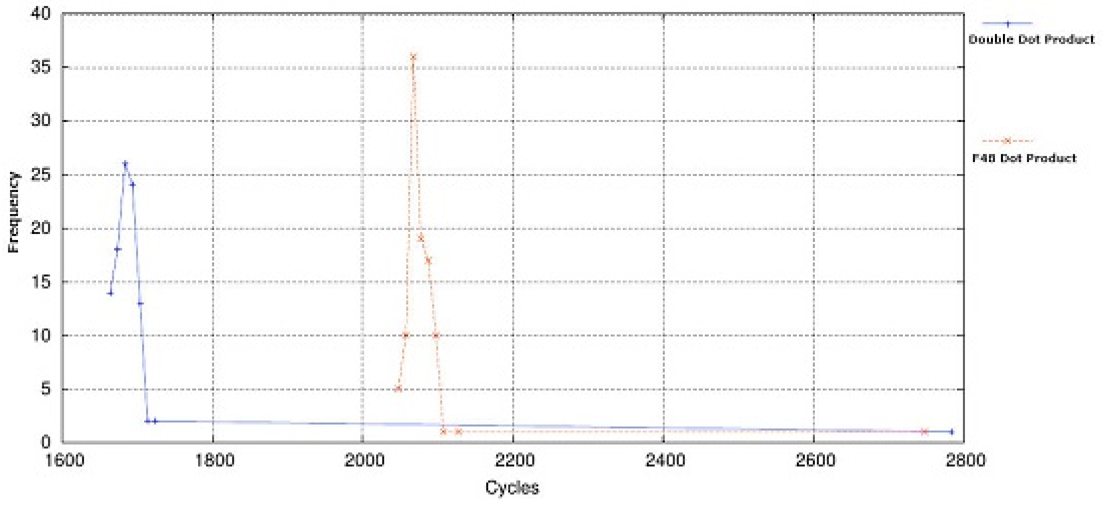

FIGURE 5.1: Histogram of the dot product timings for size 1024.

Having stabilised the results, then the mean of the number of cycles out of 100 runs can be used to be compared against the corresponding value on the standard type.

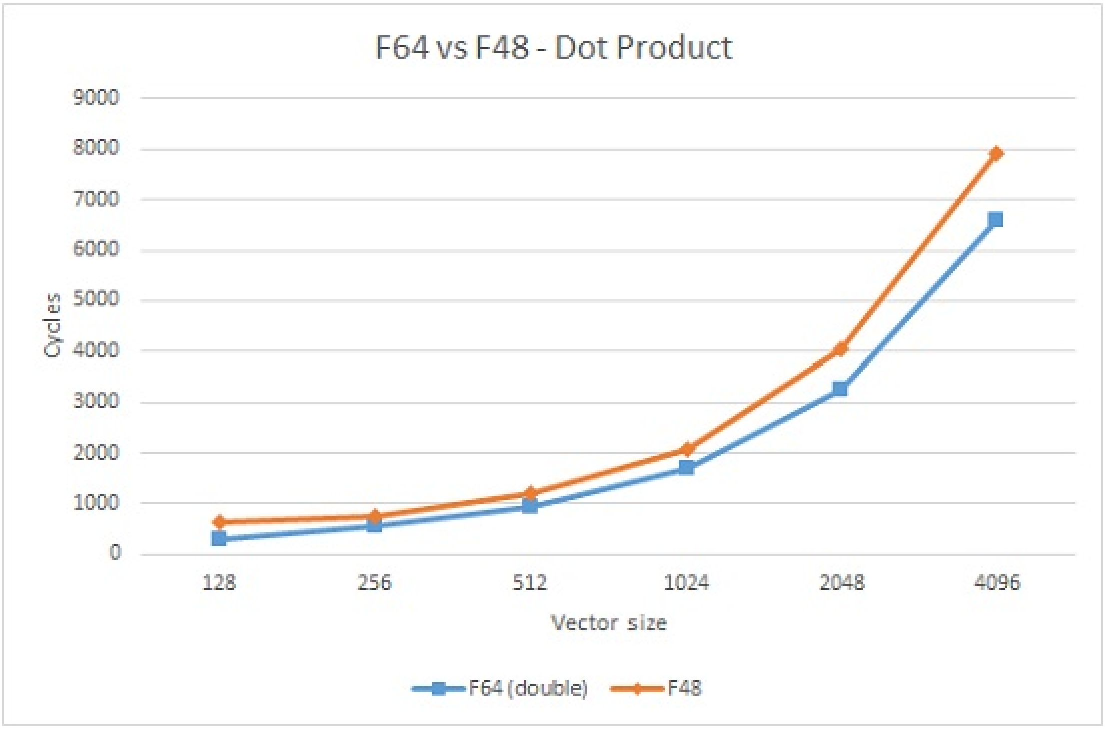

FIGURE 5.2: Comparison between the non-standard F48 and the standard F64 types across different sizes of the vector in computing the dot-product of a vector.

The figure 5.2 above shows a clear comparison between the number of cycles required to compute the dot product using the standard double-F64 type (blue line) against the non-standard F48 type (orange line). This figure is based upon table 5.1. In the table, a speed-up row is present, calculating the time observed for F64 divided by the time observed for F48. The closer this speed-up value is to 1, the closer the timings are together. A lower than 1 value denotes a slow-down of F48 compared to F64. However, a higher value than 1 represents an actuall speed-up from the F64. In this figure (5.2) the lower the line the faster the computation is. Clearly can be observed that the F48 is performing slightly slower than the standard F64 (double) in the case of dot product.

	Vector size					
	128	**256**	**512**	**1024**	**2048**	**4096**
F64 (double)	293.2	576.4444	956.6869	1696.8	3257.96	6607.03
F48	631	756.404	1209.818	2082.1	4058.242	7923.529
Speed-up	**0.464659**	**0.762085**	**0.790769**	**0.814946**	**0.802801**	**0.833849**

TABLE 5.1: Dot product timing results and speed-up

Similarly, the stabilisation of the processor usage has been done for all the testings, therefore the following test results will not include the histogram of spread of the timing over 100 runs, as the results are stable and the mean can be used to interpret as the average number of clock cycles required.

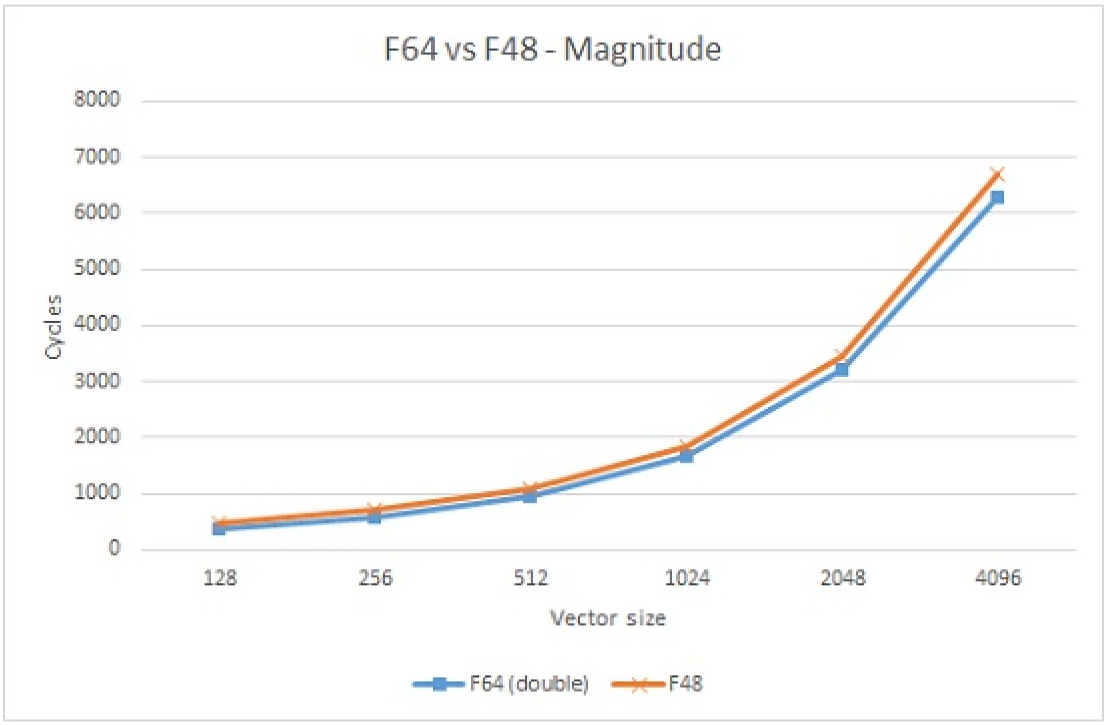

FIGURE 5.3: Comparison between the non-standard F48 and the standard F64 types across different sizes of the vector in computing the magnitude of a vector.

Another BLAS level 1 function that has been implemented for testing the F48 type performance against F64 (double) type is the ‘**magnitude**’. Figure 5.3 shows a comparison between the F64 (double) type timing and the non-standard F48 type timing. The lower values are faster, however the F48 (orange line) is very close to the standard (blue line). Table 5.2 shows the actual results of the timings and the speed-up row discussed previously. The speed-up value is seen in figure 5.3 as it is closer to 1, and the lines in the figure are very close.

	Vector size					
	128	**256**	**512**	**1024**	**2048**	**4096**
F64 (double)	380.08	572.88	954.44	1683.88	3217.4	6303.2
F48	478.16	711.96	1102.44	1830.02	3453.6	6683.2
Speed-up	**0.79488**	**0.804652**	**0.865752**	**0.920143**	**0.931608**	**0.943141**

TABLE 5.2: Magnitude timing results and speed-up

Similarly the analysis of the timings for the ‘**scale**’ function of BLAS level 1 has been conducted. Table 5.3 shows the results of the timings along with the speed-up value (as

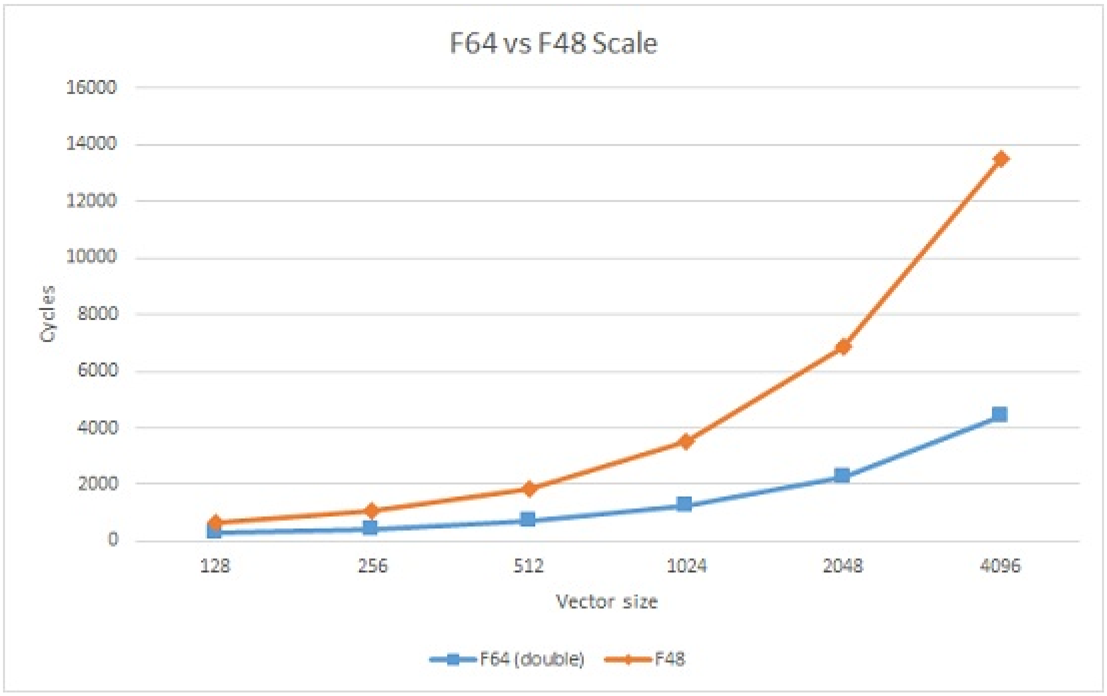

FIGURE 5.4: Comparison between the non-standard F48 and the standard F64 types across different sizes of the vector in computing the scale of a vector.

described above). The difference between the F48 performance and the F64 (double) as can be seen in figure 5.4. This requires further investigation that is described in this section below.

	Vector size					
	128	**256**	**512**	**1024**	**2048**	**4096**
F64 (double)	293.2	428.4	684.76	1247.8	2272.1	4389
F48	631	1035.44	1832.76	3537.8	6843.55	13521.8
Speed-up	**0.464659**	**0.413737**	**0.373622**	**0.352705**	**0.332006**	**0.324587**

TABLE 5.3: Scale timing results and speed-up

The timings for the '**absolute maximum**' and '**absolute minimum**' on both the types (standard F64 - double and non-standard F48) are shown in table 5.4. Based on this results table, figure 5.5 has been created. The comparison between the absolute maximum on F64 type (blue line) and the absolute maximum on the non-standard F48 type (orange line) shows that the F48 performs slower. Similar in computing the absolute minimum the F48 type (yellow line) performs slower than the standard F64 type (grey line).

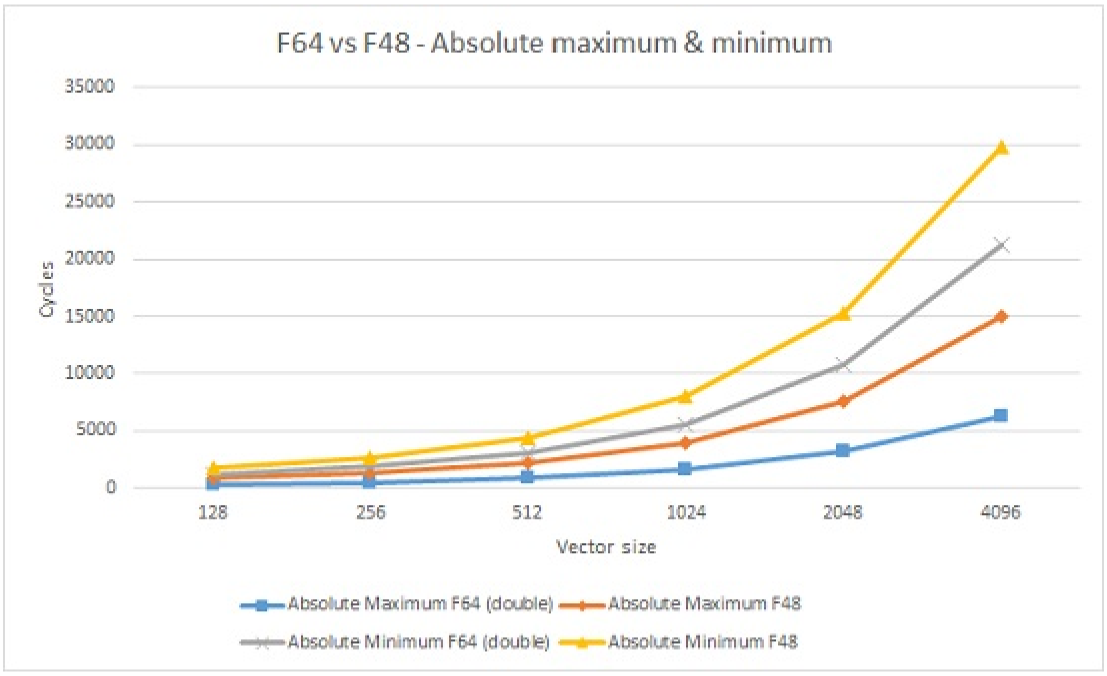

FIGURE 5.5: Comparison between the non-standard F48 and the standard F64 types across different sizes of the vector in computing the absolute maximum and minimum of a vector.

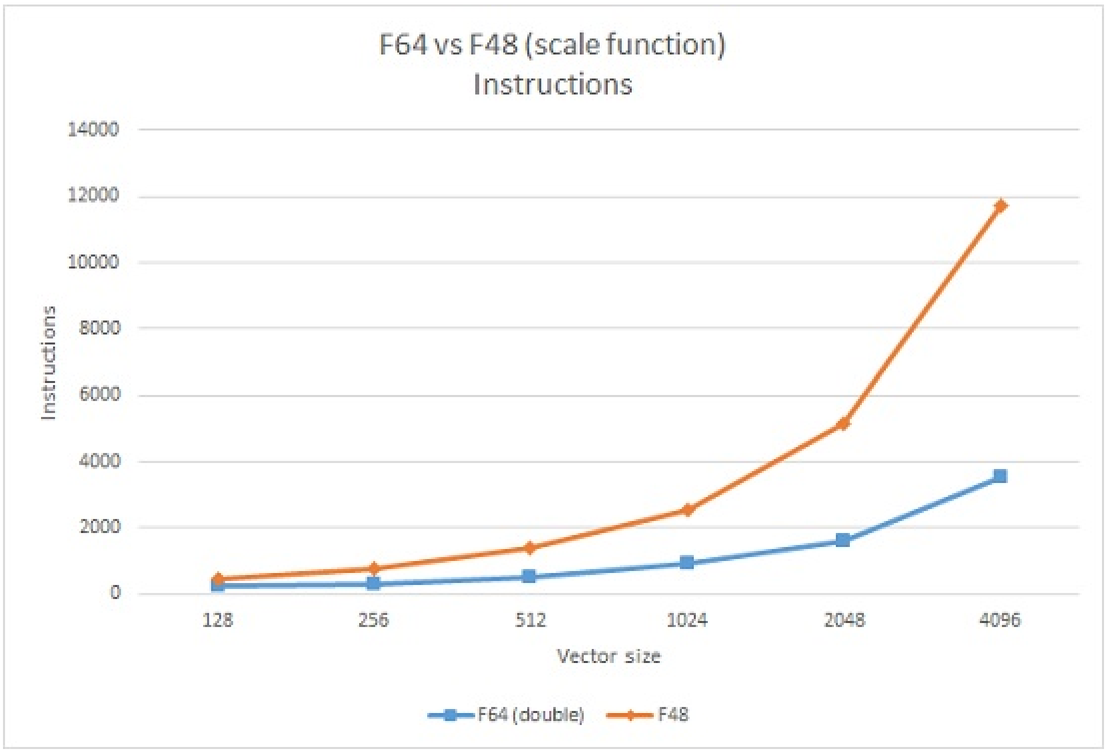

FIGURE 5.6: Graphical representation of the number of instructions performed in both F64 and F48 when computing the scale function.

		Vector size					
		128	**256**	**512**	**1024**	**2048**	**4096**
Absolute maximum	**F64 (double)**	344.76	541.8	915.2	1657.6	3203.2	6268.2
	F48	515.48	800.8	1282	2322.2	4439.152	8671.5
	Speed-up	**0.668814**	**0.676573**	**0.713885**	**0.713806**	**0.721579**	**0.722851**
Absolute minimum	**F64 (double)**	350.56	536.6	928.6	1650	3204.6	6273.5
	F48	579.72	807.8	1335.6	2356.4	4450.8	8658.2
	Speed-up	**0.604706**	**0.664273**	**0.695268**	**0.700221**	**0.720005**	**0.724573**

Table 5.4: Absolute maximum & minimum results and their corresponding speed-up

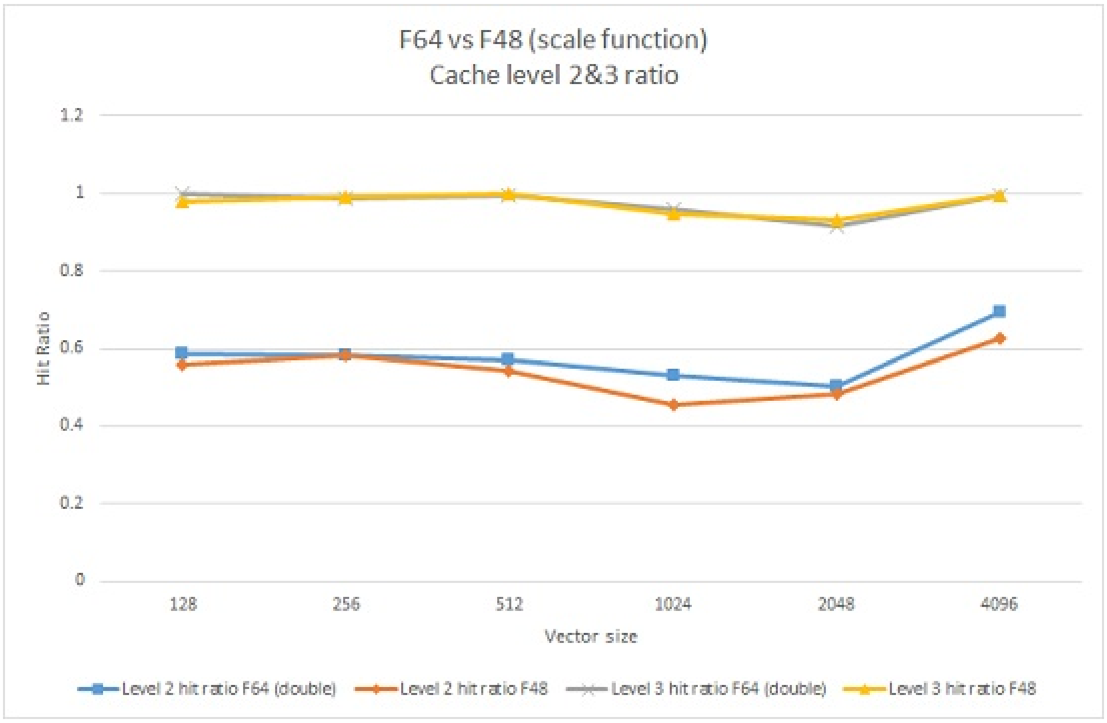

FIGURE 5.7: Graphical representation of the hit ratio of level 2 and 3 cache in both F64 and F48 when computing the scale function.

I have pursued Further investigation in order to examine a relationship between the slowness of the scaling function on the F48 type in relation with total number of instructions, level 2 cache hit ratio and level 3 cache hit ratio.

Figure 5.6 shows the total number of instructions executed on both (F64 and F48 types) when computing the scale function. The total number of instructions is derived from the instructions per cycle multiplied by the average cycle. The instructions per cycle(IPC) is observed using the Intel's PCM tool [16]. There is a correlation between the number of instructions executed by the F48 type and it's slowdown shown in figure 5.4. However, the level 2 and level 3 cache hit ratio shown figure 5.7 do not provide enough support in reasoning the cause of the slow-down of the scale function.

5.2.2 BLAS level 2

The function I have considered representative for BLAS level 2 is matrix-vector multiplication. Initially the tests I have ran on this function were using implementations without SSE, and the results were as shown in figure 5.8. Inspecting the number of cycles required to perform the matrix-vector multiplication without the use of SSE, the focus of the implementation has changed towards the full use of SSE therefore the two different approaches both of which explained in details in Chapter 4.

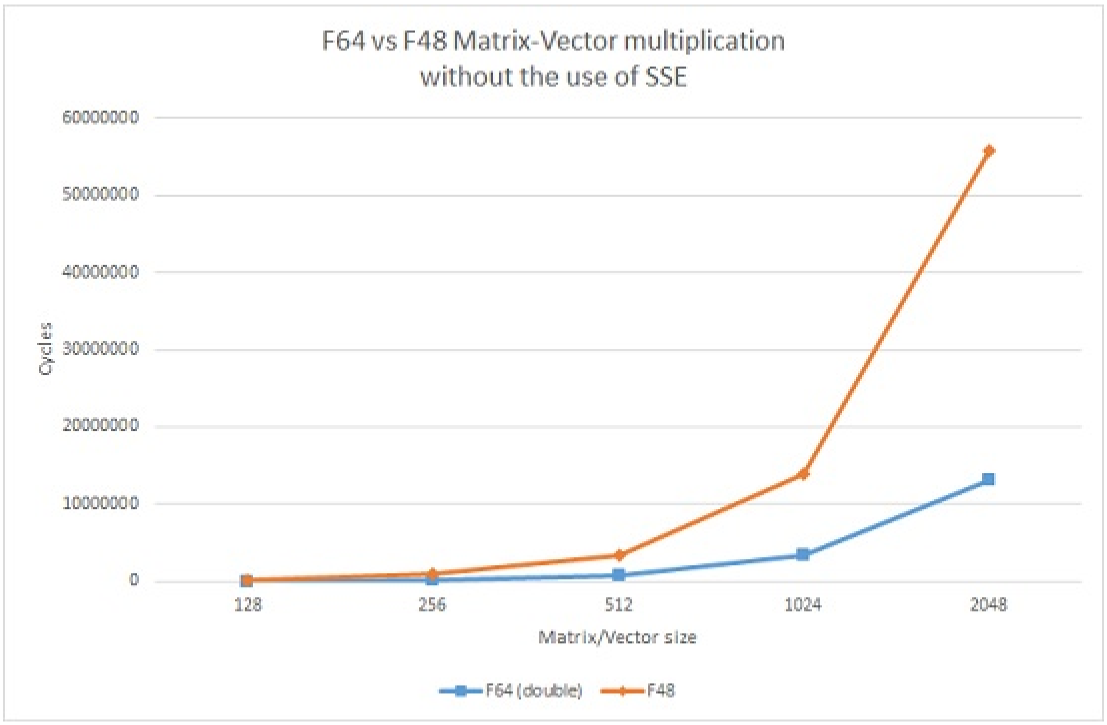

FIGURE 5.8: Graphical representation of the timings of the matrix-vector multiplication function without the use of SSE intrisics.

As described in chapter 4(Implementation) there have been implemented two different versions of the function using SSE. Table 5.5 shows timings of the run of the matrix-vector multiplication. The matrix size is defined as $x \times x$, where x is one of the sizes in the table (for example: 128 size of the vector and the matrix size is: 128×128).

Figure 5.9 shows the graphical representation of the results. Version 1 of the implementation is performing slower when used over the non-standard F48 type. However, when looking at the version 2 implementations, F48 is comparative more performant than the standard F64 type. For a better visualisation of the results, the number of cycles counted for completion of the function has been divided by the total number of elements. This is only reflected in figure 5.9. The actual cycles are shown in the table 5.5.

	Matrix/Vector size				
	128	**256**	**512**	**1024**	**2048**
F64 (double) - v1	27656	105864	408066	2164836	8354452
F48 - v1	35284	132122	497978	2054976	8539432
Speed-up	**0.783811**	**0.801259**	**0.819446**	**1.05346**	**0.978338**
F64 (double) - v2	16532	69794	265252	1827224	7633828
F48 - v2 (unrolled)	15624	62228	232936	1139860	6441200
Speed-up	**1.058116**	**1.121585**	**1.138733**	**1.603025**	**1.185156**

TABLE 5.5: Matrix-vector multiplication results on different matrix/vector sizes across both implementations and types (F64 & F48)

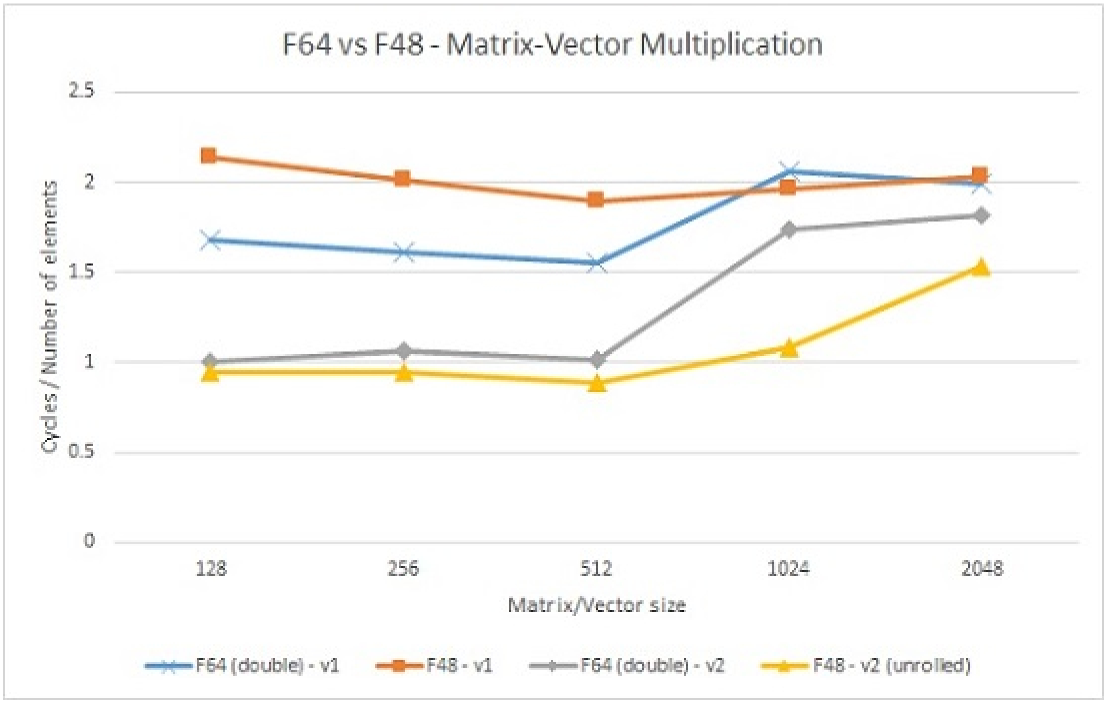

FIGURE 5.9: Graphical representation of the timings of the matrix-vector multiplication function.

5.2.3 BLAS level 3

For BLAS level 3, I have selected the matrix-matrix multiplication function. Figure 5.10 shows a graphical representation of the timing results from table 5.6. The values plotted on the graphical representation in the figure 5.10 are divided by the total number of elements. This allows for a better graphical representation. The size of both the matrices is defined as: $x \times x$ where x is one of the sizes in the table (for example on the column size 128, both the matrices are of the size: 128×128 with a total of 16384 elements in the matrix).

	Matrix size			
	128	**256**	**512**	**1024**
F64 (double)	3639847.16	28283522.4	220551085	2086150393
F48	5068690.56	37562567	285406534.7	2301423409
Speed-up	**0.718104038**	**0.75297097**	**0.772761161**	**0.906460925**

Table 5.6: Matrix-matrix multiplication results on different matrix sizes across both types (F64 & F48)

Interpreting the results, the F48 type (orange line) performs slower than the standard F64 (double) type (blue line). This requires some further investigation by inspecting the instructions per cycle (IPC) using Intel's PCM tool [16] and deriving the total number of instructions. Figure 5.11 shows the number of instructions of both types for each of the sizes timed. This shows that there is a relationship between the slowdown of the F48 (in comparison with F64) and the number of instructions executed. However figure 5.12 correlates the slowdown of the F48 implementation of matrix-matrix multiplication to the level 2 cache hit ratio. Level 2 hit ratio of the F48 (orange line in figure 5.12) is a lot lower than the standard F64 (blue line in the same figure 5.12). The higher the line in the figure, the better the hit ratio, meaning that more hits into the respective cache are happening.

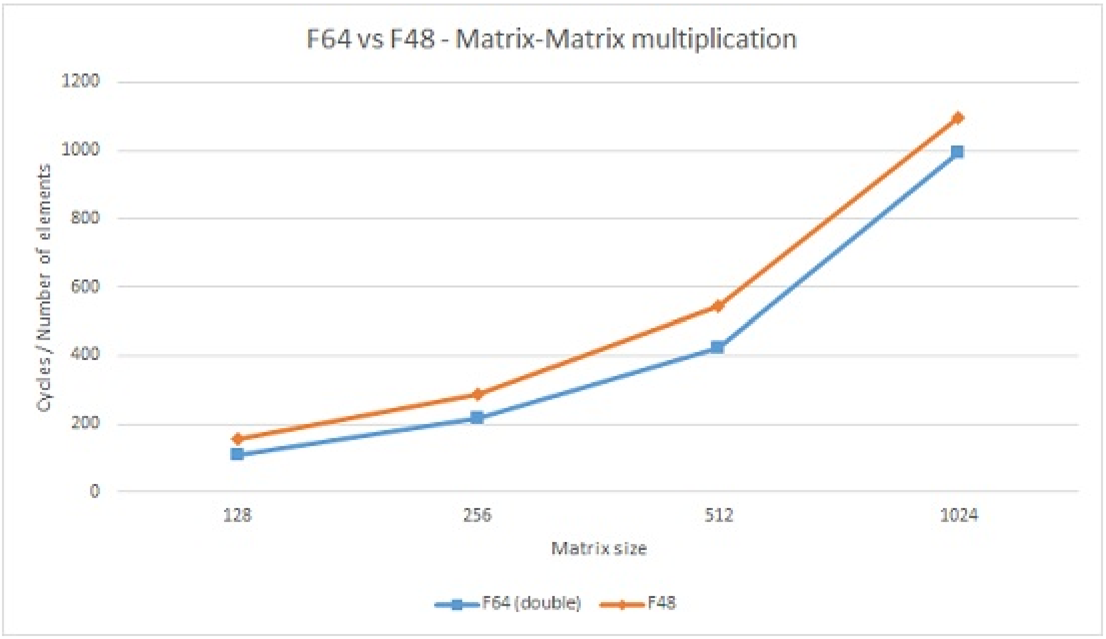

Figure 5.10: Graphical representation of the timings of the matrix-matrix multiplication function.

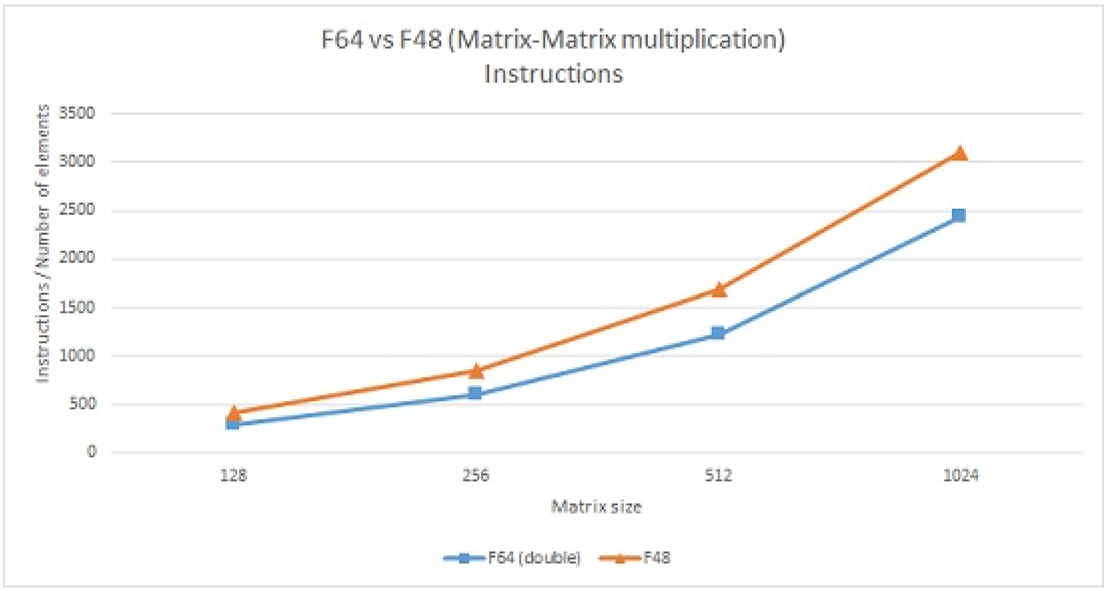

FIGURE 5.11: Graphical representation of the number of instructions on the matrix-matrix multiplication function.

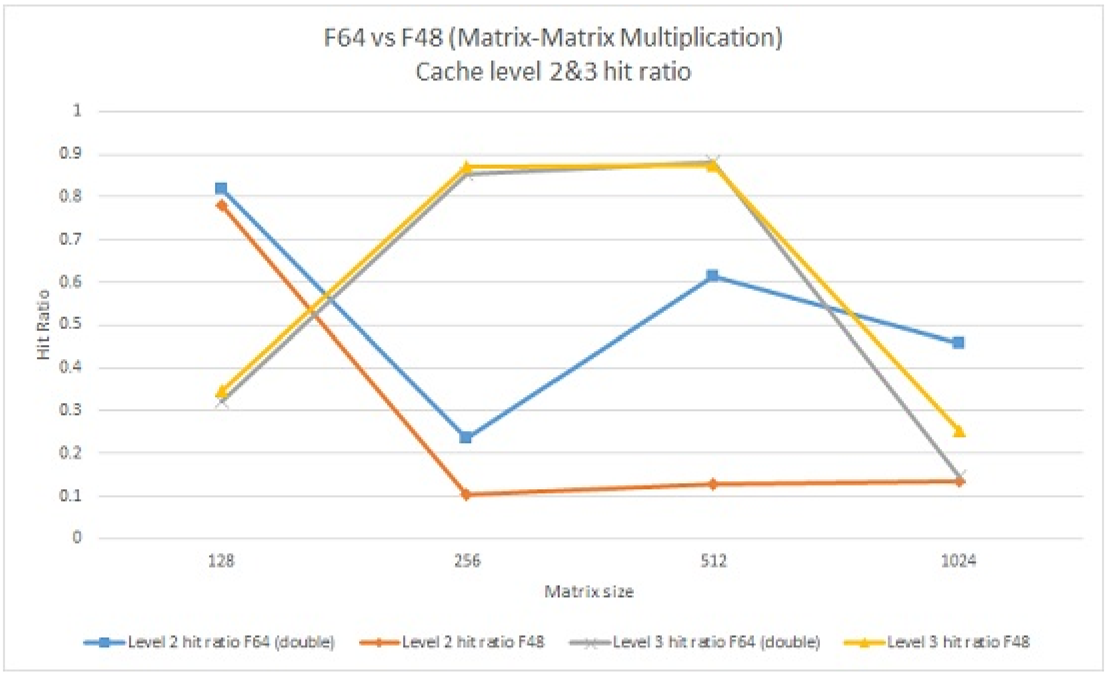

FIGURE 5.12: Graphical representation of the hit ratio of level 2 and 3 cache in both F64 and F48 when computing matrix-matrix multiplication function.

Chapter 6

Conclusions

This chapter would present a general assessment of the experimentation results and a general conclusion with respect to the initial research question presented in chapter 1 section 2.

6.1 Assessment

The assessment of the results shown in chapter 5 is based on the findings(timings) from experimentation. The experiments conducted provide a solid comparison between the standard F64 (double) type and the newly defined non-standard F48 type.

In the functions of BLAS level 1, the F48 type in terms of performance is ranging from 6% to 70% slowdown compared to the standard F64 (double) type. From the experiments using the functions implemented within this thesis, BLAS level 1 scale function shows the slowest F48 at a vector size of 4096.

However, BLAS level 2 function, matrix-vector multiplication shows that different approaches to the implementation of the same function can be advantageous for one type and not for the other. The two approaches implemented using both types (F64 and F48) vary in results. The first version (v1) starting with a 30% slowdown compared to the standard F64 on the matrix size of 128×128 and vector size of 128. Increasing it's performance with size, getting to a matrix size of 2048×2048 and vector size of 2048 to only have a slowdown of 3%. However, version 2 implementation starts with a 0.05% speed-up and reaches 60% speedup on the matrix size of 1024×1024 and vector size of 1024.

Inspecting the results of BLAS level 3, the non-standard F48 type is performing slower (30% slower decreasing with size to 10% slow-down) when compared to the standard F64 (double).

Overall there is a difference between the levels due to the read/writes proportions of the functions in each level. For example the dot product is reading all the items and produces a single value that requires to be written back. On the other side, the matrix-vector multiplication and it's two different implementation approaches (see chapter 4) shows a difference where the number of reads is reduced and the number of writes is the same, but the performance increases. This concludes that the implementation that improves locality and reuse of that will perform faster using F48 type due to less memory bandwidth and data movement.

6.2 General conclusion

The general conclusion of the use of non-standard floating point type (such as F48 used in this research) is that F48's performance is competitive with F64. A good result for a floating point type that is not supported in hardware and it is only being emulated.

F48 type to be used comes with a trade-off between accuracy of the floating point representation ($\approx$5 decimal digits less), memory usage (16 bits less than the standard double — 25% less memory usage) and performance (which is shown to be fluctuating between speedup and slowdown - function and implementation dependant). In order to use the F48 type one must consider the need of overhead in the implementation. Therefore it's usage requires attention to implementation details.

Finally, to answer the thesis question, the F48 type considered as the non-standard floating point is competitive with the standard double (64-bit floating point) type given it is not supported in hardware. Generally the performance of the non-standard type could be considered of the same as the standard type.

6.3 Future work

As the experiments have reflected some fluctuation in the consistency of the results, further investigation in finding the reasons why the final results of F48 have been potentially slower and what has been causing it. Further investigation in the BLAS level 1 functions should be considered.

For the level 2 and 3 of the BLAS, more experiments should be conducted with higher matrix and vector sizes. This will allow the evaluation of current results and allow for better performance predictions on different sizes.

An improved level 3 of the BLAS implementation (making use of the technique of locality improvement and other implementation specific techniques) could be developed to make a comparison similar to the level 2 versions implementations.

Bibliography

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi: 10.1109/IEEESTD.2008.4610935.

[2] Guillaume Revy. *Implementation of binary floating-point arithmetic on embedded integer processors — Polynomial evaluation-based algorithms and certified code generation.* Theses, Université de Lyon ; Ecole normale supérieure de lyon - ENS LYON, December 2009. URL `https://tel.archives-ouvertes.fr/tel-00469661`.

[3] Jean-Pierre Deschamps, Géry Jean Antoine Bioul, and Gustavo D Sutter. Floating-point unit. *Synthesis of Arithmetic Circuits: FPGA, ASIC, and Embedded Systems*, pages 513–548, 2006.

[4] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[5] S Hammarling, J Dongarra, J Du Croz, and R Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–32, 1988.

[6] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[7] Intel Intel. Sse4 programming reference. *Intel Corporation*, 2007.

[8] Intel intrinsics guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. Accessed: 2014-11-17.

[9] Telemetry Standard. Rcc documents 106-07. *Telemetry Group, Sep*, 2007.

[10] MP Halbert and SM Bose. Design approach for a vlsi self-checking mil-std-1750a microprocessor. In *International Symposium on Fault Tolerant-Computing, Kissemmee, USA*, 1984.

[11] Performance benefits of half precision floats. `https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats`. Accessed: 2015-03-22.

[12] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, 2011.

[13] David S Dodson, Roger G Grimes, and John G Lewis. Sparse extensions to the fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 17(2):253–263, 1991.

[14] Gabriele Paoloni. How to benchmark code execution times on intel IA-32 and IA-64 instruction set architectures. *Intel Corporation, September*, 2010.

[15] Intel Coorporation. Using the rdtsc instruction for performance monitoring. *Technical report, Intel Corporation*, page 22, 1997.

[16] Intel performance counter monitor — a better way to measure CPU utilization. `http://www.intel.com/software/pcm`. Accessed: 2015-04-23.