POLITECNICO DI MILANO
Corso di Laurea magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Precision Tuning of Mathematically Intensive Programs: a comparative study between fixed point and floating point representations

Relatore : *Giovanni Agosta*
Correlatori : *Daniele Cattaneo*
Stefano Cherubin

Tesi di Laurea di:
Gabriele Magnani
Matricola 905110

Anno Accademico 2020-2021

Contents

Abstract

The demand of computational power on every kind of device, from low-cost processors to High Performance Computing, is continuously growing. Approximate computing has seen significant interest as a design philosophy oriented to performance and energy efficiency. In precision tuning, accuracy is traded in favor of performance and energy by employing less precise data types, such as fixed-point instead of floating-point. Thanks to the advantages that it brings, precision tuning is increasingly being used in many application areas, exploiting multiple data types for number representation, each one with its own strengths. However, the current state-of-the-art does not consider the possibility of optimizing mathematical functions whose computation is usually offloaded to a library. In this work, we extend a precision-tuning framework to perform tuning of trigonometric functions. We developed a new kind of mathematical function library, which is parameterizable at compile-time depending on the data type and works natively in the fixed point numeric representation. Our approach, which we test on two microcontrollers with different architectures, achieves a speedup of up to 282%, and energy savings up to 60%, with a negligible cost in terms of error in the results.

We also extend a precision-tuning framework mixing precision capabilities to support new data types. This was possible developing a new algorithm to choose which data type allocate and changing the generation procedures. We tested it against the Polybench benchmark suite to prove the proposed solution's effectiveness.

Sommario

La richiesta di capacità di elaborazione dei dati sta cresendo su ogni tipo di dispositivo siano processori a basso costo o grandi centri di elaborazione, l'approximate computing sta suscitando sempre più interesse come paradigma di progettazione orientato sia al risparmio energetico sia alle prestazioni. Nel precision tuning, l'accuratezza della computazione è ridotta in favore di maggiori prestazioni e minor consumi. Tutto questo è possibile grazie alla possibilità di cambiare i tipi dei dati utilizzati, pre esempio da floating-point a fixed-point. Per queste ragioni il precision tuning sta venendo sempre più adottato, per poter sfruttare a pieno tutti i tipi di rappresentazione possibili e tutti i loro punti di forza. Lo stato dell'arte però non considera la possibilità di ottimizzare le librerie matematiche che spesso vengono calcolate da librerie esterne. In questo lavoro abbiamo esteso un framework per il precision tuning affinchè fosse ingrado di calcolare nativamente le funzione trigonometriche. Abbiamo quindi sviluppato un nuovo tipo di approccio alle librerie matematiche, totalmente parametrizzabile, capace di lavorare a compile-time generando codice pensato appositamente per i fixed point. Il nostro approccio è stato testato su due micocontrollori differenti aventi differenti tipi di architetture. Lo speedup massimo ha raggiunto i 282% con un rispormio energetico del 60% e trascurabili costi in termini di errore generato.

Abbiamo inoltre esteso lo stesso framework di precision tuning per supportare nuovi tipi di dato. Per raggiungere questo obbiettivo abbiamo modificato gli algorittmi di selezione e di generazione del codice. Infine abbiamo testato l'efficacia di questi algoritmi sulla suite Polybench.

Ringraziamenti

Vorrei ringraziare tutte le persone che mi sono sono state vicine e mi hanno aiutato durante questi anni di studi presso il politecnico. Un particolare ringraziamento va alla mia famiglia che mi ha supportato durante tutti questi anni di studio. A Silvia la mia compagna che mi è sempre stata vicino nonostante tutte le difficoltà. Ai miei compagni di infanzia Andre e Stephanie che nonostante abbiamo scelto strade differenti siamo riusciti a rimanere sempre in contatto. Ai miei amici conosciuti durante questo percorso che hanno aiutato a rendere unica la mia permanenza al politecnico. Infine ringrazio il prof. Giovanni Agosta, il Dott. Stefano Cherubin, il Dott. Daniele Cattaneo, il Dott. Michele Chiari e tutti i membri del team TAFFO, che nonostante questo anno difficile e pieno di complicazioni sono riusciti tramite la loro passione e la loro ineguagliabili competenze mi hanno permesso di completare la stesura della tesi.

Chapter 1

Basic Notions

In this section, we provide basic knowledge required to better understand the thesis. First, we will describe the primary data types utilized and the advantages and disadvantages of each one. Afterwards, some background about the algorithms used will be provided.

1.1 Real number representations

Due to the boundlessness of real numbers, it is impossible to have a bijective function that maps $\mathbb{R}$ to a finite number of bits. In fact, given n bits to represent a number, and knowing that all the possible combinations of 2 elements in n space are 2^n, there is no way to find such bijective function. For this reason, real numbers have various different representations, each one with a different purpose.

1.1.1 Integer representation

The integer representation is used to represent numbers without a fractional part. Integer types can be of any dimension; however, they are typically a multiple of the specific architecture's word size as reported in Table 1.1. Every integer data type has a range determined by its size.

When it is necessary to represent negative (signed) numbers as well, two's complement representation is used. This representation has the advantage that all the basic mathematical operations (addition, subtraction, multiplication) can be performed in exactly the same way both for positive and negative numbers. Therefore, the same hardware can be used handle signed and unsigned numbers. In the two's complement representation, the leading bit is the sign bit.

More formally, given an integer number with n bits ($b_{n-1}b_{n-2}b_{n-3}...b_1b_0 \mid b_* \in \{0,1\}$) represented in two complement, the representation shall be interpreted as described in Eq. (1.1).

$$-1 * 2^{n-1} * b_{n-1} + 2^{n-2} * b_{n-2} + 2^{n-3} * b_{n-3}...2^1 * b_1 + 2^0 * b_0 \tag{1.1}$$

Finally, when using two's complement representation for signed numbers, and considering n as the number of bits in the representation, the range of numbers representable is shown in Eq. (1.2).

$$\begin{cases} [-2^{n-1}, 2^{n-1} - 1] & Signed \\ [-2^n, 2^n - 1] & Unsigned \end{cases} \tag{1.2}$$

1.1.2 Floating point Representation

A floating-point number is represented by multiple fields, namely a sign, an exponent, a significand, and a predefined base. The exponent is a signed integer number, and the base is typically equal to 2.

Given the value of each field, the interpretation of a floating point number is shown in Eq. (1.3).

$$-(1)^{sign} * significand * base^{exponent} \tag{1.3}$$

There are many floating point standards as reported in Table 1.3, and the most wildly used is the IEEE-754 [1]. The most relevant are single and double precision (32 and 64 bit), also called binary32 and binary64.

In the binary32 standard, to gain more range, the significand is normalized, so the leading bit d_0 is always equal to one and can be omitted.

Type	**Range signed**	**Range unsigned**
8 bit	$[-2^7, 2^7 - 1]$	$[0,2^8]$
16 bit	$[-2^{15}, 2^{15} - 1]$	$[0,2^{16}]$
32 bit	$[-2^{31}, 2^{31} - 1]$	$[0,2^{32}]$
64 bit	$[-2^{63}, 2^{63} - 1]$	$[0,2^{64}]$

Table 1.1: List of the most common integer types

	Sign	**Exponent**	**Significand**	**Base**
12.12122058 =	0	10000010	10000011111000010000101	*Binary*
	0	130	1.5151525735855103...	*Decimal*

Table 1.2: Example of floating point representation

Additionally, the exponent ranges from -126 to $+127$ because exponents of -127 and $+128$ are reserved for limits and special error values. In the actual representation, the exponent is represented in excess 127. Therefore, starting from an unsigned exponent of 130, to retrieve the actual exponent 127 will need to be subtracted. So, referring to Table 1.2, the map between float and real number is achieved as in Eq. (1.4).

$$-1^0 \times 2^{(130-127)} \times 1.5151525735855103 = 12.12122058 \tag{1.4}$$

The IEEE-754 standard also specifies all the possible operations and exceptions that can occur when working with floating point numbers. For this reason, the floating point data type supports two unique states, *NaN* and infinity. The former is used to handle the outcome of operations that result in undefined behavior such as $\frac{0}{0}$ or $\frac{\infty}{\infty}$. The latter is used to support operations that lead to infinity or to handle overflows.

Other important floating point formats are the x87 extended 80-bit – which is similar to binary64 but with a longer significand – and bfloat16 (or simply bfloat) which is like binary32 but with a shorter significand. bfloat16 was intended to be used with deep learning algorithms, where less precision than binary32 is enough.

Computers have dedicated hardware to work with different floating point types, the FPU (floating point unit). Typically the FPU only supports a subset of all the possible floating point formats. However, it is still possible to find architectures without hardware support to floating point, like some ATMega and STM32 ones [2] [3]. This is done either to achieve better power consumption, or to reduce the costs. In these cases, if needed, floating point operations are emulated by the CPU

1.1.3 Fixed Point Representation

A fixed point number is essentially an integer with a fractional part.

If we call x the position of the first bits of the fractional part, a signed fixed point with n bits ($b_{n-1}b_{n-2}b_{n-3}...b_1b_0 \mid b_* \in \{0,1\}$) is interpreted as in Equation 1.5.

Type	Significand bits	Exponent bits
binary16	10	5
binary32	23	8
binary64	52	11
binary128	112	15
binary256	236	19
bfloat16	7	8
x86 80bit	63+1	15

Table 1.3: List of the most common Floating points types

Typically, the fixed point size n is a multiple of the architecture word.

$$(-1) * b_{n-1} * 2 *^{n-x-1} + b_{n-2} * 2^{n-x-2} + b_{n-3} * 2^{n-x-3} ... b_1 * 2^{1-x} + b_0 * 2^{-x} \quad (1.5)$$

Fixed point representation has the advantage of being compatible with the hardware for integer numbers. This lead to the possibility of using fixed point instead of floating point in low energy devices. The hardware does not enforce the fractional part's position, so multiplication and division require a scaling operation to preserve the decimal place after the operation. The dimension of the fractional and integer parts determines the precision and the range. An example illustrating the fixed point representation is reported in Table 1.4.

Writing a program exploiting fixed point data types is an error-prone and time-consuming procedure.

1.2 Errors in numerical representation

As the number of bits representing a type is finite, it is impossible to map all the numbers to a data type without introducing errors. In this section we will present an overview of such errors and the situations in which they occur.

	Sign	Integer	Fractional	Base
12.1212205 =	0	1100	000111110000100001001110100	*Binary*
	0	12	0.121220499277115...	*Decimal*

Table 1.4: Example of fixed point representation

1.2.1 Arithmetic Overflows

Arithmetic Overflows are errors that occur when an operation generates a number that exceeds the range of representability. For integer and fixed point, two behaviors are possible: wrapping overflow, and saturating overflow. When a number exceeds the limits in a wrapping overflow, it is truncated, and only the lower part of the representation is preserved. Wrapping overflow can lead to a counterintuitive result, like adding two positive signed integer results in a negative one as shown in Table 1.5.

22509	+	28605	=	-14422
0101011111101101	+	0110111110111101	=	1100011110101010

Table 1.5: Example of wrapping overflow

Saturating overflow instead limit the result to the maximum or minimum value representable, as exemplified in Table 1.6. In some architectures like x86, a flag is set when a signed integer overflow occurs. Floating point overflows result in $-\infty$ or $+\infty$.

22509	+	28605	=	32767
0101011111101101	+	0110111110111101	=	0111111111111111

Table 1.6: Example of Saturating overflow

1.2.2 Round-off

While overflow errors are caused by situations where the range of a representation is not sufficient to represent the result of a computation, round-off errors originate from situations where the precision of the representation is not sufficient.

Fixed point representations are not affected by round-off when considering addition and subtraction. This can be demonstrated by considering that fixed point representations can be mapped exactly to the set of rational numbers with denominator 2^n where n is the size of the fractional part. This subset of rational numbers is closed with respect to addition and subtraction, and therefore no round-off can occur when performing such operations. Other operators, such as multiplication and division, can be affected by round-off even in fixed point representations.

Floating point representations can be affected by round-off from all operators. This is due to the fact that the density of the representation is dependent on the specific exponent utilized.

Let us consider a generic floating point x_0, and the smallest representable number x_1 such that $x_1 > x_0$. All the numbers between x_0 and x_1 are approximated with some policy to one extreme. Specifically, the IEEE 754[1] requires that the result of an elementary operation rounds to the nearest. The difference between x_0 and x_1 is called ULP(unit of least precision), and it is used as a measure of accuracy in numeric calculations[4]. The ULP of a generic floating point x with $\mid x \mid \in [2^e, 2^{e+1}]$ and precision p is shown in Eq. (1.6)

$$2^{\max(e, e_{min}) - p + 1} \tag{1.6}$$

1.2.3 Representation Mismatches

Some types like floating point have specials states like ∞ or *NaN*. If a floating point in such a special state is converted to an integer type, a representation mismatch occurs because there is no corresponding representation in the integer type.

1.3 The CORDIC Algorithm

The CORDIC algorithm was introduced in 1959 by Volder [[5], [6]]. In Volder's version, CORDIC makes it possible to perform rotations (and therefore to compute sine, cosine, and arctangent functions) and to multiply or divide numbers using only shift-and-add elementary steps. For this reason, it is perfect for low power environments in conjunction with fixed point.

This algorithm was developed to substitute the analog navigation computer of the B-58 bomber aircraft with a digital computer [7]. Since then, CORDIC has been implemented in many pocket calculators, such as Hewlett Packard's HP-35 [8], and in arithmetic coprocessors such as the Intel 8087 [9].

CORDIC comes in vectoring and rotation modes, which we will illustrate in more detail in the following sections.

1.3.1 Rotation mode

The basic idea of the rotation mode of CORDIC is to perform a rotation of angle θ as a sequence of elementary rotations of angles $\pm \arctan 2^{-n}$. The algorithm is based on the decomposition of $\theta = z_0$ on the discrete base $w_n = \arctan 2^{-n}$. To obtain the correct decomposition, the nonrestoring decomposition algorithm can be used.

The nonrestoring decomposition algorithm states that if we let (w_n) be a decreasing sequence of positive real numbers such that $\sum_{i=0}^{\inf} w_i$ converges and $\forall n : w_n \leq \sum_{k=n+1}^{\inf} w_k$ then for any $t \in [-\sum_{k=0}^{\inf} w_k, \sum_{k=0}^{\inf} w_k]$ the sequences t_n and d_n defined as in Eq. (1.7) satisfy Eq. (1.8).

$$\begin{cases} t_0 = 0 \\ t_{n+1} = t_n + d_n w_n \\ d_n = \begin{cases} 1 & t_n \leq t \\ -1 & otherwise \end{cases} \end{cases} \tag{1.7}$$

$$t = \sum_{n=0}^{\inf} d_n w_n = \lim_{n \to \inf} t_n \tag{1.8}$$

Considering $w_n = \arctan 2^{-n}$ the hypothesis of nonrestoring decomposition are satisfied and $\forall \theta :\mid \theta \mid \leq \sum_{n=0}^{\inf} \arctan 2^{-n}$ equation Eq. (1.9) will hold.

$$\theta = \sum_{k=0}^{\inf} d_k w_k, d_k = \pm 1, w_k = \arctan 2^{-k} \tag{1.9}$$

The base iteration of the Cordic algorithm is given by mixing a rotation of angle $d_n w_n$ as shown in Eq. (1.11) with the nonrestoring decomposition of θ shown in Eq. (1.10).

$$\begin{cases} t_0 = 0 \\ t_{n+1} = t_n + d_n w_n \\ d_n = \begin{cases} 1 & t_n \leq t \\ -1 & otherwise \end{cases} \end{cases} \tag{1.10}$$

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} cos(d_n w_n) - sin(d_n w_n) \\ sin(d_n w_n) - cos(d_n w_n) \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{1.11}$$

Using the relations in Eq. (1.12) we can rewrite the rotation as in Eq. (1.13), and the non restoring iteration can be rewritten as in Eq. (1.14) if we set $z_n = \theta - t_n$.

$$\begin{aligned} tan(w_n) &= 2^{-n} \\ cos(d_n w_n) &= cos(w_n) \qquad \text{as: } d_n = \pm 1 \\ sin(d_n w_n) &= d_n sin(w_n) \qquad \text{as: } d_n = \pm 1 \end{aligned} \tag{1.12}$$

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 - d_n 2^{-n} \\ d_n 2^{-n} - 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{1.13}$$

$$\begin{cases} z_0 = \theta \\ z_{n+1} = z_n - d_n w_n \\ d_n = \begin{cases} 1 & z_n \geq 0 \\ -1 & otherwise \end{cases} \end{cases} \tag{1.14}$$

Settling all the information together, we obtain the Cordic iteration shown in Eq. (1.15)

$$\begin{cases} x_{n+1} = x_n - d_n * y_n * 2^{-n} \\ y_{n+1} = y_n + d_n * x_n * 2^{-n} \\ z_{n+1} = z_n - d_n * arctan(2^{-n}) \end{cases} \tag{1.15}$$

The $\arctan(2^{-n})$ member, being constant, can be computed ahead of time and tabulated.

Under the conditions shown in Eq. (1.16) and Eq. (1.17), each iteration asymptotically converges as shown in Eq. (1.18).

Therefore, we conclude that choosing the initial conditions $\{x_0 = 1/K, y_0 = 0, z_0 = \theta\}$, we can compute $\sin\theta$ and $\cos\theta$.

$$K = \prod_{n=0}^{\inf} \sqrt{1 + 2^{-2n}} = 1.646760258121 \tag{1.16}$$

$$\mid z_0 \mid \leq \sum_{n=0}^{\inf} \arctan 2^{-n} \tag{1.17}$$

$$\lim_{n \to \inf} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = K * \begin{pmatrix} x_0 \cos z_0 - y_0 \sin z_0 \\ x_0 \sin z_0 + y_0 \cos z_0 \\ 0 \end{pmatrix} \tag{1.18}$$

1.3.2 Double Rotation mode

This method was suggested independently by Takagi et al. [[10], [11], [12]] and by Delosme [13] but with a different purpose. The basic idea behind the double rotation method is to perform the similarities of angle $\arctan(2^{-i})$ twice. The basic iteration becomes Eq. (1.19).

$$\begin{cases} x_{n+1} = x_n - d_n * y_n * 2^{-n} + (1 - 2d_n^2)2^{-2n-2}x_n \\ y_{n+1} = y_n + d_n * x_n * 2^{-n} + (1 - 2d_n^2)2^{-2n-2}y_n \\ z_{n+1} = z_n - 2d_n * arctan(2^{-n-1}) \end{cases} \tag{1.19}$$

The new scale factor is $K = \prod_{k=0}^{\inf}(1 + 2^{-2i}) = 1.3559096738634793803$. With double rotation mode it's possible to compute cos^{-1}, see Eq. (1.20), and sin^{-1}, see Eq. (1.21).

$$\begin{cases} \theta_0 = 0 \\ x_0 = 1 \\ y_0 = 0 \\ t_0 = t \\ d_n = \begin{cases} sign(y_n) & if x_n \geq t_n \\ -sign(y_n) & otherwise \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} = \theta_n + 2d_n \arctan 2^{-n} \\ t_{n+1} = t_n + t_n 2^{-2n} \end{cases} \tag{1.20}$$

$$\begin{cases} \theta_0 = 0 \\ x_0 = 1 \\ y_0 = 0 \\ t_0 = t \\ d_n = \begin{cases} sign(x_n) & if y_n \le t_n \\ -sign(x_n) & otherwise \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} = \theta_n + 2d_n \arctan 2^{-n} \\ t_{n+1} = t_n + t_n 2^{-2n} \end{cases} \tag{1.21}$$

1.3.3 Vector mode

This mode is used for computing arctangents. Starting from the rotation mode's iteration with $z_0 =_\theta$, we can define a new variable $z'_0 = \theta + z_n$. Remember that θ is unknown. z'_n measures the opposite of the angle by which (x_0, y_0) must be rotated to get (x_n, y_n). If we have rotated by an angle whose opposite is greater than θ, then (x_n, y_n) is below the x-axis; hence y_n is negative. Otherwise, y_n is positive. Therefore the test $z'_n \ge \theta$ can be replaced by $y_n \le 0$. By doing this, we get the vectoring mode of CORDIC.

The iteration step is shown in Eq. (1.22) and it converges as shown in Eq. (1.23).

$$\begin{cases} x_{n+1} = x_n - d_n * y_n * 2^{-n} \\ y_{n+1} = y_n + d_n * x_n * 2^{-n} \\ z_{n+1} = z_n - d_n * arctan(2^{-n}) \\ d_n = \begin{cases} sign(-y_n) \\ -sign(-y_n) \end{cases} \end{cases} \tag{1.22}$$

$$\lim_{n \to \inf} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = \begin{pmatrix} K\sqrt{x_0^2 + y_0^2} \\ 0 \\ z_0 + \arctan \frac{y_0}{x_0} \end{pmatrix} \tag{1.23}$$

1.4 Linear Programming

Linear Programming [14] is a moderately recent discipline of mathematics. The first rigorous formulations for a specific problem were introduced in 1939 by the Soviet mathematician and economist Lonid Kantrovich. In 1947 the first general formulation of a Linear Program was made with the critical work by Dantzig.

The simplex algorithm [15] he invented is still used today in many solver tools. It was originally intended to solve planning problems in the US Air Force, but it was soon noted that this method solves many classical optimization problems.

The first machine implementation of the Simplex algorithm was done on the SEAC computer at the National Bureau of Standards [16]. In the 50s, the first implementation on a scientific computer was made for IBM 701 [17] and then for IBM 704 [18]. With these implementations, the first attempts to solve Mixed Integers programs were made.

During the 70s and 80s, there were many innovations, the most notable was the introduction of the dual simplex algorithm [19]. With MIP (Mixed Integer Programming) solvers, new procedures to explore the search tree were introduced. These optimizations were made possible mainly due to innovations and progression in computer architectures.

In 1979, Khachiyan proved that LP problems could be solved in linear time [20], although his approach was never used in actual programs. In general, MIP algorithms in use today remain pretty similar to the ones used in the 70s. The gains in terms of speed come predominantly from advantages in computer processors and parallelization algorithms, and exploiting the power of multi-core processors.

1.4.1 Problem Forumulations

A Linear Program Problem aims to maximize or minimize a linear function (called objective function) subject to a finite number of linear constraints.

The general form is shown in Eq. (1.24).

$$\begin{aligned}
&\text{[maximize | minimize]} && \sum_{j-1}^{n} c_j x_j \\
&\text{subject to} && \sum_{j-1}^{n} a_{ij} x_j [=, \geq, \leq] b_i \quad (i = 1, 2, ..., m) \\
& && x_j [\geq, \leq] 0 \qquad\qquad (i = 1, 2, ..., m)
\end{aligned} \tag{1.24}$$

The general form is translated into an equivalent form called standard form. This is done because the simplex algorithm used to solve a Linear Program takes as input the standard form.

To bring a general form into a standard form, all the constraints must be equality constraints, and all variables must be non-negative. let us consider the example in Eq. (1.25).

$$\begin{aligned} \text{minimize} \quad & z = 3x_1 + 2x_2 - x_3 + x_4 \quad \textit{(objective function)} \\ \text{s.t.} \quad & \begin{cases} x_1 + 2x_2 + x_3 - x_4 \leq 5 \\ -2x_1 - 4x_2 + x_3 + x_4 \leq -1 \\ x_1 \geq 0 \\ x_2 \leq 0 \end{cases} \end{aligned} \tag{1.25}$$

This set of equations can be converted into a standard form by following the following steps:

- The right-hand side is non-negative:

General Form	Standard Form
$-2x_1 - 4x_2 + x_3 + x_4 \leq -1$	$2x_1 + 4x_2 - x_3 - x_4 \geq 1$

  Just multiply both sides by -1.

- All constraints are expressed as equality constraints:

General Form	Standard Form
$x_1 + 2x_2 + x_3 - x_4 \leq 5$	$x_1 + 2x_2 + x_3 - x_4 + s_1 = 5$
$+2x_1 + 4x_2 - x_3 - x_4 \geq 1$	$+2x_1 + 4x_2 - x_3 - x_4 - s_2 = 1$

  We added two variables s_1, s_2. s_1 is called slack variable. s_2 is called surplus variable. Slack variables are added in the case of $\leq$, and they are prefixed with a $+$. Surplus variables are added in the case of $\geq$, and they are prefixed with a $-$.

- Non-negativity constraints for all variables:

General Form	Standard Form
$x_1 + 2x_2 + x_3 - x_4 + s_1 = 5$	$x_1 - 2y_2 + x_3 - x_4 + s_1 = 5$
$+2x_1 + 4x_2 - x_3 - x_4 - s_2 = 1$	$+2x_1 - 4y_2 - x_3 - x_4 - s_2 = 1$
$x_2 \leq 0$	$y_2 \geq 0$

We substitute the variable x_2 with $y_2 = -x_2$. So for each variable $x_i \leq 0$ a new variable $y_i = -x_i$ with constraint $y_i \geq 0$

- Remove Variables that are Unconstrained in Sign:

General Form	Standard Form
Min $z = 3x_1 + 2x_2 - x_3 + x_4$	**Min** $z = 3x_1 + 2x_2 - (w_3 - k_3) + (w_4 - k_4)$
$x_1 - 2y_2 + x_3 - x_4 + s_1 = 5$	$x_1 - 2y_2 + (w_3 - k_3) - (w_4 - k_4) + s_1 = 5$
$+2x_1 - 4y_2 - x_3 - x_4 - s_2 = 1$	$+2x_1 - 4y_2 - (w_3 - k_3) - (w_4 - k_4) - s_2 = 1$
$y_2 \geq 0,\ x_1 \geq 0$	$y_2 \geq 0,\ x_1 \geq 0\, w_3 \geq 0\, k_3 \geq 0\, w_4 \geq 0\, k_4 \geq 0$

Variable x_3, x_4 where substitute in order with $x_3 \rightarrow [w_3 \geq 0, k_3 \geq 0]$, $x_4 \rightarrow [w_4 \geq 0, k_4 \geq 0]$. So each variable x_i unconstrained in sign is substituted with $(w_i - k_i)$

So an LP problem in standard form is formulated as in Eq. (1.26) where a_{ij} , b_i and c_j are real constants.

$$\begin{array}{lll} \text{[maximize, minimize]} & \sum_{j-1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j-1}^{n} a_{ij} x_j = b_i & (i = 1, 2, ..., m) \\ & x_j \geq 0 & (i = 1, 2, ..., m) \end{array} \tag{1.26}$$

When at least one variable x_i is required to be an integer, the problem becomes a Discrete Linear Programming problem. If all the variables are integers, then the problem is an Integer Linear programming (ILP problem). Although its formulation is very similar to an LP Problem, an ILP Problem cannot be solved using LP algorithms; moreover, it was proved that solving ILP Problems is NP-complete [21].

Chapter 2

Frameworks

The contribution this thesis introduces lies in the field of precision tuning and originates from prior work in the state-of-the-art. This chapter provides an introduction to approaches, methodologies, and previously developed frameworks.

2.1 Precision tuning frameworks

In this section, we illustrate the most popular frameworks and their approach to precision tuning. They are representative of three main classes of precision tuning approaches: binary-level transformations (CRAFT), compiler-level transformations (HiFPTuner), and source-level transformations (Daisy). For a more complete discussion, we demand the reader to recent surveys [22, 23].

2.1.1 Binary Transformations

CRAFT [24] is a framework for analyzing a previously compiled binary that uses double-precision data types and successively modify it to build mixed-precision versions. It can be run on any programming language, but it is dependent on the target architecture. Moreover, it implements a search algorithm to find the best precision mix in the original program. The target binary to be analyzed needs to be instrumented by using the Intel Pin library [25]. So each function, basic block, and instruction is analyzed to understand which parts of the code will exploit reduced precision computation. The Configuration Generator then builds, using a breadth-first algorithm, multiple mixed-precision configurations for the same executable. Each configuration is stored as a series of human-readable mappings; Each instruction is also flagged with one particular type:

Single when the precision of the instruction needs to be decreased
Double when the precision required is given only by the double type.
Ignored if the computation must be left in the original computation. For each generated configuration, the binary is modified by exploiting the Dynist library, a tool for Binary Modification. Each instruction that is not flagged as ignore is replaced by a routine that performs the necessary casts and then carries out the computation. The single-precision values are stored in the same place as the double-precision ones, precisely in the lower 32 bits, while in the upper 32 bits contain a flag (`0x7ff4DEAD`) that if the value is incorrectly interpreted as a double value, it transforms into a NaN value, thus preventing the propagation of wrong values in the program. The final binaries are then tested against some test input and the a user-specified routine evaluates results. This framework has been evaluated on many open-source benchmarks.

2.1.2 Compiler Transformations

HiFPTuner [26] follows another type of approach. Instead of instrumenting a binary, it analyzes both the source code and the program's runtime behavior. This way, the algorithm reduces the search cost by reducing the search space for the sake of scalability. To work, it exploits the dependence analysis and Edge profiling techniques to build a graph where each variable is represented with a node, and the arc represents the dependency between them. The weights on the arcs represent how many times that dependence is found. The weight of each edge is found by analyzing the run time behavior. After this step, the tools group the variable using a community detection problem algorithm, so each group will be composed by the variable that frequently interacts. A hierarchy is established running the previous passes multiple times and continuously collapsing the variables in the same set into a single node. Once the hierarchy is built, the tool performs the actual precision tuning. The algorithm starts from the topmost level of the graph and searches for the best configuration by treating all the variables in the same set as having the same data type. The information is propagated to the following nodes to reduce the search space and avoid looking for more precise alternatives. When a configuration is found, it is verified against a set of test inputs to check if the error is in the required bound. HiFPTuner is independent both from the source language and the destination architecture as it works with the LLVM IR.

2.1.3 Source Transformations

Daisy [27] is a source to source compiler whose aim is to reduce the precision of variables in a given code region while still preserving the wanted precision as specified by the programmer.

The input program is written in a real-valued language, but it is not limited to it, and it will be processed a single function at the time. For each function, preconditions and postconditions of the computation are specified using the keywords **require** and ensuring So the compiler will find a precision mix that will respect the specified condition. The function's body comprises all the common operations, transcendental functions, and local variables but does not support loops. After the analysis, Daisy will produce a Scala or C source code as output. The data types currently supported by Daisy are fixed-point (16 or 32 bits) and IEEE754 single, double, and quad precision. Daisy has the benefit of being highly customizable and easy to extend also to other representations, possibly custom ones. The tuning process is so composed:
First step the tool rewrites the expression into equivalent ones with a small round-off error. The new expression is generated with a genetic algorithm. This procedure is based on the Xfp tool, which has been extended to perform more effective transformations at each iteration of the algorithm. The performance and uniform precision guide the generation, so the resulting expression will never be slower in computation than the original one.
Second step Daisy rewrites the expressions in a sort of static single assignment form.
Third step The range analysis is carried out using one of the possible supported algorithms like interval arithmetic, affine arithmetic, or more expensive alternatives
Fourth step The mixed precision tuning is performed. Daisy does not require to run the original program as it performs static error analysis. Because of this, the error bounds computed by Daisy are proven to be valid and will always be correct, disregarding the specific input values.
Fifth step the result is materialized into a human-readable programming language (C or Scala), together with all the necessary casts and declarations. The tool can have guarantees on the output program in terms of errors and execution time at the cost of not supporting conditional execution.

2.1.4 About Our Contribution

We presented three different approaches to precision tuning, each one with its own benefit and drawback. We have based our works on a tool called TAFFO (Tuning Assistant for Floating Point to Fixed point Optimization), and HiFPTuner is the most closely relatable to it. Indeed TAFFO works with general-purpose programming languages starting directly from an annotated source code likes HiFPTuner. Daisy lacks the first feature and CRAFT the latter. However, they differ on the knowledge input required: TAFFO does not necessarily rely on the profiling of target application as, depending on the use case, that may not always be a feasible solution. Instead, TAFFO uses static analyses, similar to the Daisy framework, to comes up with the proper precision mixing. This approach is more general. In fact, nothing prevents the input of the static analyses to be generated from dynamic application profiling. In the following sections, more information about compilers and TAFFO will be provided.

2.2 The Structure of a Compiler

Programming languages are used to describe computations to people and machines. The world depends on programming languages because all the software was written in some programming language. A compiler is a program that can read a source language and translate it into an equivalent program in the target language. One of the compiler's roles is to report any errors in the source program that it detects during the translation process. The first compilers were simple syntax driven translators. They took as input a low-level programming language, such as assembly code, and translated it directly into machine code in binary format. With the growth in popularity of higher level languages, compilers became even more critical in order to let programmers obtain a final program that was as quick as if it was written directly in assembly. Another possibility is using an interpreter that directly executes the operations specified in the source program on inputs supplied by the user. In Figure 2.1 A representation of the different structure between a compiled program and an interpreted one is given.

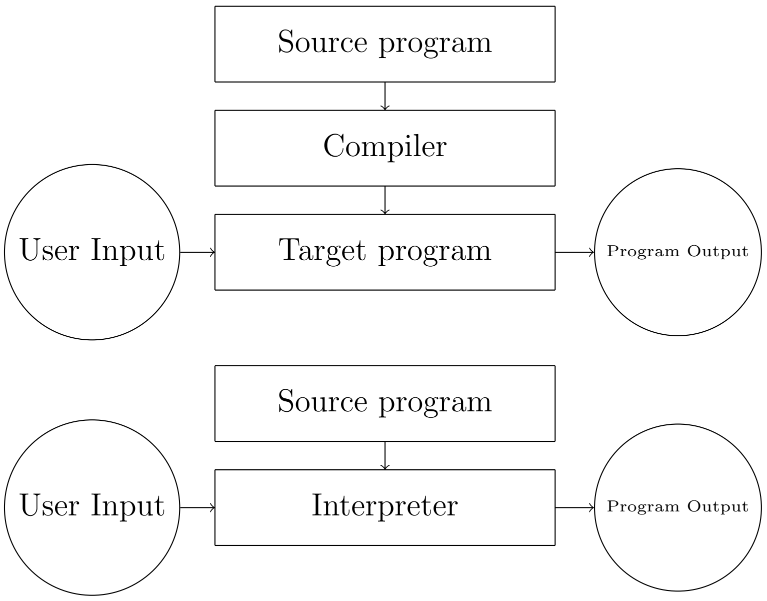


Figure 2.1: High level structure of compiled and interpreted program

In addition to the compiler, several other programs work together to ease the process of compilation. For example, the preprocessor is typically used to expand all the macro defined in the source code before passing the code to the compiler. An example of preprocessing is given in Listing 1

```c
#define NUMBER 12
#define SUM(x,y) x+y

int main(){
    int x = NUMBER;
    int y = NUMBER*2;
    int sum = SUM(x,y);
    return sum;
}
```

```c
int main(){
int x = 12;
int y = 12*2;
int sum = x+y;
return sum;
}
```

Listing 1: On the left the unprocessed C version. On the right, the same C file after the Preprocessing phase.

The compiler often does not output machine code but emits assembly code that is later fed to an assembler. The assembler takes the assembly as input and produces machine code as its output. The linker is responsible for linking together different object files and library files into the code that runs on the machine.

The loader then puts together all of the executable object files into memory for execution. In the Figure 2.2 an example of a compiler pipeline is given.

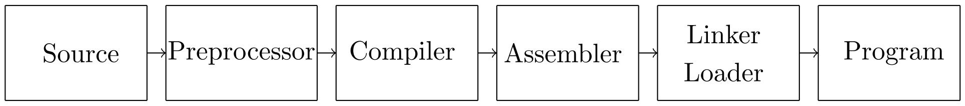


Figure 2.2: Typical compiling pipeline

We have treated a compiler as a black box that maps a source program into a semantically equivalent target program. Since a compiler is a complex machine with very different phases, it is often divided into three subcomponents. The LLVM Compiler Framework is an example of a modern compiler following the best practices in state of the art. It has gained notoriety in the last years, both from academic research and the industry, due to its modularity.

2.2.1 Front-end

The front-end is the compiler component, which is in charge of analyzing the source code, given as input in a high level language, and lowering it into a low level representation used internally by the compiler, called intermediate representation (IR). In the LLVM toolchain, Clang is the front-end for the C and the C++ languages, converting them into LLVM's IR, called "LLVM-IR".

Lexical analyzer

The first component of the front-end is the lexical analyzer or scanner. The lexical analyzer read each character of the source program and generate a series of tokens.

(**token-name**, **attribute-value**)

token-name is an abstract symbol that is used during syntax analysis, and the second component **attribute-value** points to an entry in the symbol table for this token.

Syntax Analysis

The tokens are passed to Syntax Analysis or parser, which generates a tree-like intermediate representation that depicts the token stream's grammatical structure. All the tree leaves will contain terminal symbols, which are part of the text that cannot be split furthermore using the grammar rules. If no rule can match a particular statement, an error is produced. In Figure 2.3 the tree intermediate representation used by Clang is reported.

```
int main(){
int red = 10;
int number = 2;
int big = red -number/10;
}
```

Listing 2: Simple C program for AST

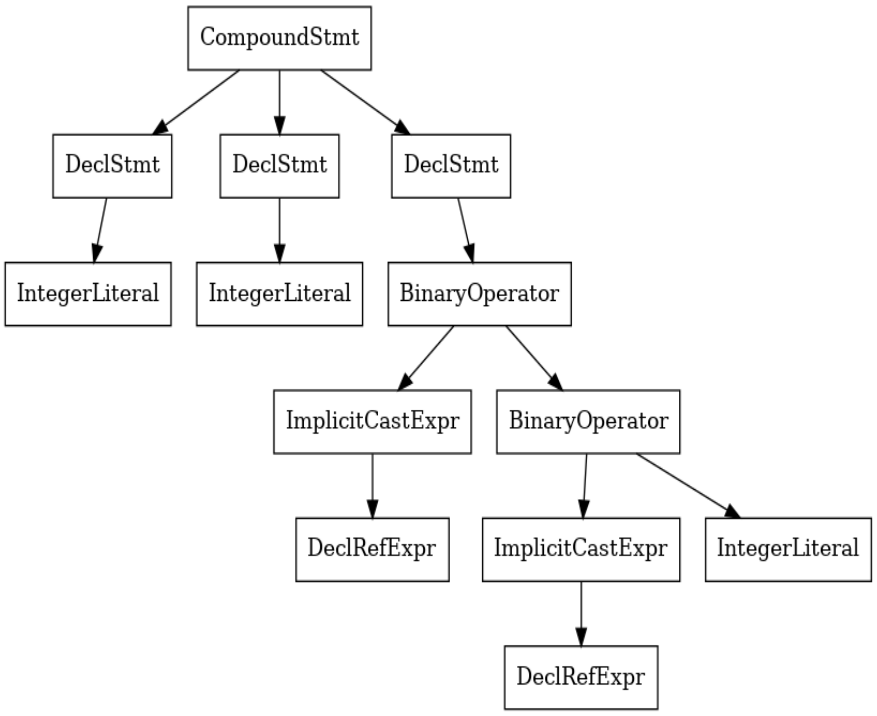


Figure 2.3: AST generated from Clang with in input the simple C program 2

Semantic analyzer

The semantic analyzer uses the syntax tree in combination with the symbol table's information to check the semantic consistency with the source program's language definition. An essential part of semantic analysis is type checking, where the compiler checks that each operation matches his operand and his return type. The front-end is independent of the target architecture. It depends only on the source language.

Intermediate representation

In the process of translating a source program, the compiler may construct one or more intermediate representations. Syntax trees are a type of intermediate representation; they are commonly used during syntax and semantic analysis. Many compilers generate other intermediate representation likes an explicit low-level or machine-like intermediate representation, to decouple the optimization phases from the source language. The LLVM-IR is as low-level as the assembly language; however, it is independent of the destination machine. Like in the assembly language, each statement describes a primitive operation, such as mathematical operations, function calls, memory accesses, etc. LLVM-IR has an unlimited number of registers, and the memory is viewed as a set of logical variables. LLVM-IR is in the Static Single Assignment form (SSA). In the Single Static Assignment form, each register is assigned exactly once in the program, so each operation generates a new register. This is done mainly to simplify further code transformations. LLVM-IR represents the program as a collection of global variables and functions. Each function is composed of basics blocks, and instructions compose the basic block. In Listing 4 an example of LLVM-IR generated by Clang is reported.

```c
#include <stdio.h>

int main(){
    int red = 10;
    int number = 2;
    int big = red -number/10;
    printf("%i\n", big);
    return 0;
}
```

Listing 3: Simple C program to exhibit clang output

```
@.str = private unnamed_addr constant [4 x i8] c"%i\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 10, i32* %2, align 4
    store i32 2, i32* %3, align 4
    %5 = load i32, i32* %2, align 4
    %6 = load i32, i32* %3, align 4
    %7 = sdiv i32 %6, 10
    %8 = sub nsw i32 %5, %7
    store i32 %8, i32* %4, align 4
    %9 = load i32, i32* %4, align 4
    %10 = call i32 (i8*, ...) @printf(i8* getelementptr
    inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %9)
    ret i32 0
}
declare dso_local i32 @printf(i8*, ...) #1
```

Listing 4: LLVM-IR generated with Clang from the simple C program 3

2.2.2 Middle-end

The middle-end is the part of the compiler that performs optimizations that are independent of the target architecture. It executes units called passes on the IR. There are two types of passes: analysis and transformation. The analysis passes are used to collect information about the programs. Transformation passes are used to change the IR. Each pass can declare dependencies upon others' passes and which passes invalidates. In addition, the pass manager schedules each pass in order to optimize the speed of execution. Examples of analysis passes are the CallGraphWrapperPass[28], DependenceAnalysisWrapperPass [29] An example of Transformation passes are the constant propagation SCCPPass[30], or inlining createPartialInliningPass[31]. Optimization passes can be costly in terms of execution time. Therefore only a reduced set of them are enabled by default. The programmer can control whether to enable a specific optimization or not. LLVM decouples the middle end from the front-end and the back-end, meaning that the same pass can be used for different source languages and destination targets. In addition to the LLVM framework's passes, developers can write other passes and use them as a plugin or included in the LLVM build. These plugins are compiled as external objects that are loaded at runtime. Under an example of loop unrolling transformation of a for loop is provided. Loop unrolling is used to unroll a loop to take fewer branches and improve the parallelization of the execution. In Listing 6 is reported the LLVM-IR of a loop without optimization. In Listing 7 is reported the LLVM-IR of a loop with optimization.

```c
int main(int argc, char ** argv){
int tmp[10];
for (int i = 0; i < 10; ++i){
    tmp[i] = argc;
}
return tmp[3];
}
```

Listing 5: Simple C program used to demonstrate loop unrolling

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main(i32 %0, i8** %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  %6 = alloca [10 x i32], align 16
  %7 = alloca i32, align 4
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  store i32 0, i32* %7, align 4
  br label %loop_start

  loop_start:
  %9 = load i32, i32* %7, align 4
  %10 = icmp slt i32 %9, 10
  br i1 %10, label %loop_body, label %end

  loop_body:
  %12 = load i32, i32* %4, align 4
  %13 = load i32, i32* %7, align 4
  %14 = sext i32 %13 to i64
  %15 = getelementptr inbounds [10 x i32],
  [10 x i32]* %6, i64 0, i64 %14
  store i32 %12, i32* %15, align 4
  br label %loop_inc

loop_inc:
  %17 = load i32, i32* %7, align 4
  %18 = add nsw i32 %17, 1
  store i32 %18, i32* %7, align 4
  br label %8, !llvm.loop !2

end:
  %20 = getelementptr inbounds [10 x i32], [10 x i32]* %6, i64 0, i64 3
  %21 = load i32, i32* %20, align 4
  ret i32 %21
}
```

Listing 6: LLVM-IR of an unoptimized loop with as source code Simple C program 5. The loop starts at basic block loop_start and ends at basic block end. The basic block loop_start test the conditions and takes the jumps. The basic block loop_body compose the body of the loop. The basic block loop_inc compose the epilogue of the loop where the loop counter is increased.

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @main(i32 %0, i8** nocapture readnone %1)
local_unnamed_addr #0 {
  %3 = alloca [10 x i32], align 16
  %4 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 0
  store i32 %0, i32* %4, align 16
  %5 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 1
  store i32 %0, i32* %5, align 4
  %6 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 2
  store i32 %0, i32* %6, align 8
  %7 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 3
  store i32 %0, i32* %7, align 4
  %8 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 4
  store i32 %0, i32* %8, align 16
  %9 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 5
  store i32 %0, i32* %9, align 4
  %10 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 6
  store i32 %0, i32* %10, align 8
  %11 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 7
  store i32 %0, i32* %11, align 4
  %12 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 8
  store i32 %0, i32* %12, align 16
  %13 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 9
  store i32 %0, i32* %13, align 4
  %14 = getelementptr inbounds [10 x i32],
  [10 x i32]* %3, i64 0, i64 3
  %15 = load i32, i32* %14, align 4
  ret i32 %15
}
```

Listing 7: LLVM-IR of the loop unrolling optimization with as source code Simple C program 5. The loop unrolling removes all the branches as it is capable of unrolling all the loop

2.2.3 Back-end

The compiler's back end is in charge of transforming the intermediate representation from the middle end into the final machine code. The significant difference between the IR and the assembly is the managing of register and memory. The LLVM-IR suppose that there is an unlimited amount of register and memory. Assembly instead, being linked to real hardware, has to handle a finite number of registers. So the back end has to choose which register to keep and what register to store into the memory. This phase is called register allocation, and it strongly depends on the liveness interference analysis. The liveness analysis on the IR decided if two registers are live in the same part of the code. If not, they can share the same register. The problem is related to the graph coloring algorithm, which is intractable due to the exponential complexity. So Compilers usually implement a heuristic to speed up the execution of the algorithm. The back end is also responsible for running all the hardware-dependent optimization like utilizing SIMD (single instruction multiple data) instruction or the processor's atomic capability. For each new change in the hardware, a separate back end must be developed to best match the new capability. LLVM supports many target architectures because the back end development is completely decoupled from the front/middle-end development. The main supported targets are ARM, MIPS, PowerPC, x86, AMD GPU (OpenCL), AVR, and WebASM. For this reason, the compilation can also be made for a different target machine than the system running the compiler; in this case, the compiler is performing a cross compilation. In Listing 8 an example of x86 assembly is reported.

```
 .text
 .file        "a.c"
 .globl        main                   # -- Begin function main
 .p2align        4, 0x90
 .type        main,@function
main:                                  # @main
 .cfi_startproc
# %bb.0:
 pushq        %rbp
 movq        %rsp, %rbp
 movl        %edi, -48(%rbp)
 movl        %edi, -44(%rbp)
 movl        %edi, -40(%rbp)
 movl        %edi, -36(%rbp)
 movl        %edi, -32(%rbp)
 movl        %edi, -28(%rbp)
 movl        %edi, -24(%rbp)
 movl        %edi, -20(%rbp)
 movl        %edi, -16(%rbp)
 movl        %edi, -12(%rbp)
 movl        -36(%rbp), %eax
 popq        %rbp
 retq
.Lfunc_end0:
 .size        main, .Lfunc_end0-main
 .cfi_endproc
                                     # -- End function
 .ident        "clang version 10.0.0-4ubuntu1 "
 .section        ".note.GNU-stack","",@progbits
 .addrsig
```

Listing 8: X86 of the loop unrolling optimization. The loop unrolling removes all the branches as it is capable of unrolling all the loops.

2.3 TAFFO

The Tuning Assistant for Floating Point to Fixed point Optimization framework, also known as TAFFO [32, 33, 34], is an optimization tool whose aim is to help developers automatically change the program precision mix to optimize the execution time while preserving its correctness. The developers annotate a subset of variables to be transformed by the tool into fixed points. TAFFO takes care to handle all the interactions between the transformed variable and the old program. As this process is entirely automated, it falls in the auto-tuning framework category. The tool is based on the LLVM compiler toolchain [35] and is implemented as a collection of multiple passes as reported in Figure 2.4. TAFFO operates with the LLVM-IR in the middle-end stage of the compiler. So TAFFO can operate on all the source language supported and to all the target supported by the LLVM framework. Moreover, each pass is completely decoupled from the successive one. Thus, a different version of each pass employing a different algorithm can be used without modifying other tool parts. The annotations expressed as text strings in the source code inform TAFFO about the variable's range and an initial seed value for the error propagation analysis.

2.3.1 Passes

Initialization

The initialization pass is the first one to be executed by TAFFO. Given the source program as an LLVM-IR file, it parses the annotations and creates the metadata, a type of data structure with all the annotation information but easier to handle in the various passes of LLVM. This pass also creates a copy of each function for each different call site. This is done to ease the auto-tuning process as each call site's arguments could be different fixed point types.

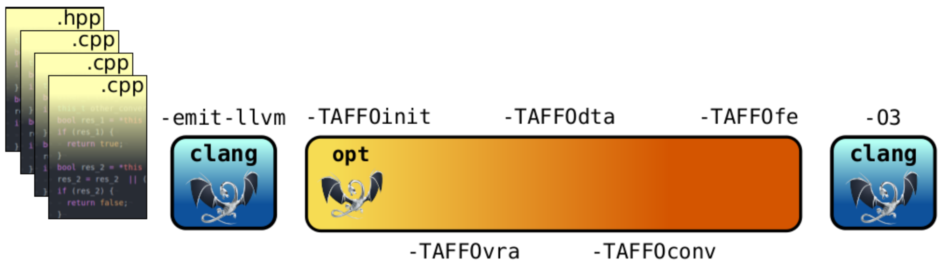


Figure 2.4: TAFFO pipeline

Value Range Analysis

The Value Range Analysis calculates the range of every value that interacts with an annotated one. It is an essential and complex operation because the more precise the data type's selection will be, the more accurate it is. Final ranges for the variables are chosen by exploiting range arithmetic and source code information. Utilizing only the range arithmetic is the fastest method but inaccurate, as the predicted range is very pessimistic and distant from the actual program behavior. Therefore, a new technique relying on symbolic execution is in development. This new analysis aims to simulate the program behavior at compile and increase the amount of valuable data. Unfortunately, this new approach may raise the compilation time.

Data Type Allocation

The Data Type Allocation pass (DTA) goal is to find an appropriate data type for each program's value. The selection process is led primarily by the range of each value provided by the previous pass. Indeed, the fixed point representation that fits the whole computed range must be chosen. The DTA, to avoid a very heterogeneous precision mix that can lead to slower programs due to casting, tries to merge similar fixed point types when used in the exact computation. As multiple definitions of each function were generated during the initialization, the DTA pass will collapse all functions with the same type for the arguments to reduce the final code size.

Conversion

Using information coming from the DTA pass, each variable is converted to the appropriate integer type, and, if needed, instructions to convert between the fixed point representation and the original data type are generated. If a particular instruction cannot be converted, fallback code is generated, converting the values back to their original types. This can happen when handling external functions like a dynamically loaded library that TAFFO sees as an external call without a body to analyze.

Feedback Estimator

Finally, the Feedback Estimator pass analyzes the final code produced by the conversion pass. It evaluates if a useful speedup has been achieved and if the error introduced by the data type selection is small enough. If any of these evaluations has a negative outcome, the conversion starts again from the DTA step, with different parameters.

2.3.2 TAFFO strengths and limits

Multiple output data type

The use of annotations allows a certain degree of selectivity TAFFO allows three types as output: fixed point, float, and double. This generates the possibility to use the best type for each different part of the program. Even if fixed points are unsuitable for a portion of the program, other floating point types may speed up the computation while keeping the error into the required bounds.

No guarantees on execution time

Working with the LLVM-IR representation, TAFFO does not consider the target architecture where the code will run. This is because for some architectures, the transformation to fixed point may slow down the code with respect to the original one.

No guarantees on error

The final program is proven to be correct. All the intermediate registers are chosen with enough integer bits to contain all the possible ranges, and in case of a problem, a fallback code is generated to use the original type. However, the use of fixed point does not guarantee that the error in the computation results will be accurate enough for the specific application.

Non convertible code

Particular regions of code, such as calls to an external library, must be handled carefully. In fact, TAFFO does not have access to external function bodies and cannot transform the types. So TAFFO generates a wrapper for these functions that converted back to its original data type the arguments, and the return value into fixed point once again if necessary. This can generate slowdowns because of unnecessary conversions.

Chapter 3

The FixM Approach

FixM is our approach to on-demand code generation of mathematical functions using fixed point arithmetic [36]. FixM has the objective to overcome the problem of external code call for the mathematical functions in an innovative way. In fact, the traditional approach to fixed point arithmetic assumes the program uses a given fixed point data type, and the whole code generation is pivoted around this rigid assumption. Instead, FixM uses dynamic fixed point [37], a technique that proved to be effective to reduce the rounding error on the final output.

Dynamic fixed point consists of potentially changing the bit partitioning of the fixed point with each program's different instruction. This scaling operation tunes the number of bits assigned to integral and fractional parts of the fixed point data type. The scaling operation may not change the total number of bits of the representation but changes each bit's meaning. Thus, it creates a different representation.

In the traditional approach, each mathematical function's fixed point implementation must be added manually for each fixed point representation in the program. When it comes to dynamic fixed point, redundant implementations are required for each possible bit partitioning which could be used in the code. Support to dynamic fixed point with the traditional approach requires each function to be implemented with each type of fixed point, leading to a huge static library. Considering F the number of mathematical functions, f_i the number of arguments of the i-th function, k the number of supported fixed points. For a simple implementation of the mathematical library, we can estimate $F \approx 35$, and the average $f_i \approx 2$.

$$\sum_{i=1}^{F} k^{f_i} \tag{3.1}$$

Formula that calculate the required number of functions in a simple implementation of the mathematical library with k supported fixed point formats

For a single bit width, k is equal to the bit width itself. Therefore, if we consider signed and unsigned 8 bit, 16 bit, and 32-bit types, k grows up. In such an implementation, the number of functions to be instantiated would be 350000. This number grows linearly with the number of functions and quadratically with the number of types. Therefore, even a simple expansion to include 64-bit types would increase the size of the library to 1.5 million functions. This approach becomes unfeasible in the case of highly dynamic fixed point implementations and in the case of automatic tuning of bit partitioning. Our approach automatically inserts the implementation of each mathematical function only for those fixed point representations that require its use. FixM acts after every precision tuning analysis that modifies the fixed point representations. In particular, it is designed to run during the code generation and optimization stages within the compiler.

3.1 Generality

FixM works at the compiler intermediate representation level (IR), which means FixM can abstract concepts depending on specific source programming languages or language extensions. Working at the IR level also abstracts many features of the underlying hardware architecture – a critical feature when targeting embedded systems due to the wide variety of platforms. Nonetheless, some assumptions are needed. In particular, FixM works whenever fixed point data types are mapped onto integer registers, and there is essential bit arithmetic support (`shift`, `and`, and `or`). Furthermore, these assumptions are verified in most cases, thus supporting a good level of generality for FixM.

3.2 Minimality

An application leveraging FixM may require different levels of precision in different situations. This information is codified in the fixed point data types selected by the precision tuning analysis. FixM, being part of TAFFO, knows where the mathematical library functions are employed and their expected input and output. Knowing the input and the output, FixM determines which are the required function to generate. Apart from providing the correct fixed point format, we also leverage the data type information to adapt the computation itself to the required precision. In other words, the generated fixed point functions are specialized not only with respect to the external interface.

3.3 Scalability

While FixM transforms each function call and generates the corresponding function bodies, it stores the information about which functions it previously inserted and for which representation. This internal cache allows FixM to implement code reuse techniques to minimize the code size. Therefore, FixM never generates two identical fixed point mathematical functions. Furthermore, the code cache can be used to further factorize the code generation across multiple compilation units in the same program, generating a library of specialized functions. A baseline approach has to generate all possible specializations of a function f for all possible dynamic fixed point types k in each of its n f parameters.

Chapter 4

FixM Implementation

The FixM approach to generate mathematical functions is based on the TAFFO framework. The new component of TAFFO is a template-based code generator called FixMAGE (FIXed point MAthematical function GEnerator). In this section, we will illustrate its theory of operation and how it integrates with the rest of TAFFO.

4.1 FixMAGE

In the current state-of-the-art, mathematical function libraries operate on floating point data types only. The interface of these libraries is specified by the POSIX [38] standard. If the CPU provides hardware acceleration of these functions, the mathematical library will use these resources. Otherwise, a software implementation of the function will be chosen. This latter situation is the industry standard

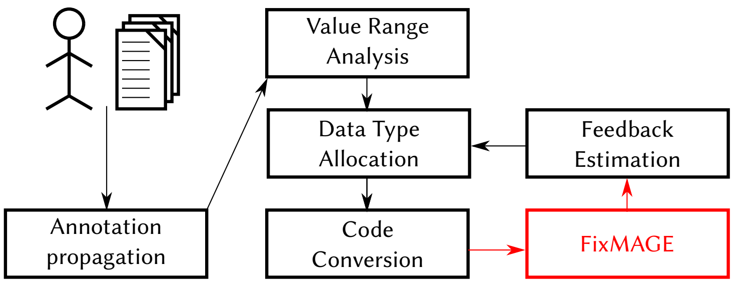


Figure 4.1: FixMAGE Structure and how it interacts with other passes of TAFFO

for embedded systems. In both cases, the mathematical library is an external piece of software. In FixM, the mathematical library is implemented as part of the compiler.

We implement FixM by adding a new component to TAFFO named FixMAGE. The pipeline of TAFFO with the integration of FixMAGE is shown in Figure 4.1. FixMAGE runs within the conversion pass of TAFFO, and performs three crucial steps. First, it detects where the program uses one of the supported POSIX standard mathematical functions. The functions currently supported are *sin*, *cos*, sin^{-1}, cos^{-1}, *abs*. FixMAGE also determines the fixed point type used in the arguments, the return value types, and the best algorithm to implement the function by exploiting the TAFFO DTA pass's analysis results [33]. After this preliminary step is performed, specialized fixed-point mathematical functions are generated for each call site. Currently, for each trigonometric function, two possible implementations can be chosen: one uses Cordic, the other a LUT table. The original function calls in the program are replaced with a call to the newly generated mathematical functions. If a suitable function had already been generated, the existing function is used instead of generating a new one.

4.2 `Sin` and `Cos`

`Sin` and `Cos` are available with two different algorithms: Cordic and LUT. In the following we outline how the algorithms are implemented.

4.2.1 Cordic `Sin` and `Cos`

The Cordic algorithm for `Sin` and `Cos` is composed of three distinct sections: the Reduction phase, the Computation phase, and the Epilogue phase.

In the Reduction phase, we reduce the input angle to the allowed set of values. In fact, the Cordic algorithm only converges when the following inequality holds:

$$\mid \theta \mid \leq \sum_{k=0}^{\inf} \arctan 2^{-k}$$

The inequality includes the range between $[-\frac{\pi}{2}, \frac{\pi}{2}]$. However, to simplify the implementation, we further reduce the angle to the $[0, \frac{\pi}{2}]$ interval instead. An ad-hoc cycle and a set of constants are used to perform this task. The values of the constants depend on the fixed point format used by the input angle. Therefore, the code generator has to analyze the difference in representation capability between the fixed point format used to represent the angle, and the fixed point format required to represent the biggest constant used by the CORDIC algorithm.

The code generated for this loop can be divided into two main parts. The first part keeps the fixed point in a normalized form. The second part uses the information generated from the normalization to choose which representation of 2π to retrieve. That value is subtracted from the angle until another normalization is needed or the angle is in the right interval. In Listing 9 we show example LLVM-IR code implementing this step for a fixed point argument with 32 bit of size and 26 bits of fraction.

At this point, to conclude the reduction, the angle is brought to the first quadrant. This step is performed using the appropriate trigonometric identity for the generated function. A code example as generated by FixMAGE is shown in Listing 10.

After the Reduction phase, the Computation phase begins. In the computation phase, the tool implements the Cordic algorithm as illustrated in Section 1.3.1. In Listing 11 we show a code example of the computation phases for `sin` and `cos` with 32 bit of size and 26 bits of fraction.

Finally, in the Epilogue phase, the actual return value is computed from the x and y variables previously produced restoring. An example of code is shown in Listing 12.

4.2.2 LUT `Sin` and `Cos`

In the LUT version of Sin and Cos, only the computation phases are changed, and the Cordic algorithm is substituted with a lookup table. Due to the symmetry between `cos` and `sin`, it is possible to use only one lookup table for both operations in order to reduce the code size. To gain more precision at the cost of a slower lookup table, the argument is reduced in a similar way to what is required for the Cordic `sin` and `cos` implementation. The default options generate a table with 2048 entries of the `sin` function values in the range $[0, \frac{\pi}{2}]$. This is configurable with a command-line option.

In Listing 13 we show an example of LUT. The data table is omitted for brevity.

```
  true_greater_zero:                                ; preds = %body
  %21 = load i32, i32* %arg
  %22 = load i32, i32* @zero_arg.26
  %23 = sub i32 %22, %21
  %24 = load i8, i8* %changeSign
  %25 = xor i8 %24, -1
  store i8 %25, i8* %changeSign
  store i32 %23, i32* %arg
  br label %body1
body1:                                            ; preds = %true_greater_zero, %body
  %26 = bitcast [1 x i32]* %pi_2_array to i8*
  %27 = bitcast [1 x i32]* @pi_2_global.26_26 to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 4 %26, i8* align 4 %27, i64 4, i1 false)
  store i32 26, i32* %point_arg
  store i32 26, i32* %point_ret
  store i32 0, i32* %Iterator_pi_2
  br label %cmp_bigger_than_2pi
cmp_bigger_than_2pi:                              ; preds = %bigger_than_2pi, %body1
  %28 = load i32, i32* %Iterator_pi_2
  %29 = getelementptr [1 x i32], [1 x i32]* %pi_2_array, i32 0, i32 %28
  %30 = load i32, i32* %29
  %31 = load i32, i32* %arg
  %32 = icmp sle i32 %30, %31
  br i1 %32, label %bigger_than_2pi, label %shift
bigger_than_2pi:                                  ; preds = %cmp_bigger_than_2pi
  %33 = load i32, i32* %arg
  %34 = sub i32 %33, %30
  store i32 %34, i32* %arg
  br label %cmp_bigger_than_2pi
shift:                                            ; preds = %cmp_bigger_than_2pi
  store i32 0, i32* %Iterator_pi_2
  %35 = load i32, i32* %point_ret
  %36 = load i32, i32* %point_arg
  %37 = icmp eq i32 %36, %35
  br i1 %37, label %body4, label %body3
body3:                                            ; preds = %shift
  %38 = load i32, i32* %point_ret
  %39 = load i32, i32* %point_arg
  %40 = icmp slt i32 %39, %38
  br i1 %40, label %left_shift, label %right_shift
left_shift:                                       ; preds = %body3
  %41 = load i32, i32* %point_arg
  %42 = load i32, i32* %point_ret
  %43 = sub i32 %42, %41
  %44 = load i32, i32* %arg
  %45 = shl i32 %44, %43
  store i32 %45, i32* %arg
  br label %body4
right_shift:                                      ; preds = %body3
  %46 = load i32, i32* %point_ret
  %47 = load i32, i32* %point_arg
  %48 = sub i32 %47, %46
  %49 = load i32, i32* %arg
  %50 = ashr i32 %49, %48
  store i32 %50, i32* %arg
  br label %body4
declare dso_local i32 @printf(i8*, ...) #1
```

Listing 9: FixMage reduction phase to bring the argument into the range $[0, 2\pi]$. Unoptimized LLVM-IR.

```
  in_II_quad:                                      ; preds = %body13
  %69 = load i8, i8* %changeSign
  %70 = xor i8 %69, -1
  store i8 %70, i8* %changeSign
  %71 = load i8, i8* %changefunc
  %72 = xor i8 %71, -1
  store i8 %72, i8* %changefunc
  %73 = load i32, i32* @pi_half.26
  %74 = load i32, i32* %arg
  %75 = sub i32 %74, %73
  store i32 %75, i32* %arg
  br label %body15
body15:                                            ; preds = %in_II_quad, %body13
  %76 = load i32, i32* %arg
  %pi17 = load i32, i32* @pi.26
  %arg_greater_pi = icmp slt i32 %pi17, %76
  %pi_32 = load i32, i32* @pi_32.26
  %77 = load i32, i32* %arg
  %arg_less_pi_32 = icmp slt i32 %77, %pi_32
  %78 = and i1 %arg_greater_pi, %arg_less_pi_32
  br i1 %78, label %in_III_quad, label %body16
in_III_quad:                                       ; preds = %body15
  %79 = load i8, i8* %changeSign
  %80 = xor i8 %79, -1
  store i8 %80, i8* %changeSign
  %81 = load i32, i32* @pi.26
  %82 = load i32, i32* %arg
  %83 = sub i32 %82, %81
  store i32 %83, i32* %arg
  br label %body16
body16:                                            ; preds = %in_III_quad, %body15
  %84 = load i32, i32* %arg
  %pi_3219 = load i32, i32* @pi_32.26
  %arg_greater_pi_32 = icmp slt i32 %pi_3219, %84
  %pi_2 = load i32, i32* @pi_2.26
  %85 = load i32, i32* %arg
  %arg_less_2pi = icmp slt i32 %85, %pi_2
  %86 = and i1 %arg_greater_pi_32, %arg_less_2pi
  br i1 %86, label %in_IV_quad, label %body18
in_IV_quad:                                        ; preds = %body16
  %87 = load i8, i8* %changefunc
  %88 = xor i8 %87, -1
  store i8 %88, i8* %changefunc
  %89 = load i32, i32* @pi_32.26
  %90 = load i32, i32* %arg
  %91 = sub i32 %90, %89
  store i32 %91, i32* %arg
  br label %body18
```

Listing 10: FixMage Reduction Phase to reduce the argument to the first quadrant. Unoptimized LLVM-IR.

```
epilog_loop:                                      ; preds = %start_loop, %body18
  %97 = load i32, i32* %iterator
  %98 = icmp slt i32 %97, 64
  %99 = load i32, i32* %iterator
  %100 = icmp slt i32 %99, 32
  %101 = and i1 %98, %100
  br i1 %101, label %start_loop, label %return_point
start_loop:                                       ; preds = %epilog_loop
  %102 = load i32, i32* %arg
  %103 = icmp sge i32 %102, %94
  %104 = select i1 %103, i32 1, i32 -1
  %105 = load i32, i32* %iterator
  %106 = load i32, i32* %x
  %107 = ashr i32 %106, %105
  %108 = load i32, i32* %iterator
  %109 = load i32, i32* %y
  %110 = ashr i32 %109, %108
  %111 = load i32, i32* %iterator
  %112 = getelementptr [64 x i32], [64 x i32]* %1, i32 %94, i32 %111
  %113 = load i32, i32* %112
  %114 = icmp sgt i32 %104, %94
  %115 = sub i32 %94, %113
  %116 = select i1 %114, i32 %115, i32 %113
  %117 = load i32, i32* %arg
  %118 = add i32 %116, %117
  store i32 %118, i32* %arg
  %119 = sub i32 %94, %110
  %120 = select i1 %114, i32 %119, i32 %110
  %121 = load i32, i32* %x
  %122 = add i32 %120, %121
  store i32 %122, i32* %x
  %123 = sub i32 %94, %107
  %124 = select i1 %114, i32 %107, i32 %123
  %125 = load i32, i32* %y
  %126 = add i32 %124, %125
  store i32 %126, i32* %y
  %127 = load i32, i32* %iterator
  %128 = add i32 %127, 1
  store i32 %128, i32* %iterator
  br label %epilog_loop
```

Listing 11: Cordic rotation mode generated by FixMage. Unoptimized LLVM-IR.

```
return_point:                                    ; preds = %epilog_loop, %Special
%6 = load i32, i32* @zero.26
%7 = load i32, i32* %y
%8 = load i32, i32* %x
%9 = load i8, i8* %changefunc
%10 = icmp eq i8 %9, 0
%11 = select i1 %10, i32 %8, i32 %7
store i32 %11, i32* %arg
%12 = load i32, i32* %arg
%13 = sub i32 %6, %12
%14 = load i32, i32* %arg
%15 = load i8, i8* %changeSign
%16 = icmp eq i8 %15, 0
%17 = select i1 %16, i32 %14, i32 %13
store i32 %17, i32* %arg
%18 = load i32, i32* %arg
%19 = ashr i32 %18, 4
store i32 %19, i32* %arg
%20 = load i32, i32* %arg
br label %end
```

Listing 12: Epilogue generated by FixMage. Unoptimized LLVM-IR.

```
body18:                                          ; preds = %in_IV_quad, %body16
%89 = load i32, i32* %arg
%90 = shl i32 %89, 4
store i32 %90, i32* %arg
%91 = load i32, i32* @zero.26
%92 = load i32, i32* %arg
%93 = load i32, i32* @pi_half_internal_30
%94 = lshr i32 %93, 11
%95 = call i32 @llvm.udiv.fix.i32(i32 %92, i32 %94, i32 19)
%96 = lshr i32 %95, 19
%97 = getelementptr [2048 x i32], [2048 x i32]* @sin_global.30_32, i32 %91, i32 %96
%98 = load i32, i32* %97
store i32 %98, i32* %y
%99 = sub i32 2048, %96
%100 = getelementptr [2048 x i32], [2048 x i32]* @sin_global.30_32, i32 %91, i32 %99
%101 = load i32, i32* %100
store i32 %101, i32* %x
```

Listing 13: LUT of Sin and Cos generated by FixMage. Unoptimized LLVM-IR.

4.3 `ASin` and `ACos`

Just like `Sin` and `Cos`, `ASin` and `ACos` are also available with two different algorithms: Cordic and LUT.

4.3.1 Cordic `ASin` and `ACos`

Cordic `ASin` and `ACos` have a structure similar to their `Cos` and `Sin` counterparts. Due to the narrow range of the input domain ($[-1, 1]$), the reduction phase is composed just by a shift which brings the argument to the best internal representation: giving only two bits to represent the integer part. The Computation phase implements the double rotation methods as explained in section 1.3.2. In Listing 14 we show an example of generated code for `ACos` with as argument a fixed point of 32 bits of size and 29 bits of fraction.

The epilogue phase takes care to convert the optimal internal representation – as generated by the reduction phase – to the requested return fixed point type. Both the reduction and the epilogue phase are rarely generated, as TAFFO often chooses as argument and return type the same one of the internal representation, so there is no need for conversion.

4.3.2 LUT ASin and Acos

The LUT implementation for `ASin` and `ACos` follows a path similar to its counterpart for LUT `sin` and `cos`. The main difference is that the table for `asin` is generated in the range $[0, 1]$. The number of entries for the generated table defaults at 2048, but it is customizable during the program's compilation and through the command-line options. The negative part of the domain is restored with the equality $-asin(-x) = asin(x)$. `ACos` is calculated from `ASin` through the following identity: $acos(x) = \frac{pi}{2} - asin(x)$. This is done during the reduction phase. The computation phase consists of simply a lookup in the table. In the epilogue, the result is shifted back to the required fixed point used as return type.

```
  "Loop entry":                                         ; preds = %"cordic body", %Entry
  %1 = load i32, i32* %i
  %2 = icmp slt i32 %1, 16
  br i1 %2, label %"cordic body", label %end
"cordic body":                                          ; preds = %"Loop entry"
  %3 = load i32, i32* %y
  %4 = icmp sge i32 %3, 0
  %5 = select i1 %4, i32 0, i32 536870912
  %6 = select i1 %4, i32 536870912, i32 0
  %7 = load i32, i32* %t
  %8 = load i32, i32* %x
  %9 = icmp sge i32 %8, %7
  %10 = select i1 %9, i32 %6, i32 %5
  store i32 %10, i32* %d
  %11 = load i32, i32* %i
  %12 = getelementptr [64 x i32], [64 x i32]* @arctan_g.29, i32 0, i32 %11
  %13 = load i32, i32* %12
  %14 = shl i32 %13, 1
  %15 = sub i32 0, %14
  %16 = load i32, i32* %d
  %17 = icmp eq i32 %16, 0
  %18 = select i1 %17, i32 %15, i32 %14
  %19 = load i32, i32* %theta
  %20 = add i32 %19, %18
  store i32 %20, i32* %theta
  %21 = load i32, i32* %i
  %22 = shl i32 %21, 1
  %23 = load i32, i32* %t
  %24 = ashr i32 %23, %22
  %25 = load i32, i32* %t
  %26 = add i32 %25, %24
  store i32 %26, i32* %t
  %27 = load i32, i32* %i
  %28 = load i32, i32* %y
  %29 = ashr i32 %28, %27
  %30 = load i32, i32* %x
  %31 = sub i32 %30, %29
  %32 = load i32, i32* %x
  %33 = add i32 %32, %29
  %34 = load i32, i32* %d
  %35 = icmp eq i32 %34, 0
  %36 = select i1 %35, i32 %33, i32 %31
  store i32 %36, i32* %x1
  %37 = load i32, i32* %i
  %38 = load i32, i32* %x
  %39 = ashr i32 %38, %37
  %40 = load i32, i32* %y
  %41 = add i32 %40, %39
  %42 = load i32, i32* %y
  %43 = sub i32 %42, %39
  %44 = load i32, i32* %d
  %45 = icmp eq i32 %44, 0
  %46 = select i1 %45, i32 %43, i32 %41
  store i32 %46, i32* %y1
  %47 = load i32, i32* %i
  %48 = load i32, i32* %y1
  %49 = ashr i32 %48, %47
  %50 = load i32, i32* %x1
  %51 = sub i32 %50, %49
  %52 = load i32, i32* %x1
```

Listing 14: First Part of Acos Cordic generated by FixMage. Unoptimized LLVM-IR.

```
%53 = add i32 %52, %49
%54 = load i32, i32* %d
%55 = icmp eq i32 %54, 0
%56 = select i1 %55, i32 %53, i32 %51
store i32 %56, i32* %x
%57 = load i32, i32* %i
%58 = load i32, i32* %x1
%59 = ashr i32 %58, %57
%60 = load i32, i32* %y1
%61 = add i32 %60, %59
%62 = load i32, i32* %y1
%63 = sub i32 %62, %59
%64 = load i32, i32* %d
%65 = icmp eq i32 %64, 0
%66 = select i1 %65, i32 %63, i32 %61
store i32 %66, i32* %y
%67 = load i32, i32* %i
%68 = add i32 %67, 1
store i32 %68, i32* %i
br label %"Loop entry"
```

Second Part of Acos Cordic generated by FixMage. Unoptimized LLVM-IR.

```
 Entry:
  %x = alloca i32
  %y = alloca i32
  store i32 %0, i32* %x
  %1 = load i32, i32* %x
  %2 = icmp sge i32 %1, 0
  %3 = select i1 %2, i32 0, i32 536870912
  store i32 %3, i32* %y
  %4 = load i32, i32* %x
  %5 = sub i32 0, %4
  %6 = load i32, i32* %x
  %7 = load i32, i32* %x
  %8 = icmp sge i32 %7, 0
  %9 = select i1 %8, i32 %6, i32 %5
  store i32 %9, i32* %x
  %10 = load i32, i32* %x
  %11 = lshr i32 %10, 18
  %12 = getelementptr [2049 x i32], [2049 x i32]* @asin_global.29_32, i32 0, i32 %11
  %13 = load i32, i32* %12
  store i32 %13, i32* %x
  %14 = load i32, i32* %y
  %15 = icmp eq i32 %14, 536870912
  br i1 %15, label %restore, label %end
restore:                                          ; preds = %Entry
  %16 = load i32, i32* %x
  %17 = sub i32 0, %16
  store i32 %17, i32* %x
  br label %end
end:                                              ; preds = %restore, %Entry
  %18 = load i32, i32* %x
  %19 = load i32, i32* @pi_half_29
  %20 = sub i32 %19, %18
  ret i32 %20
```

Listing 15: Acos LUT generated by FixMage, Unoptimized LLVM-IR.

4.4 Absolute Value

The absolute value function simply implements the naive algorithm. The first part of the generated code checks if the number is negative, and if it is true, its negation is returned.

Due to the small amount of code required to implement this function, instead of showing an example of the generated LLVM-IR, we show the generation algorithm instead to better demonstrate how it is possible to work within the LLVM framework.

The most crucial class used in the generation code is `IRBuilder`, which generates and links the new instructions into the basic blocks.
The main structure of the abs generator is composed by a single block called `entry`. To create the basic block we used the function `Create` of the class `BasicBlock` which takes as input, the context the name and a function in which it will be added. `BasicBlock::Create(context, block name, function)`
The core of the function is composed by a ternary operation in which we control the leading bit to choose if return the argument or his negation. The generation of `Select` follows this structure
`builder.CreateSelect( condition, value return if true, value return if false)` To check the leading bit we insert a shift to the right by a number of bits that depends on the size of the generated function argument types.
`builder.CreateLShr(Value to shift, Amount to shift)` and we compare it with a constant to check if it is equal.
`builder.CreateICmpEQ(first value to compare, second value to compare)`
Finally to get the negation we can use
`builder.CreateSub(first value, second value)`.

If we compose all together, we obtain 16

```
builder.CreateSelect(
   builder.CreateICmpEQ(
       builder.CreateLShr(inst, arg_type->getScalarSizeInBits() - 1),
       llvm::ConstantInt::get(
           llvm::Type::getIntNTy(cont, arg_type->getPrimitiveSizeInBits()),
           1)),

   builder.CreateSub(llvm::ConstantInt::get(
                             llvm::Type::getIntNTy(
                                 cont, arg_type->getPrimitiveSizeInBits()),
                             0),
                         inst)
   inst);
```

Listing 16: FixMage Core of the abs value

```
bool FloatToFixed::createAbs(llvm::Function *newfs, llvm::Function *oldf) {
newfs->deleteBody();
llvm::LLVMContext &cont(oldf->getContext());
// get first basick block of function
auto arg_type = newfs->getArg(0)->getType();
auto ret_type = newfs->getReturnType();
BasicBlock::Create(cont, "Entry", newfs);
BasicBlock *where = &(newfs->getEntryBlock());
IRBuilder<> builder(where, where->getFirstInsertionPt());
//handle unsigned int
if (this->hasInfo(oldf->getArg(0)) &&
    !(this->valueInfo(oldf->getArg(0))->fixpType.scalarIsSigned())) {
  builder.CreateRet(newfs->getArg(0));
  return true;
}

//cast the argument to integer
auto *inst = builder.CreateBitCast(
    newfs->getArg(0),
    llvm::Type::getIntNTy(cont, arg_type->getPrimitiveSizeInBits()));

inst = builder.CreateSelect(
    //check the most significant bit
    builder.CreateICmpEQ(
        builder.CreateLShr(inst, arg_type->getScalarSizeInBits() - 1),
        llvm::ConstantInt::get(
            llvm::Type::getIntNTy(cont, arg_type->getPrimitiveSizeInBits()),
            1)),
//if equal one return the negation
    builder.CreateBitCast(
        builder.CreateSub(llvm::ConstantInt::get(
                              llvm::Type::getIntNTy(
                                  cont, arg_type->getPrimitiveSizeInBits()),
                              0),
                          inst),
        llvm::Type::getIntNTy(cont, arg_type->getPrimitiveSizeInBits()))),
//if equal zero return the original
    inst);

LLVM_DEBUG(dbgs() << "\nType ret" << (ret_type->dump(), " ") << "\n");
LLVM_DEBUG(dbgs() << "\nType arg" << (arg_type->dump(), " ") << "\n");

//handle all the type of return type
if (arg_type->isFloatingPointTy() && ret_type->isFloatingPointTy()) {
  inst = builder.CreateFPCast(inst, ret_type);
} else if (arg_type->isFloatingPointTy() && ret_type->isIntegerTy()) {
  inst = builder.CreateFPToSI(inst, ret_type);
} else if (arg_type->isIntegerTy() && ret_type->isFloatingPointTy()) {
  inst = builder.CreateSIToFP(inst, ret_type);
} else if (arg_type->isIntegerTy() && ret_type->isIntegerTy()) {
  inst = builder.CreateIntCast(inst, ret_type, true);
}
builder.CreateRet(inst);
}
```

Listing 17: FixMage Code that is in charge of generate the abs value

Chapter 5

Extending TAFFO ILP Mixed Precision Capabilities

In its current state of development, TAFFO supports changing only single-precision and double-precision floating-point types to fixed point [39]. We extended the solution adopted by TAFFO, adding support to all the floating-point types supported natively by the LLVM framework. The types supported are reported in Table 5.1

Type	Description
half	16-bit floating-point value
bfloat	16-bit “brain” floating-point value (7-bit significand)
float	32-bit floating-point value
double	64-bit floating-point value
fp128	128-bit floating-point value (113-bit significand)
x86_fp80	80-bit floating-point value (X87)
ppc_fp128	128-bit floating-point value (two 64-bits)

Table 5.1: Types supported by LLVM-IR

For most of the new supported type, it is possible to apply the methods proposed in [39] based on the evaluation of the instructions, and the results were similar to the one illustrated in that work.

In contrast, a new selection strategy was introduced specifically for bfloat types. In fact, for bfloat types it is not possible to measure the time that each possible instruction takes on hardware because only few operations are currently supported for bfloat. On Armv8.6-A [40] conforming CPUs with Neon SIMD extensions, a new set of instructions is available to accelerate the multiplication of matrices with bfloat values. With the AVX512_BF16 extension, Intel also introduced three new instructions to work with bfloat types, which specifically facilitate computing matrix products [41]. The LLVM framework gives access to these special instructions through the use of intrinsics: special functions whose implementation is provided by the compiler itself. To implement support for these new types, we use strategies based on the cost model offered by the LLVM framework [42]. In this way, when full support to bfloat types will be introduced to LLVM, TAFFO will be also able to generate code supporting bfloat. Finally, The ILP algorithm was modified to enable the possibility to specify which operations are possible for each type. This was done because for some low-power architecture, not all the possible arithmetic operations are supported for each type.

5.1 Overview of Previous work

The Data Type Allocation implemented in TAFFO generates an Integer Linear Programming problem and use or-tools [43] to solve the optimization problem. The cost of each mathematical and cast instruction was collected with a benchmark for each different architecture. This profiling phase is external to the TAFFO toolchain, and its output result can be reused.

The Data Type Allocation is generated through optimization of the final objective function:

$$min\,[W_1\frac{C_c}{N_c} + W_3\frac{Ex_c}{NE_x} + W_2\frac{E_c}{Ne}]$$

C_c is the cost of each cast operation introduced in the model. So the model can take into consideration the overhead of the casting operations. The case of zero cost casts, like passing from unsigned integer to integer, is taken into consideration during the profiling phase.

Ex_c represents the error introduced in the program. This is computed with a metric that is called **IEBW**. Informally, the IEBW of a number x represented with a type t is defined as the minimum number of fractional bits that an unrestricted fixed-point (a fixed point with infinite bits after the fractional point) should have to represent the number x with a relative error lesser than or equal to x represented in t.

E_c is the cost of each arithmetic operation introduced by the model, as specified by the profiling. W_1, W_2, W_3 are the weights used to choose which parameter to prioritize. N_c, NE_x, Ne are coefficients chosen to bring the fraction in the range

$[0, 1]$ as the values of Cc, Exc, and Ec are in general uncorrelated and on a different scale.

5.1.1 New Cost model

As already mentioned, a new cost model was introduced in the DTA pass. This was done because it was impossible to profile the bfloat instructions, as hardware manufacturers are still developing support for this type.

Our cost model for bfloat is built around the TCK_RecipThroughput metric given by the LLVM framework. It provides a basic cost estimation to approximate the cost of any IR instruction when lowered to machine instructions. The cost results are unit-less and the cost number represents the reciprocal of throughput of the machine assuming that all loads hit the cache, all branches are correctly predicted. Like with the previously employed method, it's possible to collect all such metrics for all possible arithmetic instructions before the compilation of the program. This has the advantage of decoupling the profiling phase from the compilation phase but comes with a significant disadvantage as we lost the capability of a more fine cost analysis. Running the profiling phase within the compilation allows the LLVM Framework to analyze the cost of instruction inside functions and give a more accurate prediction of the actual cost. We set up TAFFO to work directly with all the cost models and they are reported in Table 5.2 offered by LLVM.

LLVM Cost Model	
TCK_RecipThroughput	Reciprocal throughput
TCK_Latency	The latency of instruction
TCK_CodeSize	Instruction code size
TCK_SizeAndLatency	The weighted sum of size and latency

Table 5.2: LLVM-IR Cost Model

In the current state, the only cost model supported both by ARM and Intel architectures is TCK_RecipThroughput. The other cost models are currently under development and follow the same principle of TCK_RecipThroughput, so when they are going to be fully completed, they will return a unit-less number representing the specific cost of each LLVM-IR targeted instructions. As their interface is already defined, it was possible to integrate them into TAFFO before their release. As we were interested in a metric that can consider the size of the instructions and it's performance, and knowing that we can not utilize TCK_Size, we choose to utilize TCK_RecipThroughput but to adjust the metric for the size of the type utilized. So if we consider a generic instruction like the following:

%mul = fmul float %i, 2.000000e+00

We define $t \in \{supported\ type\}$ and $B_t \in \{0, 1\}$
and calling $Sz(t)$ a function that return the size of the type t
calling $Tk(t)$ a function that return the TCK_RecipThroughput of the type t
$\forall i \in \{supported\ types\}$ an instruction will be added of the form:

$$mathCostObj \mathrel{+}= Sz(i) * Tk(i) * B_i$$

Example of instruction added in the model for the new cost

Only one B_t can be true for each converted instruction. Thus $\sum(B_t) = 1$

5.1.2 IEBW for new data Type

The IEBW for the newly supported data types follows the same schema as other floating-point types. Here we report a quick recap. The absolute error while representing a number in a floating-point data type varies with the value itself. Calling p the precision of the decimal part and e the number of exponent bits of a given floating point number T the fractional part is allocated to exactly $p - 1$ bits due to the normalization process. The number of fractional bits needed for an unrestricted fixed-point to represent T with the same precision is given by:

$$p_{fix} = p_{float} - e_{float} - 1$$

e_{float} is the only parameter that depends on the specific number that is being represented. If the number is in normalized form, it can be calculated as

$$e_{float} = min(log_2(x), e_{max})$$

5.1.3 Constraints for operations

Lastly, we shall add the possibility to select which arithmetic operations are supported for each type. This information is passed to TAFFO as a file that obeys to the following grammar:

$\langle\text{start}\rangle \rightarrow [\langle\text{exp}\rangle\mathbf{,}\ ]^*\langle\text{exp}\rangle \mid \epsilon$
$\langle\text{exp}\rangle \rightarrow \mathbf{N}\langle\text{item}\rangle$
$\langle\text{exp}\rangle \rightarrow \mathbf{\text{-}}\langle\text{stype}\rangle\_\langle\text{item}\rangle$
$\langle\text{exp}\rangle \rightarrow \mathbf{\text{-}}\langle\text{mtype}\rangle\_\langle\text{item}\rangle\_\langle\text{item}\rangle$
$\langle\text{stype}\rangle \rightarrow \mathbf{ADD}|\mathbf{SUB}|\mathbf{MUL}|\mathbf{DIV}|\mathbf{REM}$
$\langle\text{mtype}\rangle \rightarrow \mathbf{CAST}$
$\langle\text{item}\rangle \rightarrow \mathbf{HALF}|\mathbf{FIX}|\mathbf{FLOAT}|\mathbf{DOUBLE}|\mathbf{QUAD}|\mathbf{FP80}|\mathbf{PPC128}|\mathbf{BF16}$

Listing 18: Grammar for constraint on types

N<item> allows to disable all the operations for a particular type. -<stype>_<item> allows to disable specific types of operations for a particular type. -<mtype>_<item>_<item> allows to disable specific types of casts.

In the generation of the model, if N<item> is specified for a specific type T, T is not considered in the set of possible types to use. This implies that there will be no trace of type T in the final model, which is enough to exclude it from the code generation. Instead, if only some operations are disabled when such operations are encountered, TAFFO adds a constraint in the following form:

$$B_T = 0$$

This has the result of disabling the possibility for the boolean variable B_T to select the given type T.

Chapter 6

Experimental evaluation

To validate the results of FixM in a precision tuning context, we partition the embedded systems hardware architectures into two different classes.

The first class of embedded systems microcontrollers is characterized by the absence of the floating point unit (FPU). On this kind of hardware, floating point computation can be handled through an emulation library provided by the compiler.

The second class of hardware consists of embedded systems microcontrollers with an FPU. These microcontrollers are nowadays more common than in the past, but they are more costly and less energy-efficient [44]. In such microcontrollers, the software emulation layer is not required.

Our approach based on the fixed point numeric representation is available in all the cases where basic operations (addition, subtraction, multiplication and division) on integers is supported by hardware. Therefore, we avoid the computational overhead inherent in floating point emulation libraries.

Additionally, the FPU of microcontrollers often does not support hardware-based computation of transcendental functions. A software implementation is used instead. Consequently, our solution's performance improvement can be attributed to FixMAGE and our implementation of such functions.

6.1 Hardware Setup

Each of the two hardware classes is represented by one embedded systems development board. Such boards are described in Table 6.1:

Identifier	**Board Name**	**HW FP**	**CPU core**	**CPU f_{ck}**	**RAM size**
M3	STM3220G-EVAL	No	ARM Cortex-M3	25 MHz	2 MB
M4	STM32F4-Discovery	Yes	ARM Cortex-M4	8 MHz	192 KB

Table 6.1: The list of development boards used for the experimental evaluation of our precision tuning solution.

- Identifier: Name used when referring to the development board
- Board Name: Full board name
- HW FP: whether the board has hardware supports for floating point
- CPU core: Name of the CPU architecture
- CPU f_{ck}: Clock frequency of the microcontroller CPU during the experiments.
- RAM size: Amount of RAM available on the board

All the boards we used in our experimental campaign do not feature any level of data cache between the processing unit and the RAM. Instruction cache is not implemented as well.

6.2 Benchmarks

To test FixM and its capability, we used applications from a well-known state-of-the-art benchmark suite, AxBench [45]. AxBench consists of a set of applications that mimics real-world computational needs. These applications are available both in CPU-based and GPU-based implementations. For the sake of our tests, only the CPU version is considered. The applications within AxBench are very good targets of approximate computing, as they focus on algorithms where it is commonly applied.

It is crucial in the approximate computing field to rely on a solid quality metric to measure the technique's functional side effects. For this reason, AxBench provides an error metric for each of the applications it contains. From the multitude of available benchmarks within AxBench, we choose the only two that contain only trigonometric functions: InverseK2J and FFT. Additionally, we evaluated FixM on FBench [46], a synthetic benchmark for floating-point performance. Lastly, we test FixM within a real code scenario. We tested it on an implementation of

Field Oriented Control (FOC)[47]. FOC is an industry-standard for controlling induction motors and other AC-based motors.

In the following we describe these benchmarks in more detail.

InverseK2J

This algorithm finds application in the workload performed by industrial robots. In such robots, the most common kind of implement is a two-joint arm. In this case, two stepper motors control the movement of a joint each, and the instructions sent to the motors take the form of angle differences. The actual computations of the position of the arm are performed in Cartesian coordinates. This benchmark implements the conversion from the cartesian to polar coordinates and vice-versa.

FBench

FBench is a synthetic benchmark meant for measuring the floating point performance of hardware architectures. The benchmark implements the ray-tracing algorithms to simulate the behavior of 100 light rays passing through a telescope lens. The retracing algorithms rely heavily on trigonometric functions, thus it is a good performance test.

FFT

FFT is often used in the signal-processing domain, and it is used to transform a signal from the time domain to the frequency domain. FFT applications range from audio engineering to communication networks. The AxBench FFT benchmark implements an algorithm known as the in-place radix-2 Cooley-Tukey Fast Fourier Transform. The benchmark computes a single transform of a rectangular wave of period K and duty cycle 1%, over a window of size K

FOC

FOC targets induction motors or permanent magnet synchronous motors. In these motors, the drive coils are mounted on the stator, and the rotor is free to rotate around the coils. Motion is achieved by the attraction or repulsion of a permanent magnet. The current passing through the coils in the stator produces a magnetic field. The output of the FOC control equations is the voltage to be applied to these coils in the function of time.

6.2.1 Software Setup

All the benchmarks tested on the boards were linked with the runtime libraries provided by the board manufacturer to initialize the hardware. The benchmarks were run on bare-metal, no supporting operating system was used. The time sampling method was based on a hardware counter which provides a resolution of 1 millisecond. To allow comparing FixM to existing approaches, we compiled the applications using three different configurations each. For all configurations, unless where we specify otherwise, the compiler used was GCC, version 6 and the C standard library employed is newlib [48] version 2.5.0. The first configuration compiles without performing any precision tuning task. The second configuration is compiled via TAFFO without FixM. This configuration exploits the 32-bit fixed point representations with dynamic bit partitioning and the mathematical functions' wrapping. Alongside TAFFO, we used the LLVM compiler infrastructure, with clang version 8.0.1 as compiler. The C standard library employed is the same version of newlib. The third configuration is compiled with FixM. Thus, benchmarks use precision-tuned fixed point numeric representation thorough the entire computation, and trigonometric functions are provided by FixMAGE. We used the same versions of LLVM, clang and newlib as in the second configuration. Each benchmark was compiled with two different optimization levels. One run employs the `-O3` option, while the other is set to `-Os`. These flags' meaning is an industry-standard that does not significantly vary across the compilers we used in our experimental campaign. `-O3` exploits aggressive optimization aimed to reduce execution times. `-Os` instead performs slightly less aggressive optimizations preset, aimed at a smaller code size.

6.2.2 Time and Accuracy analysis

First, we load the software, produced as described in the previous section, onto the boards. The software on the boards is executed 100 times consecutively. For each execution, we collect the execution time of the computational kernel and the results. A reference version of the benchmark was also run on a computer equipped with a CPU Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz and used as the baseline. The results of the computations are represented as a single vector of real numbers. This vector does not change across different runs. The execution time is averaged over the 100 separate runs of the experiment. To compare the execution times between the experiments, we use the Speedup metric (S). Given two execution times t_a and t_b , the Speedup of t_a with respect to t_b is

$$100 * (\frac{t_a}{t_b} - 1)$$

When referring to the speedup of one software configuration to another, we mean the speedup between their execution times. The Average Absolute Error (AE) and Average Relative Error (RE) metrics are used to compare the computation results. Calling A and B two vectors with the same number of elements, the two error metrics are computed as such:

$$AE = avg_i(|a_i - b_i|)$$

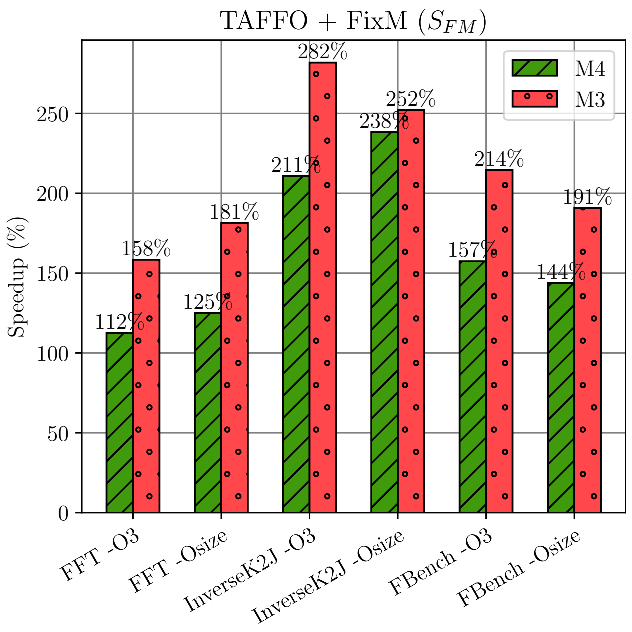
$$RE = \frac{AE}{avg_i(| b_i |)}$$

Image 6.1 shows the speedups obtained via FixM, compared with the speedups due to the precision tuning performed by TAFFO alone. Both speedups are referred to as the floating point version.

Benchmark	Opt	M3			M4		
		t_{float}	t_{TAFFO}	t_{FixM}	t_{float}	t_{TAFFO}	t_{FixM}
InverseK2J	-O3	0.168	0.166	0.044	0.115	0.116	0.034
	-Os	0.169	0.166	0.048	0.115	0.117	0.037
FFT	-O3	0.416	0.409	0.161	0.238	0.276	0.112
	-Os	0.422	0.408	0.150	0.243	0.275	0.108
FBench	-O3	0.217	0.200	0.069	0.139	0.138	0.054
	-Os	0.218	0.200	0.075	0.139	0.138	0.057

Table 6.2: FixM Execution time measurements obtained during the experiments.

Independently from the hardware class, FixM brings a substantial benefit in terms of execution time and size for all three benchmarks. In fact, we see high speedups for both boards, M3 and M4. All the speedup achieved is attributable to the FixM approach. In fact, the execution times for the floating-point versions and the TAFFO-optimized fixed point version without FixM are very similar. Hence, the speedup achieved by applying precision tuning without FixM is at most 9% and, in the worst case, we obtain a slow down of at most 13.8% on the M4 board as reported in Figure 6.1. The slowdown for the M4 board is attributable to



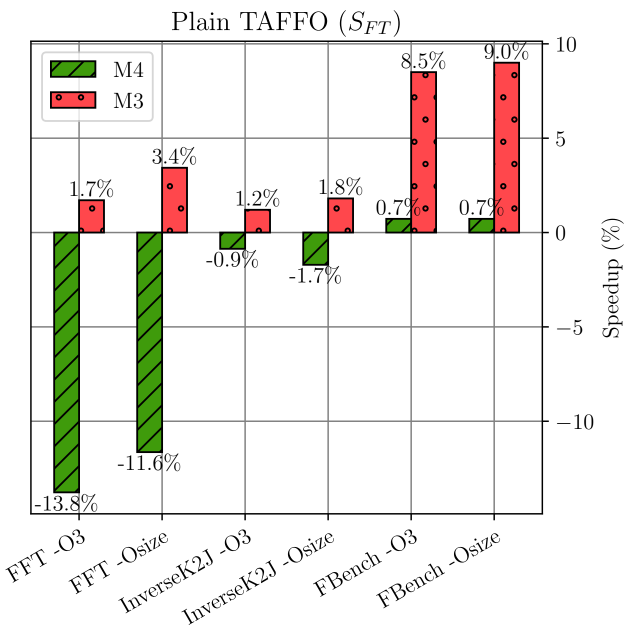


Figure 6.1: Speedup of different TAFFO optimized benchmarks with respect to the floating point baseline, with and without FixM, using different compiler optimization directives.

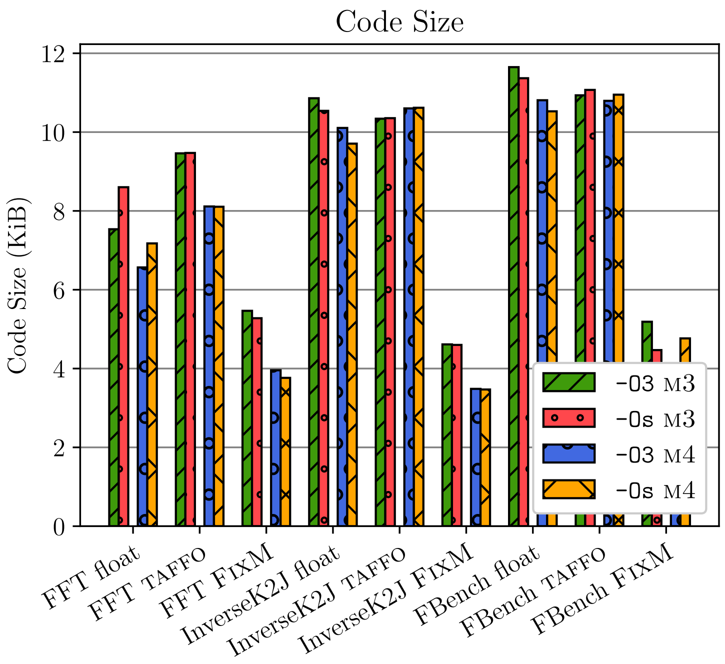


Figure 6.2: Code size of the dynamic fixed point versions w.r.t. the floating point baseline, with FixM and without TAFFO), using different compiler optimization directives. Only the size of the body of the computational loop is considered.

the low benefits provided by the fixed point numeric representation for microcontroller CPUs with floating-point support. From the large difference in execution time between the TAFFO and the FixM configurations, we also conclude that the trigonometric computations constitutes a large portion of the execution time. FFT receives the least benefit from FixM as it relies only on sine and cosine. The implementation of sine and cosine with CORDIC is slower than the implementation of arcsin and arccos, as it needs a higher number of iteration to converge. From the code size graph of Figure 6.2, we notice that FixM produces a smaller program than the others. On microcontroller CPUs we employed, instruction-level parallelism is limited. Therefore, the code size tends to have a direct correlation with the execution time.

To confirm that the speedup is not caused only by code size reductions, we can observe the graph 6.3 where the generation of arcsin and arccos are suppressed. As we can see, we achieve good speedup, even without a code size decrease. Thus, we confirm that the speedup is caused by code size reductions and the increased

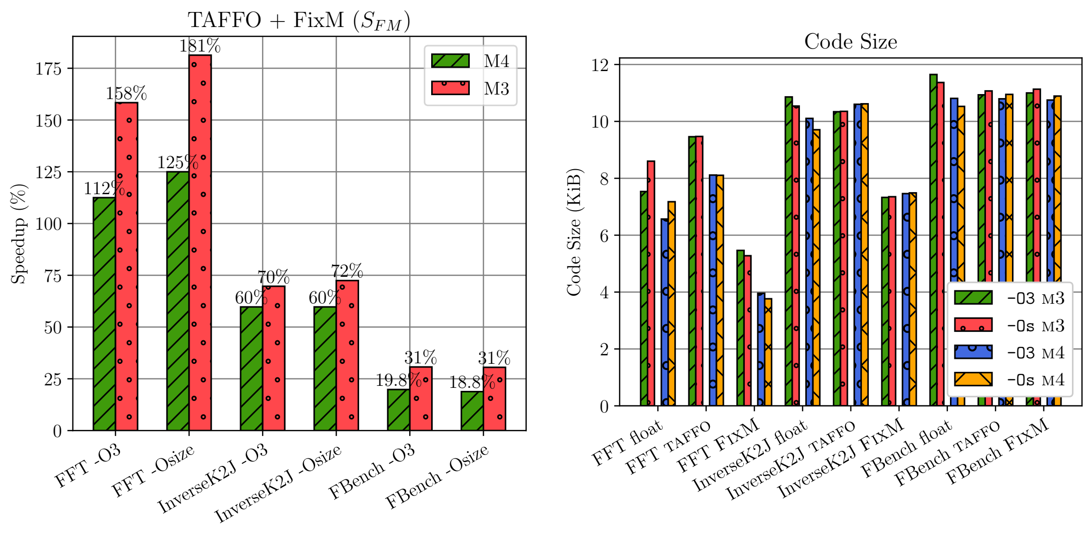


Figure 6.3: Speed up and size of the benchmark with arcsin and arccos generation disabled.

efficiency of the functions generated by FixMAGE. Another observation we make on FFT is that the `-Os` compilation option allows the fixed point based approach to achieve greater speedup. By examining the compiled code, we determine that this code size reduction is caused by inlining the code generated by FixMAGE. The last observation we make is that FixM achieves lesser speedups on the M4 board. This is expected, since standard floating-point trigonometric function implementations are faster on the Cortex-M4 architecture, as it supports floating point data types in hardware.

Benchmark	AE TAFFO	AE FixM	RE TAFFO	RE FixM
InverseK2J	2.56×10^{-7}	1.31×10^{-7}	3.74×10^{-7}	1.91×10^{-7}
FFT	5.29×10^{-5}	5.28×10^{-5}	9.96×10^{-6}	9.95×10^{-6}
FBench	2.89×10^{-2}	4.21×10^{-3}	3.06×10^{-3}	4.47×10^{-4}

Table 6.3: Absolute and relative error of the fixed point versions w.r.t. the reference baseline.

Now, let us consider the absolute error and the relative error of the FixM configuration with respect to the floating-point configuration, shown in 6.3. We show a single data set since for both M4 and M3 boards, the errors were the same independently from the compilation options (`-O3` as opposed to `-Os`). The errors do not change significantly with respect to the figures we obtain with TAFFO alone, without FixM. Therefore FixM does not introduce additional errors on its own. The more complex algorithm explains the higher errors of FFT and FBench with respect to InverseK2J.

6.2.3 Energy consumption

Benchmark	Opt	M3			M4		
		E_{float}	E_{TAFFO}	E_{FixM}	E_{float}	E_{TAFFO}	E_{FixM}
InverseK2J	-O3	62.82	62.99	25.60	48.20	55.89	22.57
	-Os	62.88	62.83	24.00	49.09	54.73	21.92
FFT	-O3	26.21	23.90	13.76	17.37	17.86	11.45
	-Os	26.28	24.24	14.11	17.14	18.02	11.52
FBench	-O3	26.80	26.06	16.09	22.60	22.79	14.00
	-Os	26.87	26.31	15.68	22.66	23.17	14.29

Table 6.4: Total energy draw for the execution of each experiment. All figures are in mJ.

We also measured the average supply voltage and current draw of the microcontroller during the duration of the experiment and reported them in the Table 6.4. These measurements were performed using an RS PRO IDM-8351 digital multimeter [49] with a current measurement resolution of $100nA$ and a voltage measurement resolution of $1\mu V$. The multimeter was connected in series with the microcontroller by exploiting specifically provided test points on the evaluation boards to exclude the current draw of all supporting hardware. The samples were then filtered to remove those that were not taken during the 100 runs of the experiments and averaged using the arithmetic mean. The energy draw was computed using the formula $E = VIt$, where V is the supply voltage of the microcontroller (5 Volt), I is the average current, and t is the execution time of the benchmark. We observe that the benchmarks compiled with FixM require significantly less energy than the others. This is primarily due to the reduced execution time. This improvement also extends to embedded platforms with floating point hardware support.

6.2.4 Different Algorithms analysis

We evaluate the impact of different generated algorithms by FixMAGE on the FOC application. The FOC was compiled with several different FixM settings to evaluate the trade-off of using the implementation based on look-up-tables as opposed to the implementation based on CORDIC:

C2 This configuration always uses CORDIC for both trigonometric calls

L1C1 This configuration uses a LUT for the first trigonometric call, but a CORDIC implementation for the second call

C1L1 This configuration uses a LUT for the second trigonometric call, but a CORDIC implementation for the first

L2 This configuration always uses LUTs for both trigonometric calls

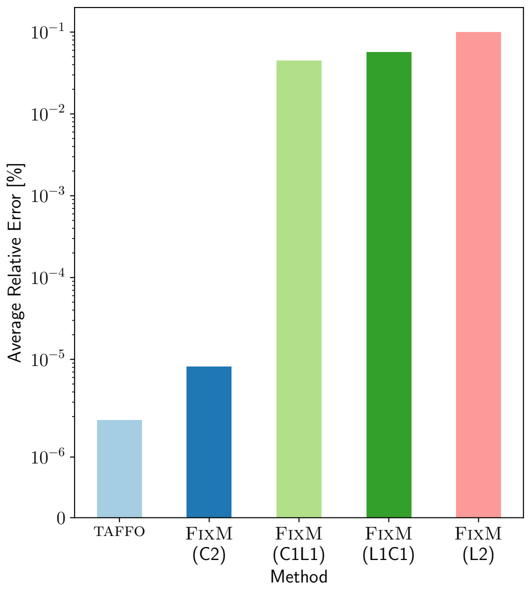


Figure 6.4: Comparison of execution times by platform and approximation method.

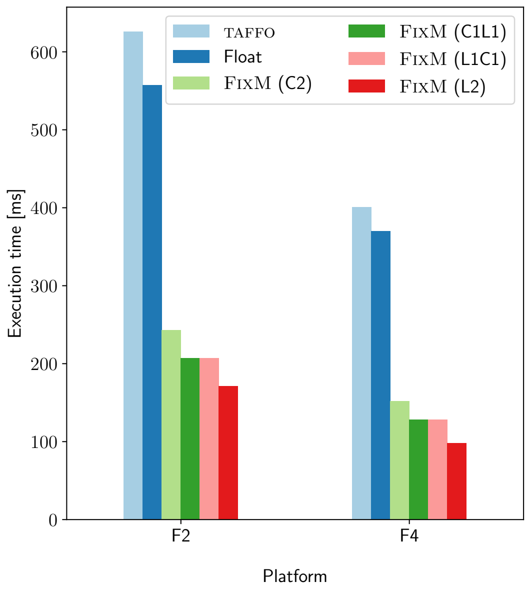


Figure 6.5: Comparison of average percentage relative errors by approximation method.

From Figure 6.4 Figure 6.5 and Table 6.5 we can observe that FixM using CORDIC provides good performance and small code size at minimum accuracy loss, the use of LUTs provides further performance at the expense of both precision

Method	**F2**			**F4**		
	Size	Size of Benchmark	Size of Constant	Size	Size of Benchmark	Size of Constant
Float	7424	1892	672	6840	1892	696
TAFFO	8912	832	688	8216	900	704
FixM (C2)	6000	2538	476	5132	2548	496
FixM (L1C1)	6404	2204	8668	5608	2252	8688
FixM (C1L1)	6452	2158	8668	5620	2240	8688
FixM (L2)	6048	1804	8412	520	1832	8432

Table 6.5: Size of the generated code in bytes split across code and constants. The code figures are further split into the full code, considering the platform specific support libraries size and the FOC bench.

and memory footprint. Instead, Mixed solutions (using CORDIC for one operation and LUT for the other) provide an intermediate point that is not Pareto-dominated by either FixM with CORDIC or FixM with both LUTs. So the addition of other supported algorithms gives FixM a valuable addition that expands the design space, providing the designer with much-needed choices, which can be exerted to cope with specific application constraints.

6.2.5 Comparison with the State of the Art

Several solutions in state-of-the-art have made use of CORDIC algorithms to implement trigonometric functions. However, most implementations share the same approach. We distinguish two main categories: hardware and software implementations. The hardware implementation comes from the generation of circuit descriptions for programmable devices, for example, FPGAs. High-Level Synthesis (HLS) is the task that aims at automatically generating these hardware descriptions starting from a high-level programming language such as C or C++. Complex mathematical functions are traditionally performed in floating-point. This is also true for HLS tools where most arithmetic functions already implemented do not support fixed point versions. However, floating point requires plenty of additional space and energy when synthesized on an FPGA, making their exploitation impractical in a certain application. It has been proven [50] that a fixed point of trigonometric function using the CORDIC algorithm significantly lowers the resource utilization on FPGAs at the cost of a minor error. In the software implementations, we find embedded systems applications [51] and hardware simulation tools [52]. However, the work scope is often narrowed to a specific use case, impairing the solution's generality. Thus, we consider precision tuning frameworks where the problem is tackled in a general way. Those that support fixed point data types do not present significant differences from the arithmetic point of view [53]. We compare the functional behavior of FixMAGE with a well-known hardware simulation environment (Matlab Simulink), and with an industrial library for hardware simulation and development (Xilinx Vivado).

Errors are reported in figure 6.6 as a difference to a reference version we computed using the arbitrary precision arithmetic mpmath library [54] of the Python [55] programming language. We compute the sin/cos value for each value in the range $[\frac{-\pi}{2}; \frac{\pi}{2}]$ with steps of 64, passing each argument as a signed fixed point datatype.

In the evaluation, we choose to use a range of popular bit widths (32, 16, 8 bits, respectively with 29, 13, and 5 fractional bits). We set the number of iterations of the CORDIC algorithm to the total number of bits, which guarantees the maximum precision. Using the same fixed point representations, we evaluated the sine and cosine CORDIC implementation provided by the System Generator for DSP by Xilinx. This tool is a Simulink plugin that provides premade blocks to accelerate FPGA development. It implements a fixed bit-width arithmetic module as it is designed to simulate hardware implementations. MATLAB data series rely on the CORDIC implementation of Simulink embedded in Matlab. This version uses the default simulation parameters of the environment that uses a round to nearest policy, which yields more accurate results. However, this implicitly exploits an intermediate representation that is not restricted to the given bit width. FixMAGE implements the truncate rounding mode, thus emulating the behavior of the most common hardware.

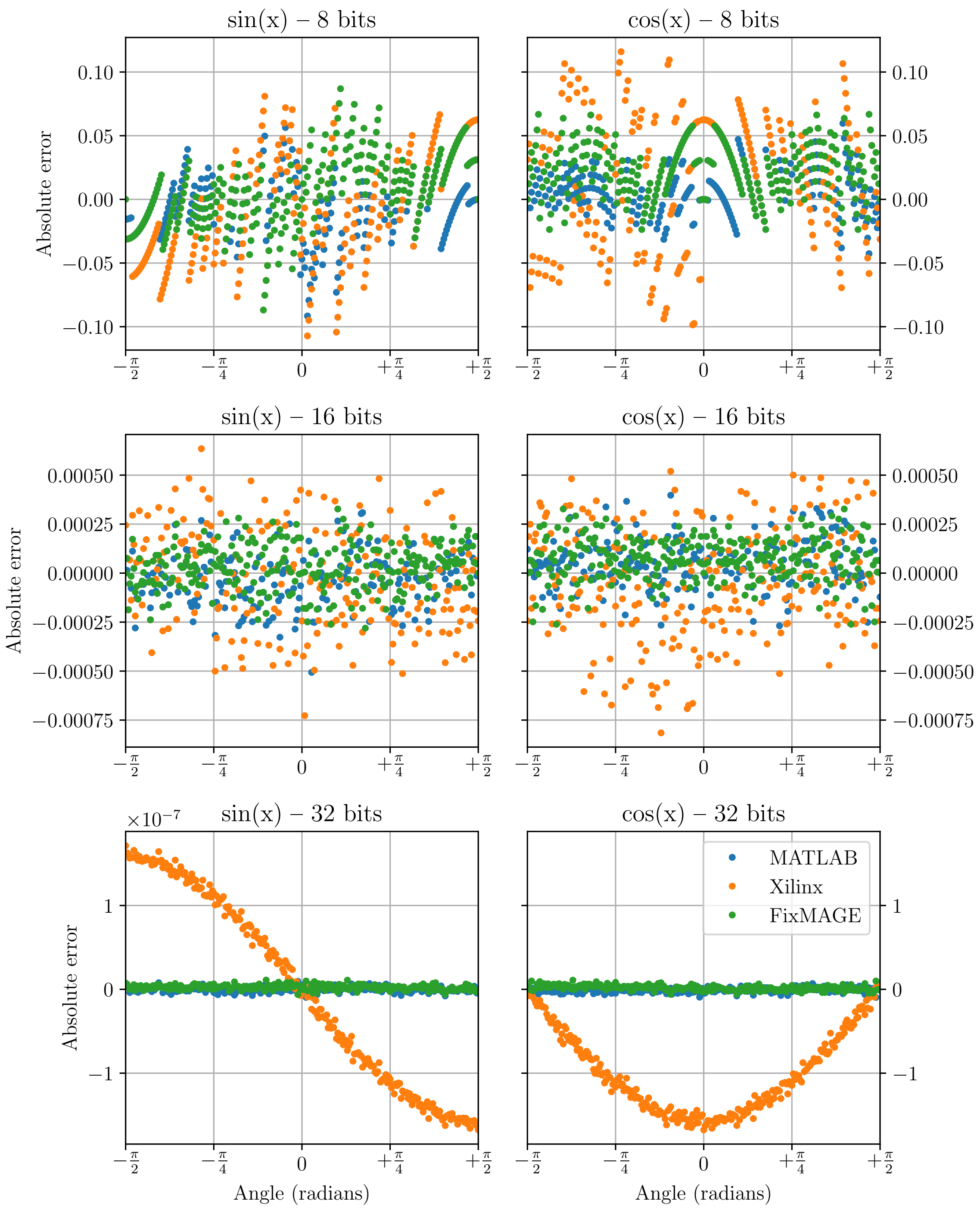


Figure 6.6: Absolute error of sin and cos implementations

Chapter 7

Evaluation of the new DTA algorithms

In this section, we present a preliminary implementation of the new DTA algorithm, and its experimental evaluation. As the solution is based on TAFFO, each test has been edited to insert the required annotations. The toolchain used is based on LLVM 11 to have access to the new bfloat type. This section will present the results obtained by the analysis of the LLVM-IR generated by the optimization of the benchmarks used for the evaluation process.

7.1 Benchmarks

The benchmark suite used is the Polybench [56] test suite version 4.2.1. This benchmark is composed of different programs written in C, each implementing a different algorithm. The algorithms chosen for each file belong to a wide spectrum of possible applications. This makes Polybench the benchmark of choice for evaluating experimental compilation optimizations in sufficiently realistic scenarios. As an example, some benchmarks are related to linear algebra, like **2mm**, which implements 2 matrix multiplications, or **atax** for matrix transposition and vector multiplication. Other benchmarks are related to multimedia, like **fdtd-2d** which implements the Fourier transform. Polybench offers the possibility to tune the amount of memory to allocate for every single test in order to be able to adapt to multiple targets, even the most memory-constrained ones, such as embedded systems. It also performs extra operations such as cache flushing before the kernel execution, and features syntactic constructs to prevent any dead code elimination on the kernel to ensure a fair comparison between different compilers and source code analyses.

7.2 Software setup

The host machine used to compile the benchmarks is a desktop PC running Ubuntu 20.04.2 LTS [57] with an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz. We employed version 11 of the LLVM toolchain, on which the new bfloat type was introduced. We used a debug LLVM build to ease the coding process, but for general use we advise to use a release version because of the compilation speed improvements it enables. TAFFO was also changed to support LLVM 11 and compiled against it.

Another required dependency is the python library ortools [43], which is used to solve the linear optimization problem generated by the DTA.

Finally, we used an auxiliary tool, written in C++ with the regex BOOST library [58], to extract, collect and organize the information from LLVM-IR and python3[55]. Since a TAFFO compilation is composed of several steps which are performed in a particular order, a helper script has been written to automatically execute each LLVM pass in the correct order and with the correct parameters.

Finally, to support the other modifications performed to the DTA pass, we introduced a new command-line argument, named `-instructionsetfile`, for specifying which operations are supported by the target hardware.

All the benchmarks were compiled with IEBW weight set to 50. For each benchmark, we considered eight different compilation configurations:

bfloat	bfloat, single, double and fixed point types enabled
bfloatnoSub	like **bfloat**, with sub instructions disabled
bfloatnoAdd	like **bfloat**, with add instructions disabled
bfloatnoRem	like **bfloat**, with rem instructions disabled
bfloatnoDiv	like **bfloat**, with div instructions disabled
bfloatnoMul	like **bfloat**, with mul instructions disabled
float	only floating point types enabled
fix	only fixed point types enabled

7.3 Result analysis

In this section, we present the results obtained processing the data generated from the execution of the benchmarks. All the tables report the number of occurrences of each operation for all the different compilation methods. In the tables, only the `alloca` and `global` generated by TAFFO are reported. Also, for this special operations the number reported represents the number of operations needed to allocate the the same amount of memory if it was not possible to allocate more than one variable in a single operation. So if in the LLVM-IR an instruction of the type `global [220 x [210 x double]]` is encountered and it's the only one global that allocate double the generated table will have an entry for `global Float` of 46200

As we can see from benchmark **covariance** 7.1, TAFFO could transform the global float to global bfloat and adjust the operation when required. In some cases, like in **bicg** 7.2, TAFFO chose not to convert from float to bfloat to preserve the precision. Choosing a different value for the IEBW weight would lead to a different result. In most cases, TAFFO chooses to use only one type for storage, and it is usually bfloat due to its small size. This is not always true in fact, in benchmark **syr2k** 7.3 we have both the allocation of `bfloat` and `float`. In this case, the cost of the conversion from bfloat to float to do multiplication exceeds the benefit brought by using the smaller type during the allocation.

We can observe that in 14 benchmarks of 22, the floating point data memory allocations were substituted with bfloat typed ones, leading to a size reduction of roughly 50%. Also, in the four benchmarks where the storage allocation was not changed, but some bfloat instructions were used, we could nonetheless obtain a benefit. In fact, knowing that the bfloat instructions can lead to a speed-up ranging from 5% to 32% [59] 16,5% (due to the increased chance for parallelization), this can be enough to overcome the cost of the cast from float introduced to use them.

All the data collected from the benchmarks are reported in Appendix A.

covariance

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	17	17	17	17	17	17	17	17
bitcast Bfloat Bfloat	0	6	6	6	6	6	6	0
bitcast Double Double	2	2	2	2	2	2	2	2
bitcast Float Float	6	0	0	0	0	0	0	6
fadd Bfloat	0	2	0	2	2	2	2	0
fadd Float	2	0	2	0	0	0	0	2
fdiv Bfloat	0	3	3	0	3	3	3	0
fdiv Float	3	0	0	3	0	0	0	3
fmul Bfloat	0	10	10	10	0	10	10	0
fmul Double	8	8	8	8	18	8	8	8
fmul Float	10	0	0	0	0	0	0	10
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	0	3	0	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptosi Bfloat Integer	0	4	4	4	4	4	4	0
fptosi Double Integer	5	5	5	5	5	5	5	5
fptosi Float Integer	4	0	0	0	0	0	0	4
fptrunc Float Bfloat	0	0	0	3	0	0	0	0
global Bfloat	0	124800	124800	124800	124800	124800	124800	0
global Float	124800	0	0	0	0	0	0	124800
global Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
or Integer	3	4	4	4	4	4	4	3
sdiv Integer	2	2	2	2	2	2	2	2
sitofp Integer Bfloat	0	5	5	5	5	5	5	0
sitofp Integer Double	3	3	3	3	3	3	3	3
sitofp Integer Float	5	0	0	0	0	0	0	5
sub Integer	3	3	3	3	3	3	3	3
trunc Integer Integer	2	2	2	2	2	2	2	2
urem Integer	1	1	1	1	1	1	1	1

Table 7.1: covariance benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

bicg

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	14	14	14	14	14	14	14	14
fmul Double	11	11	11	11	11	11	11	11
fptosi Double Integer	4	4	4	4	4	4	4	4
global Double	800	800	800	800	800	800	800	800
global Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
sdiv Integer	1	1	1	1	1	1	1	1
shl Integer	3	3	3	3	3	3	3	3
sitofp Integer Double	2	2	2	2	2	2	2	2
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	5	5	5	5	5	5	5	5
udiv Integer	3	3	3	3	3	3	3	3
uitofp Integer Double	3	3	3	3	3	3	3	3
urem Integer	5	5	5	5	5	5	5	5
zext Integer Integer	2	2	2	2	2	2	2	2

Table 7.2: bicg benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

syr2k

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	22	19	19	19	22	19	19	22
and Integer	4	2	2	2	4	2	2	4
bitcast Bfloat Bfloat	0	0	0	0	8	0	0	0
bitcast Float Float	14	0	0	0	0	0	0	14
fadd Bfloat	0	2	0	2	2	2	2	0
fadd Float	4	0	2	0	2	0	0	4
fmul Bfloat	0	12	12	12	0	12	12	0
fmul Float	18	0	0	0	15	0	0	18
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	3	0	7	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptrunc Float Bfloat	0	0	1	0	6	0	0	0
global Bfloat	0	153600	153600	153600	105600	153600	153600	0
global Float	153600	0	0	0	48000	0	0	153600
lshr Integer	1	0	0	0	1	0	0	1
mul Integer	3	3	3	3	3	3	3	3
or Integer	4	3	3	3	4	3	3	4
shl Integer	3	3	3	3	3	3	3	3
trunc Integer Integer	4	4	4	4	4	4	4	4
udiv Integer	3	3	3	3	3	3	3	3
uitofp Integer Bfloat	0	3	3	3	2	3	3	0
uitofp Integer Float	3	0	0	0	1	0	0	3
urem Integer	4	4	4	4	4	4	4	4

Table 7.3: syr2k benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

Chapter 8

Conclusion

In this thesis, we presented a novel technique to approach mathematical function optimization in a precision tuning framework, minimizing the amount of additional code generated with respect to current approaches. We were able to achieve speedups up to approximately 282% on a microcontroller based embedded system, with a negligible cost in terms of error. As a result, we saved as much as 60% of the energy required to perform the benchmarks. We demonstrated our approach on a subset of commonly used functions, namely sin, cos, acos, asin, but it is easily extended to the whole range of trigonometric and hyperbolic functions. Future developments include the implementation of the rest of the Libm and an exploration of different architectural platforms. We also extend the set of supported types for precision tuning within TAFFO. We made it capable of using the cost model of LLVM and added the ability to exclude certain operations during the code generation. This work has been evaluated on a set of standard benchmarks, proving its effectiveness and readiness for when full support of the bfloat type will be released. Afterward, the approach can be expanded to support more exotic data types. Finally, this solution can also be combined with some other techniques like dynamic recompilation to build different versions of the same kernel, which automatically adapt it to different input properties.

List of Figures

List of Tables

List of Listings

Bibliography

[1] IEEE. "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754-1985* (1985), pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928.

[2] Microchip. *megaAVR Data Sheet*. 2018. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-32%208%20-PDS-DS40002061A.pdf.

[3] STMicroelectronic. *STM32F205xx STM32F207xx Data Sheet*. 2019. URL: https://www.st.com/resource/en/datasheet/stm32f205rb.pdf.

[4] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." In: *ACM Computing Surveys* 23.1 (1991), p. 8. DOI: 10.1145/103162.103163.

[5] J. E. Volder. "The CORDIC Trigonometric Computing Technique". In: *IRE Transactions on Electronic Computers* EC-8.3 (1959), pp. 330–334. DOI: 10.1109/TEC.1959.5222693.

[6] J. E. Volder. "The Birth of Cordic". In: *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology* 25 (2000), pp. 101–105. DOI: 10.1023/A:1008110704586.

[7] Jean-Michel Muller. *Elementary Functions*. Birkhäuser Boston, 2006. DOI: doi.org/10.1007/b137928.

[8] D. Cochran. "Algorithms and accuracy in the HP-35". In: *hp journal online* 23 (1972), pp. 10–11.

[9] Rafi Nave. "Implementation of transcendental functions on a numerics processor". In: *Microprocessing and Microprogramming* 11.3 (1983). Developments in Industry, Business and Education, pp. 221–225. ISSN: 0165-6074. DOI: https://doi.org/10.1016/0165-6074(83)90151-5.

[10] N. Takagi. "Studies on hardware algorithms for arithmetic operations with a redundant binary representation". PhD thesis. Kyoto University, Japan: Dept. Info. Sci., 1987.

[11] Naofumi Takagi, Tohru Asada, and Shuzo Yajima. "A hardware algorithm for computing sine and cosine using redundant binary representation". In: *Systems and Computers in Japan* 18.8 (1987), pp. 1–9. DOI: `https://doi.org/10.1002/scj.4690180801`.

[12] N. Takagi, T. Asada, and S. Yajima. "Redundant CORDIC methods with a constant scale factor for sine and cosine computation". In: *IEEE Transactions on Computers* 40.9 (1991), pp. 989–995. DOI: `10.1109/12.83660`.

[13] J. M. Delosme. "A processor for two-dimensional symmetric eigenvalue and singular value arrays". In: *In 21st Asilomar Conference on Circuits, Systems and Computers.* 1987, pp. 217–221.

[14] Tobias Achterberg and Roland Wunderling. "Mixed Integer Programming: Analyzing 12 Years of Progress". In: Jan. 2013, pp. 449–481. ISBN: 978-3-642-38188-1. DOI: `10.1007/978-3-642-38189-8_18`.

[15] G.B. Dantzig. "Programming in a linear structure". In: (1948).

[16] A. Hoffman, M. Mannos, D. Sokolowsky, and N. Wiegmann. "Computational Experience in Solving Linear Programs". In: *Journal of the Society for Industrial and Applied Mathematics* 1.1 (1953), pp. 17–33. ISSN: 03684245. URL: `http://www.jstor.org/stable/2099061`.

[17] *IBM 701.* URL: `https://www.ibm.com/ibm/history/exhibits/701/701_intro.html`.

[18] *IBM 704.* URL: `https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html`.

[19] C. E. Lemke. "The dual method of solving the linear programming problem". In: *Naval Research Logistics Quarterly* 1.1 (1954), pp. 36–47. DOI: `https://doi.org/10.1002/nav.3800010107`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800010107`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800010107`.

[20] N. Karmarkar. "A New Polynomial-Time Algorithm for Linear Programming". In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 302–311. ISBN: 0897911334. DOI: 10.1145/800057.808695. URL: https://doi.org/10.1145/800057.808695.

[21] N. Hooda and O. Damani. "A System for Optimal Design of Pressure Constrained Branched Piped Water Networks". In: *XVIII International Conference on Water Distribution Systems, WDSA2016*. Vol. 186. Procedia Engineering. 2017, pp. 349–356. DOI: https://doi.org/10.1016/j.proeng.2017.03.211. URL: https://www.sciencedirect.com/science/article/pii/S1877705817313607.

[22] Stefano Cherubin and Giovanni Agosta. "Tools for Reduced Precision Computation: a Survey". In: *ACM Computing Surveys* 53.2 (Apr. 2020). ISSN: 0360-0300. DOI: 10.1145/3381039.

[23] Phillip Stanley-Marbell, Armin Alaghi, Carbin. Michael, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. *Exploiting Errors for Efficiency: A Survey from Circuits to Algorithms*. 2018. arXiv: 1809.05859 [cs.AR].

[24] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. "Automatically Adapting Programs for Mixed-Precision Floating-Point Computation". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. Association for Computing Machinery, 2013, pp. 369–378. DOI: 10.1145/2464996.2465018. URL: https://doi.org/10.1145/2464996.2465018.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: 40.6 (June 2005), pp. 190–200. ISSN: 0362-1340. DOI: 10.1145/1064978.1065034. URL: https://doi.org/10.1145/1064978.1065034.

[26] Hui Guo and Cindy Rubio-González. "Exploiting Community Structure for Floating-Point Precision Tuning". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New

York, NY, USA: Association for Computing Machinery, 2018, pp. 333–343. ISBN: 9781450356992. DOI: 10.1145/3213846.3213862. URL: https://doi.org/10.1145/3213846.3213862.

[27] Eva Darulova, Einar Horn, and Saksham Sharma. "Sound Mixed-Precision Optimization with Rewriting". In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. 2018, pp. 208–219. DOI: 10.1109/ICCPS.2018.00028.

[28] LLVM. *CallGraphWrapperPass*. 2021. URL: https://llvm.org/doxygen/classllvm_1_1CallGraphWrapperPass.html.

[29] LLVM. *DependenceAnalysisWrapperPass*. 2021. URL: https://llvm.org/doxygen/classllvm_1_1DependenceAnalysisWrapperPass.html.

[30] LLVM. *SCCPPass*. 2021. URL: https://llvm.org/doxygen/classllvm_1_1SCCPPass.html.

[31] LLVM. *PartialInliningPass*. 2021. URL: https://llvm.org/doxygen/classllvm_1_1PassManagerBuilder.html.

[32] Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo, and Giovanni Agosta. "Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation". In: *21st Euromicro Conference on Digital System Design (DSD)*. Vol. 00. Prague, Czech Republic, Aug. 2018, pp. 172–176. ISBN: 978-1-5386-7377-5. DOI: 10.1109/DSD.2018.00042.

[33] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. "TAFFO: Tuning Assistant for Floating to Fixed point Optimization". In: *IEEE Embedded Systems Letters* 12.1 (2019), pp. 5–8. ISSN: 1943-0663. DOI: 10.1109/LES.2019.2913774.

[34] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Agosta Giovanni. "Dynamic Precision Autotuning with TAFFO". In: *ACM Transaction on Architecture and Code Optimization* 17.2 (May 2020). ISSN: 1544-3566. DOI: 10.1145/3388785.

[35] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[36] Daniele Cattaneo, Michele Chiari, Gabriele Magnani, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. "FixM: Code Generation of Fixed Point Mathematical Functions". In: *Sustainable Computing: Informatics and Systems* 29 (Mar. 2021). ISSN: 2210-5379. DOI: `https://doi.org/10.1016/j.suscom.2020.100478`. URL: `http://www.sciencedirect.com/science/article/pii/S2210537920302018`.

[37] D. Williamson. "Dynamically scaled fixed point arithmetic". In: *EEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*. 1991, pp. 315–318.

[38] The Open Group. *The POSIX.1-2017 Standard*. 2018. URL: `https://pubs.opengroup.org/onlinepubs/9699919799/`.

[39] Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. "Architecture-aware Precision Tuning with Multiple Number Representation Systems". In: *appear in Proc. DAC* (2021), p. 6.

[40] *BFloat16 processing for Neural Networks on Armv8-A*. URL: `https://community.arm.com/developer/ip-products/processors/b/ml-ip-blog/posts/bfloat16-processing-for-neural-networks-on-armv8_2d00_a`.

[41] Intel. *Instruction Set Extensions and Future Features*. URL: `https://software.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf`.

[42] *CostModel*. URL: `https://llvm.org/doxygen/CostModel_8cpp_source.html`.

[43] *or-tools*. URL: `https://developers.google.com/optimization`.

[44] A. Pullini. "Design of energy efficient microcon- trollers". PhD thesis. ETH Zurich, 2019.

[45] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. "AxBench: A Multiplatform Benchmark Suite for Approximate Computing". In: *IEEE Design Test* 34.2 (2017), pp. 60–68. DOI: `10.1109/MDAT.2016.2630270`.

[46] J. Walker. *Floating point benchmarks*. 2016. URL: `https://www.fourmilab.ch/fbench/`.

[47] *Field Oriented Control*. URL: `https://github.com/simplefoc/Arduino-FOC`.

[48] URL: https://www.sourceware.org/git/?p=newlib-cygwin.git.

[49] URL: https://docs.rs-online.com/7045/A700000006875645.pdf.

[50] Naohiro Iwanaga, Takayoshi Abe, and Akira Yamawaki. "Development of Fixed-point Trigonometric Function Library for High-level Synthesis". In: Sept. 2013, pp. 91–94. ISBN: 9784907220013. DOI: 10.12792/icisip2013.021.

[51] Wilfried Elmenreich, Maximilian Rosenblattl, and Andreas Wolf. *Fixed Point Library Based on ISO/IEC Standard DTR 18037 for Atmel AVR Microcontrollers.*

[52] S. Karris. *Introduction to Simulink with Engineering Applications.* Orchard Publications, 2008.

[53] Stefano Cherubin and Giovanni Agosta. "Tools for Reduced Precision Computation: A Survey". In: *ACM Comput. Surv.* 53.2 (Apr. 2020). ISSN: 0360-0300. DOI: 10.1145/3381039. URL: https://doi.org/10.1145/3381039.

[54] *mpmath.* URL: https://mpmath.org.

[55] *python.* URL: https://www.python.org.

[56] *Polybench.* URL: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/#description.

[57] *ubuntu 20.04.* URL: https://releases.ubuntu.com/20.04/.

[58] *Boost regex.* URL: https://www.boost.org/doc/libs/1_72_0/libs/regex/doc/html/index.html.

[59] *bfloat speedup.* URL: https://cloud.google.com/tpu/docs/bfloat16.

Appendix A

Polybench Benchmark Data

In this appendix we show the complete instruction count data for the PolyBench benchmarks.

symm

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	29	29	29	29	29	29	29	29
and Integer	5	5	5	5	5	5	5	5
bitcast Double Double	16	16	16	16	16	16	16	16
fadd Double	12	12	12	12	12	12	12	12
fmul Double	19	19	19	19	19	19	19	19
global Integer	1	1	1	1	1	1	1	1
lshr Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
or Integer	4	4	4	4	4	4	4	4
sdiv Integer	1	1	1	1	1	1	1	1
shl Integer	3	3	3	3	3	3	3	3
sub Integer	4	4	4	4	4	4	4	4
trunc Integer Integer	4	4	4	4	4	4	4	4
udiv Integer	3	3	3	3	3	3	3	3
uitofp Integer Double	3	3	3	3	3	3	3	3
urem Integer	4	4	4	4	4	4	4	4

symm benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

jacobi-1d

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	9	9	9	9	9	9	9	9
fadd Bfloat	0	2	0	2	2	2	2	0
fadd Double	8	8	8	8	8	8	8	8
fadd Float	2	0	2	0	0	0	0	2
fmul Bfloat	0	2	0	2	0	2	2	0
fmul Double	6	6	6	6	8	6	6	6
fmul Float	2	0	2	0	0	0	0	2
fpext Bfloat Float	0	0	1	0	0	0	0	0
fptosi Bfloat Integer	0	2	0	2	2	2	2	0
fptosi Float Integer	2	0	2	0	0	0	0	2
global Integer	1	1	1	1	1	1	1	1
mul Integer	1	1	1	1	1	1	1	1
sdiv Integer	3	3	3	3	3	3	3	3
sitofp Integer Bfloat	0	1	1	1	1	1	1	0
sitofp Integer Double	2	2	2	2	2	2	2	2
sitofp Integer Float	1	0	0	0	0	0	0	1
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	2	2	2	2	2	2	2	2
urem Integer	1	1	1	1	1	1	1	1

jacobi-1d benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

doitgen

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	17	17	17	17	17	17	17	17
bitcast Double Integer	1	1	1	1	1	1	1	1
fmul Double	6	6	6	6	6	6	6	6
fptosi Double Integer	3	3	3	3	3	3	3	3
global Double	120060	120060	120060	120060	120060	120060	120060	120060
global Integer	1	1	1	1	1	1	1	1
lshr Integer	1	1	1	1	1	1	1	1
mul Integer	6	6	6	6	6	6	6	6
sdiv Integer	1	1	1	1	1	1	1	1
sext Integer Integer	2	2	2	2	2	2	2	2
shl Integer	2	2	2	2	2	2	2	2
sitofp Integer Double	1	1	1	1	1	1	1	1
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	4	4	4	4	4	4	4	4
udiv Integer	2	2	2	2	2	2	2	2
uitofp Integer Double	2	2	2	2	2	2	2	2
urem Integer	3	3	3	3	3	3	3	3

doitgen benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

nussinov

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	24	21	21	21	21	21	21	24
and Integer	5	4	4	4	4	4	4	5
bitcast Integer Integer	4	4	4	4	4	4	4	4
fadd Bfloat	0	1	0	1	1	1	1	0
fadd Float	3	0	1	0	0	0	0	3
fmul Bfloat	0	5	4	5	0	5	5	0
fmul Double	0	3	3	3	3	3	3	0
fmul Float	14	5	6	5	10	5	5	14
fpext Bfloat Double	0	3	2	3	3	3	3	0
fpext Bfloat Float	0	2	4	2	2	2	2	0
fpext Float Double	1	0	0	0	0	0	0	1
fptosi Bfloat Integer	0	1	1	1	1	1	1	0
fptosi Double Integer	0	2	2	2	2	2	2	0
fptosi Float Integer	7	2	3	2	2	2	2	7
global Bfloat	0	250000	250000	250000	250000	250000	250000	0
global Float	250000	0	0	0	0	0	0	250000
global Integer	501	501	501	501	501	501	501	501
mul Integer	1	1	1	1	1	1	1	1
or Integer	1	1	1	1	1	1	1	1
sdiv Integer	1	1	1	1	1	1	1	1
sitofp Integer Bfloat	0	4	3	4	4	4	4	0
sitofp Integer Double	0	1	1	1	1	1	1	0
sitofp Integer Float	7	3	3	3	3	3	3	7
srem Integer	1	1	1	1	1	1	1	1
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	4	4	4	4	4	4	4	4

nussinov benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

	mvt							
	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	18	18	18	18	18	18	18	18
fmul Bfloat	0	5	5	5	0	5	5	0
fmul Double	21	16	16	16	21	16	16	21
fpext Bfloat Double	0	1	1	1	1	1	1	0
fptosi Bfloat Integer	0	2	2	2	2	2	2	0
fptosi Double Integer	8	6	6	6	6	6	6	8
global Bfloat	0	400	400	400	400	400	400	0
global Double	800	400	400	400	400	400	400	800
global Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
or Integer	2	2	2	2	2	2	2	2
sdiv Integer	1	1	1	1	1	1	1	1
shl Integer	5	5	5	5	5	5	5	5
sitofp Integer Bfloat	0	2	2	2	2	2	2	0
sitofp Integer Double	4	2	2	2	2	2	2	4
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	5	5	5	5	5	5	5	5
udiv Integer	5	5	5	5	5	5	5	5
uitofp Integer Bfloat	0	1	1	1	1	1	1	0
uitofp Integer Double	5	4	4	4	4	4	4	5
urem Integer	7	7	7	7	7	7	7	7
zext Integer Integer	4	4	4	4	4	4	4	4

mvt benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

2mm

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	23	23	21	23	21	23	23	23
fadd Bfloat	0	5	0	5	2	5	5	0
fadd Float	5	0	4	0	2	0	0	5
fmul Bfloat	0	13	9	13	0	13	13	0
fmul Float	13	0	2	0	11	0	0	13
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	6	0	5	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptrunc Float Bfloat	0	0	2	0	3	0	0	0
global Bfloat	0	193300	159100	193300	119200	193300	193300	0
global Float	193300	0	34200	0	74100	0	0	193300
mul Integer	5	5	5	5	5	5	5	5
or Integer	1	1	2	1	2	1	1	1
shl Integer	4	4	4	4	4	4	4	4
trunc Integer Integer	5	5	5	5	5	5	5	5
udiv Integer	4	4	4	4	4	4	4	4
uitofp Integer Bfloat	0	4	4	4	3	4	4	0
uitofp Integer Float	4	0	0	0	1	0	0	4
urem Integer	5	5	5	5	5	5	5	5

2mm benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

atax

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	15	15	15	15	15	15	15	15
fmul Double	10	10	10	10	10	10	10	15
fmul Float	5	5	5	5	5	5	5	0
fptosi Double Integer	4	4	4	4	4	4	4	7
fptosi Float Integer	3	3	3	3	3	3	3	0
global Double	410	410	410	410	410	410	410	800
global Float	390	390	390	390	390	390	390	0
global Integer	1	1	1	1	1	1	1	1
lshr Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
or Integer	2	2	2	2	2	2	2	2
sdiv Integer	1	1	1	1	1	1	1	1
sext Integer Integer	2	2	2	2	2	2	2	2
shl Integer	3	3	3	3	3	3	3	3
sitofp Integer Double	1	1	1	1	1	1	1	3
sitofp Integer Float	2	2	2	2	2	2	2	0
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	5	5	5	5	5	5	5	5
udiv Integer	3	3	3	3	3	3	3	3
uitofp Integer Double	3	3	3	3	3	3	3	3
urem Integer	2	2	2	2	2	2	2	2

atax benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

	gramschmidt							
	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	25	24	24	24	25	24	24	25
fadd Bfloat	0	2	0	2	2	2	2	0
fadd Float	2	0	2	0	0	0	0	2
fdiv Bfloat	0	2	2	0	2	2	2	0
fdiv Float	1	0	0	2	0	0	0	1
fmul Bfloat	0	9	9	9	0	9	9	0
fmul Double	1	1	1	1	1	1	1	1
fmul Float	9	0	0	0	9	0	0	9
fpext Bfloat Double	0	2	2	2	2	2	2	0
fpext Bfloat Float	0	0	0	0	2	0	0	0
fpext Float Double	2	0	0	0	0	0	0	2
fptosi Bfloat Integer	0	2	2	2	0	2	2	0
fptosi Float Integer	2	0	0	0	2	0	0	2
fptrunc Double Bfloat	0	1	1	1	1	1	1	0
fptrunc Double Float	1	0	0	0	0	0	0	1
fsub Bfloat	0	2	2	2	2	2	0	0
fsub Float	2	0	0	0	0	0	2	2
global Bfloat	0	153600	153600	153600	153600	153600	153600	0
global Float	153600	0	0	0	0	0	0	153600
global Integer	1	1	1	1	1	1	1	1
mul Integer	5	5	5	5	5	5	5	5
or Integer	12	4	4	4	10	4	4	12
sdiv Integer	2	2	2	2	2	2	2	2
shl Integer	1	1	1	1	1	1	1	1
sitofp Integer Bfloat	0	1	1	1	1	1	1	0
sitofp Integer Double	1	1	1	1	1	1	1	1
sitofp Integer Float	1	0	0	0	0	0	0	1
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	3	3	3	3	3	3	3	3
urem Integer	3	3	3	3	3	3	3	3

gramschmidt benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

heat-3d

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	24	24	24	24	24	24	24	24
fadd Bfloat	0	12	0	12	8	12	12	0
fadd Float	12	0	12	0	4	0	0	12
fmul Bfloat	0	11	5	11	0	11	11	0
fmul Float	11	0	6	0	11	0	0	11
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	14	0	9	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptoui Bfloat Integer	0	1	1	1	0	1	1	0
fptoui Float Integer	1	0	0	0	1	0	0	1
fptrunc Float Bfloat	0	0	2	0	4	0	0	0
fsub Bfloat	0	6	6	6	6	6	0	0
fsub Float	6	0	0	0	0	0	6	6
global Bfloat	0	128000	128000	128000	128000	128000	128000	0
global Float	128000	0	0	0	0	0	0	128000
global Integer	1	1	1	1	1	1	1	1
mul Integer	3	3	3	3	3	3	3	3
sdiv Integer	1	1	1	1	1	1	1	1
sitofp Integer Bfloat	0	1	1	1	1	1	1	0
sitofp Integer Float	1	0	0	0	0	0	0	1
sub Integer	2	2	2	2	2	2	2	2
trunc Integer Integer	2	2	2	2	2	2	2	2
udiv Integer	1	1	1	1	1	1	1	1
uitofp Integer Bfloat	0	1	1	1	1	1	1	0
uitofp Integer Float	1	0	0	0	0	0	0	1
urem Integer	1	1	1	1	1	1	1	1

heat-3d benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

	gemmver							
	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	11	11	11	11	11	11	11	11
bitcast Bfloat Bfloat	0	0	4	0	4	0	0	0
bitcast Float Float	20	0	12	0	4	0	0	20
fadd Bfloat	0	12	0	12	8	12	12	0
fadd Float	12	0	10	0	4	0	0	12
fmul Bfloat	0	19	14	19	0	19	19	0
fmul Float	19	0	3	0	19	0	0	19
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	9	0	7	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptrunc Float Bfloat	0	0	3	0	4	0	0	0
global Bfloat	0	163200	162400	163200	2400	163200	163200	0
global Float	163200	0	800	0	160800	0	0	163200
lshr Integer	4	4	4	4	4	4	4	4
mul Integer	1	1	1	1	1	1	1	1
or Integer	3	6	3	6	6	6	6	3
shl Integer	3	3	3	3	3	3	3	3
sitofp Integer Bfloat	0	6	5	6	4	6	6	0
sitofp Integer Float	6	0	1	0	2	0	0	6
trunc Integer Integer	4	4	4	4	4	4	4	4
udiv Integer	5	5	5	5	5	5	5	5
uitofp Integer Bfloat	0	1	1	1	0	1	1	0
uitofp Integer Float	1	0	0	0	1	0	0	1
urem Integer	2	2	2	2	2	2	2	2

gemmver benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

	3mm							
	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	29	29	29	29	29	29	29	29
fadd Bfloat	0	2	0	2	2	2	2	0
fadd Float	2	0	2	0	0	0	0	2
fmul Bfloat	0	10	10	10	0	10	10	0
fmul Double	12	12	12	12	12	12	12	12
fmul Float	10	0	0	0	10	0	0	10
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	0	0	2	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptosi Bfloat Integer	0	4	4	4	2	4	4	0
fptosi Double Integer	4	4	4	4	4	4	4	4
fptosi Float Integer	4	0	0	0	2	0	0	4
fptrunc Float Bfloat	0	0	0	0	2	0	0	0
global Bfloat	0	111900	111900	111900	77700	111900	111900	0
global Float	111900	0	0	0	34200	0	0	111900
mul Integer	5	5	5	5	5	5	5	5
or Integer	3	3	3	3	3	3	3	3
shl Integer	4	4	4	4	4	4	4	4
sitofp Integer Bfloat	0	4	4	4	2	4	4	0
sitofp Integer Float	4	0	0	0	2	0	0	4
trunc Integer Integer	5	5	5	5	5	5	5	5
udiv Integer	4	4	4	4	4	4	4	4
uitofp Integer Double	4	4	4	4	4	4	4	4
urem Integer	5	5	5	5	5	5	5	5

3mm benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

lu

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	33	26	26	26	26	26	26	33
and Integer	8	7	7	7	7	7	7	8
bitcast Bfloat Bfloat	0	4	4	4	4	4	4	0
bitcast Bfloat Integer	0	1	1	1	1	1	1	0
bitcast Double Double	4	4	4	4	4	4	4	4
bitcast Float Float	7	0	0	0	0	0	0	7
bitcast Float Integer	1	0	0	0	0	0	0	1
fdiv Bfloat	0	1	1	0	1	1	1	0
fdiv Float	1	0	0	1	0	0	0	1
fmul Bfloat	0	11	11	11	0	11	11	0
fmul Double	2	2	2	2	2	2	2	2
fmul Float	12	0	0	0	11	0	0	12
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	0	0	2	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptosi Bfloat Integer	0	1	1	1	0	1	1	0
fptosi Double Integer	1	1	1	1	1	1	1	1
fptosi Float Integer	1	0	0	0	1	0	0	1
fptrunc Double Bfloat	0	4	4	4	4	4	4	0
fptrunc Double Float	4	0	0	0	0	0	0	4
fsub Bfloat	0	6	6	6	6	6	0	0
fsub Float	6	0	0	0	0	0	6	6
global Bfloat	0	160000	160000	160000	160000	160000	160000	0
global Float	160000	0	0	0	0	0	0	160000
global Integer	1	1	1	1	1	1	1	1
lshr Integer	1	0	0	0	0	0	0	1
mul Integer	3	3	3	3	3	3	3	3
or Integer	4	4	4	4	4	4	4	4
sdiv Integer	5	4	4	4	4	4	4	5
shl Integer	5	4	4	4	4	4	4	5
sitofp Integer Bfloat	0	3	3	3	3	3	3	0
sitofp Integer Double	1	1	1	1	1	1	1	1
sitofp Integer Float	4	0	0	0	0	0	0	4
sub Integer	2	2	2	2	2	2	2	2
trunc Integer Integer	2	4	4	4	4	4	4	2
urem Integer	1	1	1	1	1	1	1	1

lu benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

trisolv

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	9	9	9	9	9	9	9	9
bitcast Double Integer	2	0	0	0	0	0	0	2
bitcast Integer Double	1	0	0	0	0	0	0	1
fmul Bfloat	0	7	7	7	0	7	7	0
fmul Double	8	3	3	3	3	3	3	8
fmul Float	2	0	0	0	7	0	0	2
fpext Bfloat Double	0	2	2	2	2	2	2	0
fpext Bfloat Float	0	0	0	0	1	0	0	0
fptosi Bfloat Integer	0	4	4	4	3	4	4	0
fptosi Double Integer	5	2	2	2	2	2	2	5
fptosi Float Integer	1	0	0	0	1	0	0	1
fptrunc Double Bfloat	0	1	1	1	1	1	1	0
global Bfloat	0	400	400	400	400	400	400	0
global Double	400	0	0	0	0	0	0	400
global Integer	1	1	1	1	1	1	1	1
lshr Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
sdiv Integer	3	3	3	3	3	3	3	3
sext Integer Integer	4	4	4	4	4	4	4	4
shl Integer	1	1	1	1	1	1	1	1
sitofp Integer Bfloat	0	4	4	4	4	4	4	0
sitofp Integer Double	4	1	1	1	1	1	1	4
sitofp Integer Float	1	0	0	0	0	0	0	1
sub Integer	3	3	3	3	3	3	3	3
trunc Integer Integer	5	5	5	5	5	5	5	5
urem Integer	1	1	1	1	1	1	1	1

trisolv benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

jacobi-2d

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	17	17	17	17	17	17	17	17
bitcast Double Double	20	20	20	20	20	20	20	20
fadd Bfloat	0	2	0	2	0	2	2	0
fadd Double	16	16	16	16	16	16	16	16
fadd Float	2	0	2	0	2	0	0	2
fmul Bfloat	0	4	2	4	0	4	4	0
fmul Double	6	6	6	6	6	6	6	6
fmul Float	4	0	2	0	4	0	0	4
fpext Bfloat Float	0	0	2	0	3	0	0	0
fptosi Bfloat Integer	0	2	0	2	0	2	2	0
fptosi Float Integer	2	0	2	0	2	0	0	2
global Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
or Integer	2	2	2	2	2	2	2	2
sdiv Integer	3	3	3	3	3	3	3	3
sitofp Integer Bfloat	0	3	3	3	3	3	3	0
sitofp Integer Double	2	2	2	2	2	2	2	2
sitofp Integer Float	3	0	0	0	0	0	0	3
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	4	4	4	4	4	4	4	4
urem Integer	1	1	1	1	1	1	1	1

jacobi-2d benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

					gesummv			
	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	9	9	9	9	9	9	9	9
fadd Bfloat	0	5	0	5	2	5	5	0
fadd Float	3	0	5	0	3	0	0	3
fmul Bfloat	0	9	8	9	0	9	9	0
fmul Float	7	0	1	0	9	0	0	7
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	3	0	1	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptrunc Float Bfloat	0	0	1	0	3	0	0	0
global Bfloat	0	125750	125500	125750	250	125750	125750	0
global Float	125750	0	250	0	125500	0	0	125750
mul Integer	1	1	1	1	1	1	1	1
or Integer	1	1	1	1	1	1	1	1
shl Integer	3	3	3	3	3	3	3	3
trunc Integer Integer	3	3	3	3	3	3	3	3
udiv Integer	3	3	3	3	3	3	3	3
uitofp Integer Bfloat	0	3	3	3	0	3	3	0
uitofp Integer Float	3	0	0	0	3	0	0	3
urem Integer	3	3	3	3	3	3	3	3

gesummv benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

floyd-warshall

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	14	14	14	14	14	14	14	14
and Integer	2	2	2	2	2	2	2	2
bitcast Double Double	4	4	4	4	4	4	4	4
bitcast Double Integer	2	2	2	2	2	2	2	2
fadd Double	4	4	4	4	4	4	4	4
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Float Double	1	0	0	0	0	0	0	1
global Integer	1	1	1	1	1	1	1	1
mul Integer	3	3	3	3	3	3	3	3
or Integer	7	7	7	7	7	7	7	7
sdiv Integer	1	1	1	1	1	1	1	1
sitofp Integer Bfloat	0	1	1	1	1	1	1	0
sitofp Integer Float	1	0	0	0	0	0	0	1
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	3	3	3	3	3	3	3	3
urem Integer	5	5	5	5	5	5	5	5

floyd-warshall benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

syrk

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	21	19	19	19	21	19	19	21
and Integer	4	4	4	4	5	4	4	4
bitcast Bfloat Bfloat	0	0	0	0	6	0	0	0
bitcast Float Float	14	0	0	0	0	0	0	14
fadd Bfloat	0	3	0	3	3	3	3	0
fadd Float	2	0	3	0	0	0	0	2
fmul Bfloat	0	11	11	11	0	11	11	0
fmul Float	12	0	0	0	10	0	0	12
fpext Bfloat Double	0	1	1	1	1	1	1	0
fpext Bfloat Float	0	0	0	0	4	0	0	0
fpext Float Double	1	0	0	0	0	0	0	1
fptrunc Float Bfloat	0	0	0	0	7	0	0	0
global Bfloat	0	105600	105600	105600	57600	105600	105600	0
global Float	105600	0	0	0	48000	0	0	105600
lshr Integer	1	0	0	0	1	0	0	1
mul Integer	3	3	3	3	3	3	3	3
or Integer	4	4	4	4	2	4	4	4
shl Integer	2	2	2	2	2	2	2	2
trunc Integer Integer	3	3	3	3	3	3	3	3
udiv Integer	2	2	2	2	2	2	2	2
uitofp Integer Bfloat	0	2	2	2	1	2	2	0
uitofp Integer Float	2	0	0	0	1	0	0	2
urem Integer	3	3	3	3	3	3	3	3

syrk benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

seidel-2d

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	11	11	11	11	11	11	11	11
fadd Bfloat	0	1	0	1	0	1	1	0
fadd Double	8	8	8	8	8	8	8	8
fadd Float	1	0	1	0	1	0	0	1
fdiv Double	1	1	1	1	1	1	1	1
fmul Bfloat	0	2	1	2	0	2	2	0
fmul Double	1	1	1	1	1	1	1	1
fmul Float	2	0	1	0	2	0	0	2
fpext Bfloat Float	0	0	1	0	2	0	0	0
fptosi Bfloat Integer	0	1	0	1	0	1	1	0
fptosi Float Integer	1	0	1	0	1	0	0	1
global Integer	1	1	1	1	1	1	1	1
mul Integer	2	2	2	2	2	2	2	2
sdiv Integer	2	2	2	2	2	2	2	2
sitofp Integer Bfloat	0	2	2	2	2	2	2	0
sitofp Integer Double	1	1	1	1	1	1	1	1
sitofp Integer Float	2	0	0	0	0	0	0	2
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	3	3	3	3	3	3	3	3
urem Integer	1	1	1	1	1	1	1	1

seidel-2d benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.

fdtd-2d

	fix	float_bfloat	float_bfloat_noAdd	float_bfloat_noDiv	float_bfloat_noMul	float_bfloat_noRem	float_bfloat_noSub	float_double
add Integer	26	24	29	24	23	24	23	26
bitcast Bfloat Bfloat	0	0	0	0	19	0	12	0
bitcast Float Float	27	0	0	0	0	0	0	27
fadd Bfloat	0	1	0	1	8	1	0	0
fadd Float	9	0	1	0	0	0	8	9
fmul Bfloat	0	14	14	14	0	14	9	0
fmul Float	29	0	0	0	28	0	17	29
fpext Bfloat Double	0	3	3	3	3	3	3	0
fpext Bfloat Float	0	0	0	0	23	0	67	0
fpext Float Double	3	0	0	0	0	0	0	3
fptosi Bfloat Integer	0	3	3	3	0	3	3	0
fptosi Float Integer	3	0	0	0	3	0	0	3
fptrunc Double Bfloat	0	1	1	1	1	1	1	0
fptrunc Double Float	1	0	0	0	0	0	0	1
fptrunc Float Bfloat	0	0	0	0	19	0	17	0
fsub Bfloat	0	11	11	11	46	11	0	0
fsub Float	49	0	0	0	0	0	42	49
global Bfloat	0	144000	144000	144000	144000	144000	144000	0
global Float	144000	0	0	0	0	0	0	144000
global Integer	1	1	1	1	1	1	1	1
mul Integer	4	4	4	4	4	4	4	4
or Integer	2	8	8	8	11	8	9	2
sdiv Integer	4	4	4	4	4	4	4	4
sitofp Integer Bfloat	0	7	7	7	7	7	7	0
sitofp Integer Double	0	0	5	0	0	0	0	0
sitofp Integer Float	7	0	0	0	0	0	0	7
sub Integer	1	1	1	1	1	1	1	1
trunc Integer Integer	5	7	12	7	7	7	7	5
urem Integer	3	3	3	3	3	3	3	3

fdtd-2d benchmark. This table reports the count of all the instructions created by TAFFO as described in section 7.2. The instructions are reported on the beginning of the rows and in the beginning of the columns are reported the different types of compilation. The matrix generated reports all the occurrence of a specific operation for a particular compilation mode.