Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO
MILANO 1863

Microarchitecture-aware Mixed Precision Tuning

Relatore: Prof. Giovanni AGOSTA
Correlatore: Dott. Daniele CATTANEO
Dott. Stefano CHERUBIN
Dott. Michele CHIARI

Tesi di laurea di:
Nicola FOSSATI Matr. 915244

Anno Accademico 2019–2020

Ringraziamenti

Vorrei ringraziare tutte le persone che mi hanno accompagnato durante il periodo di studi ed, in particolare, durante questo difficile semestre che ha segnato la fine del mio percorso. Ringrazio il prof. Giovanni Agosta, il Dott. Stefano Cherubin, il Dott. Daniele Cattaneo, il Dott. Michele Chiari e tutti i membri del team TAFFO: le loro ineguagliabili competenze e la passione dimostrata verso questo complesso argomento di ricerca mi hanno permesso di completare la stesura della tesi.

Ringrazio, poi, tutti i membri dell'HEAPLab che, oltre a mettere a disposizione l'hardware e i consigli necessari per il progetto di tesi, hanno allietato le poche giornate di lavoro prima del lockdown con una sana dose di risate e (forse troppa) caffeina, con il rammarico di non aver potuto trascorrere maggior tempo insieme.

Ringrazio la mia famiglia, che ha reso possibile questo cammino e che mi ha supportato, soprattutto moralmente, durante il percorso di studi e questi ultimi mesi impegnativi.

Un enorme grazie a Simona, la persona che più di tutte è riuscita a capirmi e a darmi sempre, grazie alle sue parole, sostegno e motivazione per continuare, specialmente nei momenti di sconforto.

Ringrazio, inoltre, Francesco, Matteo e Giuseppe per le intense giornate di studio e le lunghe nottate passate a configurare qualche strana distribuzione Linux o a correggere simulazioni Matlab. Senza di loro questi cinque lunghi anni non sarebbero stati gli stessi.

Ringrazio, infine, tutti coloro che, in un modo o nell'altro, mi sono stati vicini e mi hanno aiutato a raggiungere questo importante traguardo. A loro è dedicata la mia tesi, momento più determinante della carriera universitaria di uno studente.

Infine, ringrazio Te, lettore, che, addentrandoti in questa tesi, stai dimostrando interesse nel lavoro svolto, sperando tu possa trovare le risposte che stai cercando.

Abstract

The increasing demand to perform more complex computations on every kind of platform — ranging from low-cost processors found in internet of things devices, to large data centers — demands an increased effort in the optimization of each algorithm for the particular target architecture. The requirement of ever faster processing time and devices with longer battery life of portable devices calls for deeper knowledge of the applications domains in order to better optimize each algorithm.

Because of these factors, mixed precision tuning is increasingly being used in a vast number of application areas. This technique aims at lowering the execution time of an algorithm by changing the precision of some values. It exploits the great multitude of data types for number representation, each one with its own strength.

In this field, scientific research has focused its efforts on finding innovative approaches to automate the process, and as a result it developed several tools to do so, although they require very long processing times.

In this thesis, we propose an innovative approach to deal with the problem of precision tuning, based on the widely used theory of linear programming. It works by building a model of the program being optimized during the compilation. This approach tackles the problem in a static way, thus reducing the computation time.

The proposed solution has been implemented by extending a state of the art tool, TAFFO, which in turn relies on the LLVM toolchain. This tool, originally built to convert existing programs to work with fixed point math, has been extended to allow it to handle multiple output data types.

Our work has been evaluated by applying mixed precision tuning on the Polybench benchmark suite, to prove the effectiveness of the solution proposed.

Sommario

La crescente necessità di effettuare computazioni sempre più complesse su ogni tipo di piattaforma, dai processori a basso costo utilizzati in contesti di tecnologia pervasiva ai grandi centri di elaborazione, richiede uno sforzo sempre maggiore nell'ottimizzare ogni algoritmo per la specifica architettura sulla quale verrà eseguito. La richiesta di tempi di elaborazione sempre più brevi e l'obiettivo di rendere l'autonomia dei dispositivi sempre più lunga richiedono una conoscenza approfondita dei domini di applicazione al fine di ottimizzare gli algoritmi.

Per questo motivo, il mixed precision tuning viene sempre più spesso utilizzato negli ambiti più diversi. Questa tecnica permette di ridurre il tempo di esecuzione di alcuni algoritmi riducendo la precisione di alcuni valori, grazie all'utilizzo di svariati tipi di rappresentazione per i dati numerici, ognuno con i propri vantaggi, all'interno di un singolo programma.

La ricerca in questo settore sta concentrando gli sforzi per trovare modi innovativi di automatizzare l'operazione, proponendo diversi tool che adempiono al compito, a discapito dei lunghi tempi di elaborazione.

In questa tesi, viene proposta una nuova metodologia per affrontare il problema, basandosi sulle tecniche emergenti della programmazione lineare, costruendo un modello del programma durante la compilazione. Il metodo proposto permette quindi di risolvere il problema in modo statico, riducendo quindi i tempi di elaborazione.

La modellizzazione proposta è stata quindi implementata in un tool allo stato dell'arte, TAFFO, basato sulla toolchain LLVM. Questo tool, pensato inizialmente per convertire programmi esistenti al fine di utilizzare i fixed point, è stato esteso per permettere la conversione in diversi tipi di dato finali.

Il lavoro è stato quindi valutato effettuando il processo di precision tuning sulla suite di benchmark Polybench, per confermare la validità del modello.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A general purpose program usually spends the majority of the time while executing a very small amount of instructions. This section of the program is called computational kernel. Normally these computational kernels have very specific requirements for the input data, and ensure that the computation is carried out within specific error bounds.

Modern consumer hardware has become more and more powerful in the last years. Programmers working on these machines usually do not focus on the specific data type used for every computation, assigning to all the numerical variables the data type *double*, the most precise floating point data type present in the majority of the modern programming languages. This choice, of course, allows to program very accurate kernels. As drawback, long computation times may be required by the kernel to produce the results.

While this is not a problem for general purpose applications running on personal computers, it may become cause of concern in particular use cases. Such issues are present in scientific research, running on High Performance Computing infrastructures [11], on embedded systems, and in general in systems with reduced computational power.

In the former case, the computational kernel may be run several times: even a small reduction in the computation time may results in saving of thousands of Joule of energy, that can impact economically the specific research in progress [54] [55].

In the latter case, embedded systems usually need to continuously carry out the computation in a loop, while being subject to strict timing requirements. Moreover, on these platform, due to consumption requirement, several kind of hardware accelerators, such as FPUs or Vector Units, are not present, thus making double precision computation too slow and therefore infeasible. Additionally, longer computation times prevent the device to go into a low-power consumption mode, thus threaten-

ing the battery life [4]. This aspect is very important in safety critical devices and systems in remote locations.

However is not always possible to choose a different data type to carry out the whole computation. Even if some computation are resilient to the introduction of some noise, others are not. In fact, the precision of the result may be lowered too much and useless results may be produced.

From this trade off, the exploitation of the so called *Mixed Precision tuning* becomes more important, so that the program will have in different parts of the computational kernel the best data type to still carry out a meaningful result, while reducing the computation time. Yet, this requires a very accurate low level understanding of the algorithm being tuned.

To help programmers, researchers developed many tools to automate this approach [9]. Nearly all the proposed solutions are based on exhaustive search. The original program is modified changing the data type of some operation performed in high precision. Then the program is executed and instrumented. If a faster version of it has been generated while keeping the error under control, this new version is accepted as best solution and the search goes on, trying all the possible combination of assignments. Unfortunately, without any type of heuristic, the complexity of this approach grows in an unsustainable way with the number of instructions in the program. This may represent a problem if combined with relatively long compilation and execution times. In fact the whole process may last hours even for very simple kernels.

In this thesis, a new approach to *Mixed Precision tuning* exploiting Integer Linear programming is proposed, to solve the problems found.

Firstly, the program is analyzed and an integer linear problem is computed from it. The model represents all the possible data type assignment in the program, and also keeps track of the casting cost introduced if mixing more than one data type, which is in general non negligible.

Afterwards, the model is solved given some parameters, such as the weight to give to the computation time or to the precision. The solution is then processed, and a version of the program implementing the data type assignment proposed by the model is generated.

The solution has been implemented as an extension of TAFFO [12], a state of the art precision tuning framework, and in particular as a set of passes of the LLVM compilation toolchain. This makes it both source language and target architecture agnostic.

We finally evaluated the solution on the Polybench benchmark suite. The results confirmed the validity of the model, in particular on embedded architectures.

The structure of the thesis is organized as follows. In Chapter 2, some basic notions about number representations in computer memory are given, to better understand the following chapters. In Chapter 3, related work on mixed precision tuning is explored. In Chapter 4, a deeper look is given to compiler theory, focusing mainly on the LLVM framework, together with an introduction to the various element of the TAFFO framework architecture. In Chapter 5 the model construction, the foundation of the proposed work, is reviewed. Afterwards, in Chapter 6 we evaluated the proposed solution. Finally, in Chapter 7, some conclusion on the whole work are discussed.

Chapter 2

Basic notions

Before discussing about the Precision Tuning process, a brief introduction regarding the different way to represent real numbers in computers memory is given in this section. The different advantages and disadvantages of each data type will be discussed. Later on, various errors that may happen during the conversion between a real number and different types of finite representation are explained. Finally, some background about linear programming is given.

The interested reader can refer to Computer Arithmetic Algorithms [33] for a more extensive description of the various number representations.

2.1 Real number representations

The problem of representing infinite real numbers in computers is as old as the invention of such devices. In detail, numbers with possibly infinite ranges have to be represented in a machine with a finite amount of memory. More formally, a bijective function that maps every element of the set $\mathbb{R}$, which is infinite, to a set of bits, which is finite, must be found, and this is impossible. In fact, if a register with n bit is considered, the number of different states in which it can be are 2^n; if each state is associated with a different number, at most 2^n different numbers can be represented.

During the years, many different ways to represent numbers in computer memory were proposed. Each representation is a trade-off between the number of bits used, the range of the representation, and the complexity of the hardware that handles the operations on them (or the time complexity of the algorithm, if hardware support is not available and software emulation is used).

2.1.1 Integer representation

The integer representation is used to represent numbers without a fractional part. There can be integer types of any dimension in terms of bits, however programs usually exploit integer data types whose size is a multiple of the specific architecture's word size.

Besides the total amount of different numbers that can be represented, as discussed before, every data type has a range that specify the minimum and the maximum number that can be represented in it.

If n is the number of bits used, the range of an integer data type is usually $[0, 2^n - 1]$ for the unsigned version, or $[-2^{n-1}, 2^{n-1} - 1]$ for the signed counterpart.

2.1.2 Fixed point representation

A fixed point number is represented by a tuple $\langle sign, integer\_part, fractional\_part \rangle$, which represents the number $(-1)^{sign} \cdot integer\_part.fractional\_part$. In the same way as integer numbers, fixed point number representation can be signed or unsigned. In the latter case the *sign* is omitted in the tuple. The total dimension of the tuple i.e. the sum of the dimensions of each single element is usually, but not limited to, a multiple of the architecture word.

Said q the number of binary digits allocated to the fractional part, to get the decimal number contained in the fixed point number, the number have to be considered as if it was an integer number. Then the number obtained have to be divided by 2^q.

This representation does not require specific structures or instruction implemented in the processor to carry out operations, as it can be stored and processed using normal integer instructions. Therefore, this representation is usually used in ultra low-power devices, where a FPU cannot be implemented due to energy and/or silicon-space constraints.

As the splitting point between the integer and the fractional parts is not enforced by the hardware, a dynamic version of this data type can be also used. In this case, the position of the point is not fixed, but can change in different blocks of the program. The conversion between different fixed point data types is achieved by the scaling operation, which is a shift operation that takes into account the sign of the number. The mathematical operations between compatible numbers are achieved by using standard integer operations. Only multiplication and division requires a scaling operation to be performed afterwards to keep the same representation in the output.

As writing a program manually exploiting the fixed point data type is an error

prone and time consuming procedure, libraries to help the programmer have been proposed[45][8]. The features range from simple auto-scaling down to complete data type abstraction, so that the developer can completely ignore about the representation being used, and focus on more complex parts of the design.

2.1.3 Floating point representation

A floating point number is represented by a tuple $\langle sign, significand, exponent \rangle$; if we represent the significand as $d_0.d_1d_2...d_{p-1}$, the number represented will be:

$$(-1)^{sign} \cdot (d_0 + d_1\beta^{-1} + d_2\beta^{-2} + ... + d_{p-1}\beta^{-(p-1)}) \cdot \beta^{exponent}$$

where β is the base used in the representation, which is usually 2, and the *exponent* is a signed integer number.

Although it is possible to represent a larger range of numbers, there is a trade off between the *magnitude* of the number being represented and the maximum reachable *precision*.

It is important to note that a number does not have an unique representation. For example the number 110_b can be represented in different ways, such as but not limited to:

- $1.10_b \cdot 2^2$ (normalized form)
- $0.110_b \cdot 2^3$
- ...

In order to widen the range of the numbers that can be represented, the numbers are usually expressed in *normalized* form, so that d_0 is always equal or greater than 1. Using a binary base, since d_0 is always forced to 1, it can be omitted, actually gaining one more bit in the significand.

The floating point representation also has values for non numeric results such as *infinity* and *NaN*, representing respectively the mathematical infinity (e.g. when an overflow happens, or in case of operations like $n \div 0$ with $n \neq 0$) and non numeric results (e.g. $\infty - \infty$, $0 \div 0$, and so on).

The most widely used format is defined by the IEEE-754 standard [27] defines, and formalizes the behavior of operations (in particular in corner and special cases with respect to rounding) and exceptions. In particular it specifies two fundamental binary representation, `single` and `double`, using 32 and 64 bits respectively. The latest revision to this standard renames these two types to `binary32` and `binary64` while introducing the `binary128` format.

Name	Common name	Total bits	Significand bits	Exponent bits
binary16	Half precision	16	10	5
binary32	Single precision	32	23	8
binary64	Double precision	64	52	11
binary128	Quadruple precision	128	112	15
binary256	Octuple precision	256	236	19
bfloat16	Brain Floating Point	16	7	8
x86 80-bit	`long double`	80	63+1	15

Table 2.1: Most common floating point formats.

The latest revision also proposes an interchange format [28], named `binary16`. Despite being proposed as a storage format, GPU manufacturers started to embed native support for binary16 floating point operations into their hardware.

Other formats have been proposed, for example the 80-bit *x86 extended precision format*, also known as `double extended` format, supported by the *x86* processors family[29]. This format is very similar to the binary64 format, but has a longer mantissa. Being 80 bits long, it needs the presence of particular registers in the CPU. Therefore, under heavy register pressure, when the register is spilled into memory, the compiler down-casts it to a 64 bit representation, nullifying the effect of the gained precision in some situations[43]. This and the fact of its dimension not being a power of 2 led to the deprecation of this data type on modern x86_64 architectures.

Another data type that is being increasingly adopted is the *bfloat16* floating point data type, which is in practice a binary32 with truncated mantissa, so that the conversion between the two formats is achieved using a zero extension or a bit shift operation. Indeed, it allows fast typecasting. Despite the intended use was in deep-learning application, other applications are being investigated.

Table 2.1.3 shows a comparison between the various formats.

As numbers are logically split into parts with different meaning, integer operation cannot be used to perform floating point computations. Therefore, software routines or special hardware is needed. This kind of hardware is nowadays present in the major part of high end processors and is usually referred to as *FPUs*. Although having separate hardware can speed up floating point processing speed, it occupies spaces on the silicon, especially if large data types are supported, increasing the production costs and power requirements. Moreover, floating point operations have different latencies with respect to normal integer operations and can introduce very

long stalls in pipelined architectures [48]. For this reasons no floating point unit was implemented in one of the first MIPS releases [23].

Today some architectures are still lacking an FPU, in particular low-power or very cheap micro-controllers like some ATMega and STM32 ones [42] [57]. Usually, support for very large data types such as `binary128` and `binary256` is not present on CPUs, but only in special purpose hardware [37].

2.1.4 Arbitrary precision representations

Arbitrary precision representations, or *bignum* representations, aim to represent every possible number with virtually no precision or range constraint, being only limited by the memory of the host system. It is mainly used where the precision of the output is the main concern, while the computation time is not a constraint. Indeed, there is no hardware support for this type of representation, exception made for some deprecated architecture, like IBM 1620.

As algorithms to carry out operations with these kind of representation could become quite complex, programmers usually make use of externals libraries, made available by most programming languages. Notable ones are *mpmath* [30], for python and *GMP* [19], for C/C++.

2.1.5 Other types of representations

Logarithmic number system (LNS)

Logarithmic number system represents numbers by storing the sign and the logarithm of its absolute value in a given base b. For example to store the number -7_{10} with $b = 2$

$$log_2(|-7|) = 2.8073549220...{}_{10}$$

that converted into binary fixed point, used as representation for the fractional result of the logarithm operation, becomes:

$$10.11001110101011101...{}_2$$

and stored as an 7-bit fixed point, 4 fractional bits:

$$0010.1100$$

. The first bit is used to represent the sign of the original number and therefore it becomes:

$$1010.1100$$

The main advantage of this representation is that it simplifies the multiplication and division operations, by exploiting the well known logarithm properties

$$log_b(a \cdot c) = log_b(a) + log_b(c)$$

and

$$log_b(a/c) = log_b(a) - log_b(c)$$

As a drawback, addition and subtraction become complex operations As a workaround the implementation of lookup tables has been proposed. Indeed, practical usage have been limited to very short word widths. Still, the European microprocessor project has an implementation of this system, with a competitive solution for addition and subtraction [13], making this kind of computation faster and more accurate than floating point operations.

Residue number system (RNS)

The RNS stores numbers as a tuple of moduli with respect to a set of numbers relatively prime. For example, fixed the moduli set $\{3, 5, 7\}$, the number 10 can be represented as

$$10 \bmod 3 = 1$$
$$10 \bmod 5 = 0$$
$$10 \bmod 7 = 3$$

and therefore by the tuple $(1, 0, 3)$. The maximum amount of different numbers which can be represented is the product of the numbers in the moduli set, in the example $3 \cdot 5 \cdot 7 = 105$.

Additions and multiplications between numbers can be made one element at a time with the respective ones in the other tuple, wrapping the operation with a modulo.

Some studies proposed the use of RNS arithmetic in signal processing, but the lack of hardware support makes them an impractical solution [56].

2.2 Errors in numeric representations

As the number of bits used to store numeric values is finite and the values to be stored are usually infinite, some values are not stored exactly. In the next section the various problems that may arise are briefly outlined [9].

2.2.1 Arithmetic Overflow

Arithmetic overflows are errors in representation that happens when an instruction tries to save in a memory location a value that exceeds the largest number that can be stored in that location. In particular these errors happens the integer and fixed point representations, while other types may includes some ways to mitigate it — such as the floating point, which has a way to store the value "infinity".

The behavior in case of an arithmetic overflow is implementation-dependent. In case of wrapping overflow, only the least significant bytes of the value are stored, thus storing a smaller value, or in case of signed values, a negative value may be stored instead of a positive one. On the other hand, in case of saturating overflow, if such a condition happens, the maximum allowed values for that data type are indeed stored.

Most programming languages have some way to prevent this error. For example, the GCC toolchain supports various builtins to check overflow conditions for the C language, while others languages, such as Java, throw an exception at runtime if an overflow occurs, when using particular methods to perform the operation.

2.2.2 Representation Mismatches

Some data types may allow the programmer to store other values besides real numbers. IEEE-754 floating point numbers, for example, have representations for *infinite* and *Not a Number*. As there is no representation for such values in the fixed point notation, the conversion of these special cases is usually not handled, and therefore an error is generated. These types of errors are called Representation Mismatches.

2.2.3 Round off

Round of errors derive from representing a real number with a finite number of bits. Some bits will be truncated in the representation and therefore, if the bits are not zeros, a cancellation error occurs. Clearly, the round off error made depends strictly on the data type used.

Let's consider two real numbers, r_1 and r_2, and a final data type. The function $F(r)$ converts the real number r into the wanted representation system. If no r' exists such that $F(r_1) < F(r') < F(r_2)$ and $r_1 < r' < r_2$ then the distance between $F(r_1)$ and $F(r_2)$ represents the bound of the error that will be made when representing any real number in the interval $[r_1, r_2]$. In other words, if $F(r_1)$ and $F(r_2)$ are two adjacent numbers in the representation system, then another real number between r_1 and r_2 that can be represented without error does not exist.

A function $ulp(x)$ can be defined as the maximum difference between x and the closest numbers to x in the representation system. These numbers are called a and b, and selected such that $a \leq x \leq b$ and $a \neq b$. In particular, for floating point number, if $|x| \in [\beta^e, \beta^{e+1})$ then $ulp(x)$ can be computed as:

$$ulp(x) = \beta^{max(e,e_{min})-p+1}$$

Another constant usually defined for data types handling fractional numbers is *machine epsilon*, ϵ_M, that stores the smallest distance between 1 and its successor. While fixed point representation have the round off error equal to the machine epsilon, i.e. $\epsilon_r = \epsilon_M = \beta^{-p}$, floating point representations have a variable round off error, that depend both on the data type and the value x stored.

2.3 Linear Programming

Linear Programming [3] is a relatively young discipline of mathematics, born in 1947 with the key work by Dantzig, the *simplex algorithm* [15], still used today in many solver tools. Initially meant to solve U.S. Air Force military logistic problems, it was soon noted that with this method, a lot of classical problems, solved until that point with hit-or-miss approaches, or by human experience, could be solved automatically with the help of a computer.

From that point, the field saw an exponential growth.

The first machine implementation of such algorithm was done on the SEAC computer at the National Bureau of Standards [26]. This implementation could automatically solve problems up to 20 variables and 10 constraints. A new and more performing implementation was made in 1953 on a CPC, Card Programmable Calculator, which were able to solve a problem with 45 constraints and 70 variables in about 8 hours.

Successive implementations raised the number of constraints and variables due to improved calculators. In the mid and late 50s, the first implementation on a scientific computer was made for the IBM 701 and then for the IBM 704. During these years, the first commercial implementation was published and gained the attention of the oil industry. The first attempts to solve Mixed Integers programs were made by these implementations. In particular, LP/90/94 was the first commercial solver exploiting the branch-and-bound technique [14]. In these years was also proved that these technique could be used to solve real world problems.

During the 70s and 80s there was a great number of innovations, among which the exploitation of the dual simplex algorithm [38] to solve the problems. All the implementation developed at that time were written in assembly and therefore were pretty

specific to every architecture. MIP solvers introduced new procedures to explore the search tree. These optimization were made possible mainly due to innovations and progression in computer architectures.

During the first years of the 80s, the first IBM personal computer was introduced. In these years the first implementations of these algorithms in general purpose language were made, proposing the PC as an alternative to mainframes, even if being more than 100 times slower. Also notable, in 1979 Khachiyan proved that LP problems could be solved in linear time [32], although his approach was never used in actual programs.

Further studies brought to the formulation of a class of algorithms known as primal-dual log-barrier algorithms and their implementation in the FORTRAN language in 1991. In these years there were also a growth in commercial solver.

For what regards MIP problems, the work of Gomory (cutting-plane techniques) [21] on pure integer programs was notable. In general, MIP algorithms in use today remain pretty similar to the ones used in 70s. The gains in terms of speed come predominantly from advantages in computer processors and parallelization algorithm, and exploiting the power of multicore processors.

2.3.1 Problem formulation

The main focus of Linear Programming is the solution of Linear Programming Problems (LP Problems).

A LP Problem aims to maximize or minimize a linear function (called *objective function*) subject to a finite number of *linear constraints* [59]. More precisely, a LP problem is usually formulated as follows:

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

where a_{ij}, b_i and c_j are real constants.

2.3.2 Discrete and Integer Linear Programming

When at least one variable x_i is required to be integer, the problem becomes a Discrete Linear Programming problem. If all the variables are integer, then the problem is a Integer Linear programming (ILP problem).

Although its a formulation is very similar to a LP Problem, an ILP Problem cannot be solved using LP algorithms; moreover it was proved that solving ILP Problems is NP-complete [47].

In general, an ILP problem can be formulated as follow:

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, 2, ..., m) \\ & x_j \in \mathbb{Z}^+ \qquad\qquad\quad (j = 0, 1, ..., n) \end{array}$$

where a_{ij}, b_i and c_j are real constants and $\mathbb{Z}^+$ is the set of positive integers numbers.

Chapter 3

State of the Art

In this section, we will focus on the precision tuning process, explaining each step of it. As this methodology has already been covered extensively in literature, we will present some research works in this field and we will describe in details each work strengths and weakness.

3.1 Precision Tuning process

Precision tuning is a technique which aims to find the best data type (or the best *mix* of data types, if more types are used) to implement in a computational kernel in order to minimize a particular *cost function*, while observing some *constraint*.

A very common choice for the cost function is the computation time, or the energy consumption, while the constraint is usually an error function, which describes the mismatch between the output of the program using the precision mix used and the *exact* output to a well known input [11]. A threshold not to be exceed can also be defined.

Conversely, especially on embedded systems, the Worst Case Execution Time plays a fundamental role. This measure gives an estimation of how long the computation will take in the worst case scenario, so that the real execution time will never exceed the WCET. In real time systems, this parameter ensures that a computation will finish before the critical deadline, making the results useful and therefore correct. In this case, the best choice could be to use the error function as the cost function and define a maximum WCET to respect, in order to provide a precision mix that respects the computation deadline while using the best possible precision to carry out the computation.

The precision tuning problem can be split into a finite number of steps, which have already been covered in literature [9]. In the following sections every step is

briefly discussed.

3.1.1 Tuning Scope Investigation

Tuning Scope Investigation aim is to find where in the program it could be advantageous to apply the Precision Tuning technique. Not every part of a program can be optimized effectively; indeed only some algorithms are resilient to the noise that will be introduced by lowering the precision of some variables, and not necessarily the whole program. It is important to identify these regions, in order to keep the transformation as simple as possible (and therefore achieve better analysis time), while still covering the portion of interest of the program.

The original program is divided into chunks called regions. Each region can have its own sensitiveness to decrease of the precision level of the computation. There are different approaches to deal with the identification and analysis of such regions.

The trivial approach is to simply ignore that the program may present code regions with different characteristics. Hence, the whole program is analyzed, and an optimal data type allocation for the computation is produced [34]. Although this is a sound approach and provides correct results, the analysis of the entire program could lead to very long processing time. Indeed, the complexity of the computation of the precision mix grows exponentially with respect to the number of the variable in the program, if an exhaustive search is used. Moreover, if used together with static analysis, this approach usually leads to extremely pessimistic results, due to its nature of being very conservative. Thus, the final precision mix would preserve all the variables to the highest precision possible.

To overcome these difficulties, a similar is to use programmer hints to specify, which specific region of code should be analyzed and which should be not. Tools usually exploit either external *configuration files*, or *annotations* [12] inside the original source code. A notable variation of this approach is the so-called contract-based programming. The user specifies the desired precision for each specific function, given some preconditions on the input. These tools require a deeper understanding of the application field because the annotations, such as variable ranges and final accuracy, will be used to make decision about variable data types. Wrong ranges could lead to mix that do not respect the requirements and therefore not suitable for the usage.

On the other hand, the dynamic approach tries to infer information about regions of the original code by running an instrumented version of the program, with some specific inputs [25]. The regions of the program more stressed by floating point execution are then flagged for conversion, together with the collection of variables and intermediate results behavior. Unfortunately, this approach only use some of the possible input of the program and thus some corner cases may not be covered,

leading to unpredictable results. This approach is particularly useful whenever the programmer does not have in depth knowledge about the algorithm and its use, but can rely on example of algorithm data input example.

3.1.2 Requirement collection

The Requirement collection step tries to infer the sensibility of the program in order to understand how the different data types can affect the algorithm output.

There are various problems to take into account when changing one data type into another, starting from the representation mismatch, as discussed before. For example in some algorithms it may not be a problem infinite is not represented, while in others, a saturating data type must be present to carry out the correct computation. This problem can be avoided if the tools only consider compatible data types as destination types (for example the types specified by the IEEE floating point standard), but may become a problem while moving, for example, from it to fixed point data type.

Usually, in this phase, the tool tries to infer ranges for the variables present in the code region considered, and hence it selects set of safe data types to use in each specific position.

There are three different types of approach.

The first approach follows a *trial-and-error* approach: the data types are changed to less precise counterparts; then the code is run with a given set of inputs to check if the results are still acceptable [53] [35]. If not, the final data types are changed and the procedure is executed again. The tools that use this approach are distinguished mainly by the algorithm used to perform the search.

The second type of approach includes all the tools that perform a static analysis of the program [12] [16]. Such an analysis is necessary mainly to reduce the number of data types that will be investigated during the data type assignment. For example, data types that are too small to represent the full range of real numbers admissible for a variable are dismissed. In this approach, the tools infer information about variables range and errors propagation only by looking at the source code of the application, never dealing with instance of program input. Sometimes, it is required that the user enriches the original source code with annotations to characterize the input of a particular routine.

All tools that perform a dynamic analysis falls into the third approach: the code is run with a set of common inputs and then profiled to obtain the range of the different variables in the program [34]. Other tools aim to find information about how much the output is susceptible to variations of the input. In this way, the tool can understand which input variables can be lowered in precision without introducing

too much noise in the program and which must remain as precise as possible.

3.1.3 Code Manipulation

During this step, the actual conversion is made, in other words, an alternative version of each code region is generated, exploiting the data types selected at the precedent step.

In literature this problem has been tackled in a large variety of ways. The tools are classified mainly depending at which level they make the manipulation. In more depth, tools can be subdivided in four main areas.

Source-to-source compilers To this category belongs all tools which take as an input the source code of the program and produces as an output a program written (usually) in the same language [16] [34]. These tools are mainly used where it is important to have a readable output after the conversion, so that a programmer can still correct the code or tweak it if necessary.

Binary modification tools This category comprises all the tools that exploit an already existing compiled executable version of the program and patch it at binary level, to implement a different precision mix. They do not need the original source code of the application, and therefore are programming language agnostic [35]. Unfortunately, they are bound to a specific architecture, and additional support for different destination architectures may require non-trivial efforts, due to the differences between ISAs.

Compiler-level Transformation This category is composed by tools that work during the compilation of the program [**Precimonius**] [12] [31]. In particular they can be an extension to the compiler, or transform the intermediate representation of the compiler, that is produced during the compilation process. They work at an intermediate level between the two precedent approaches. Therefore, the output is still in the IR language. Even though it is usually a low level language, and therefore quite easy to manipulate automatically, it is easier for humans to read than binary machine code. This approach also allows to be completely independent from the source language, as long as the required source language has a front-end for the compiler.

3.1.4 Verification

The output of the precedent step is analyzed to understand the error introduced and accept or reject the it. The decision is made by looking at a certain threshold which

can be expressed either in terms of *ulp*, absolute numeric value or relative value. In case of rejection, the *Code Manipulation* step can be executed again, in order to vary the selected mix, and then the result can be evaluated again.

Just like before, the tools can be split up in two groups based on the type of verification that is done.

The first group is composed by tools that use a static approach [12] [16]. This method allows for computing error bounds without any input sample, through the exploitation of range arithmetic. Unfortunately, these methods alone usually overestimate the error, and therefore provide less useful information. However, if an error bound is found, it is *formally proved*, and therefore correct for every input. Some tools use formal methods to restrict the bound previously found, usually implemented as SMT solvers. Other tools use affine arithmetic to tighten the error bound propagation, without increasing the complexity of the computation too much.

The other group uses a dynamic approach instead [35] [53] [25]. This kind of verification cannot be used to formally prove the error bound (therefore it cannot be use for safety critical real-time systems) but it can give an idea of how much error is introduced. The modified program is run and the output is compared with the one produced by the original program.

3.1.5 Type Casting Overhead Estimation

Changing a data type in a specific region of a program requires the conversion of the data from the original data type to the requested one, both at the start of the converted region and at the end. Such conversions requires the insertion of casting operations, which add overhead to the final program. The overhead added can become non negligible in case a lot of data needs to be converted, such as arrays of elements or casting in a loop. Therefore it may happen that the performance improvements due to the conversion are counteracted by the overhead introduced by the casting.

An high number of different data types generate a large amount of castings, thus introducing a non negligible overhead. Therefore the *Analysis* step usually tries to choose the most uniform type mix. However, the estimation of this kind of overhead is still an open challenge, poorly addressed in the literature.

Some attempts to solve this problems tries to use dynamic performance profiling, but this can introduce high costs in terms of tuning time, as the code must be executed at every transformation.

Some others tools exploit heuristic/greedy algorithms to reduce the number of casts inside the new code as much as possible. This approach can effectively reduce the number of casts, but can also fall into a locally optimal solution, and never find

the best assignment to provide the best performance-precision trade off.

3.2 Previous work about mixed precision

As the mixed precision field of research is being explored by many years, many tools have been developed exploiting state of the art techniques. In the following section the main tools will be presented, with a focus on the particularities of each one.

3.2.1 CRAFT

CRAFT [35] is a framework for analyzing a previously compiled binary that uses double precision data types, and successively modify it to build mixed precision versions. Indeed, the original source code of the application is not needed. Moreover, it implements a search algorithm to find the best precision mix in the original program. Despite not being dependent on any particular programming language, it is indeed dependent on the target architecture.

The framework works as follows.

In the first place the binary is instrumented, by using the Intel Pin library [41]. Each function, basic block, and instruction is analyzed and a search tree is built, in order to understand which parts of the code will be able to exploit reduced precision computation. All the double precision operations in the program are candidates to be reduced in precision.

The Configuration Generator then builds, using a breadth-first algorithm, multiple mixed precision configurations for the same executable. The search is completely automatic and is provided in different flavors. The first is implemented as an heuristic search, which is quicker but may not provide the optimal solution. The second one is a complete search, which only reduces the search space by purging unfeasible precision mixes, such as mix with a lower precision than a previous mix that has already been proved to be unfeasible. Each configuration is stored as a series of human readable mappings; each instruction is mapped to one of the following types:

Single when the precision of the operation need to be lowered

Double when the precision of the operation needs to be performed in double precision and casts should be made if any parameter in input to this operation single precision

Ignored if the computation must be left in the original computation, for example in random number generation routines

The file can be edited by the programmer, to provide hints to the tool.

The Binary Modification tool, by exploiting the Dynist library, performs the real binary patching and produces many versions of the input program, as specified by the given configurations. In detail, the framework replaces all the instruction which are not flagged as ignored, with a custom routine, that performs the necessary casts and then carries out the computation. This is needed even for double precision operations, as a cast may be required for the input registers. When dealing with reduced size computations, 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 mixed precision binaries are then run against some test input data, and the results are evaluated by a user-specified routine. It is very important that the input provided by the user is very similar to a final use case, as the quality of the search will depend on this factor. When the search is completed, a final mix is proposed to the user, along with an executable which implements it.

This framework has been evaluated on many open source benchmarks and programs.

An extension to this work is the CRAFT framework. CRAFT instruments the program to understand how many bits of mantissa are necessary to still carry out a useful result. The approach followed for the analysis is similar as the one explained before. However, the tool is not proposed as a precision reduction tuner tool, but only as a tool to help the programmer understand how precise each operation should be, in order to guide the precision tuning process.

Both approaches are very useful to understand how the precision of an algorithm affects the output of a program, however it does not help the programmer to actually perform the tuning in the source code. Moreover, the quality of the results depends heavily on the input data and on the Verification Routine, and no guarantee on the soundness of the proposed solution is given for other inputs.

3.2.2 Precimonius

A similar approach to CRAFT is Precimonius [53], which is a framework build against the LLVM infrastructure, which aims is to find the best precision mix to exploit in a program, while maintaining a precision comparable with the original one. It is built from four components.

The first component duty is to analyze the program, represented as LLVM bitcode, in order to list all the variables that will be tuned, each associated with a set of

floating point data types that will be explored.

The second component deals with the actual search, implementing a variant of the delta-debugging algorithm, trying to find a type configuration which runs faster, but still meets the requirements specified. The metrics used to understand the speed-up are user-provided.

The third component generates the program variant that each step of the delta-debugging algorithm needs to analyze. Indeed, it modifies directly the bitcode representation, by changing the allocations made for variables and inserting additional operations when needed (floating point extensions and truncations).

Finally, the last part validates the program generated, by checking both the correctness and the speed-up achieved. It runs the program against a known input, comparing the results with respect to the original result. If the results are acceptable (i.e. under a certain threshold given by the programmer), the results of the instrumentation are handed over to the search algorithm, which goes on with the search.

This approach is very similar to CRAFT, however it is both source language and target machine independent, because it works on an intermediate representation (LLVM-IR). Unfortunately the framework still needs to run the to be tuned program and the effectiveness of the evaluation heavily depends on the quality of the input test data and on the error threshold given by the programmer. Moreover, delta-debugging has in average a complexity of $n \log n$ (n^2 in worst case) and cannot distinguish between a local minimum and a global one.

In a successive work, the same authors use an approach called Blame Analysis, implementing the shadow execution, to reduce the search space of Precimonius. Blame Analysis provides a blame set, which specifies whose variables can be lowered to a smaller data type without changing the final precision of the program. In this way these variable can be lowered immediately, reducing the final search space where Precimonius will act on. With shadow execution, multiple versions of the same program are executed in parallel, with different data types for each floating point instruction, thus requiring only one true execution. More precisely, the tool keeps for every variable a set of values with different types, with a trace of the instructions that generates it.

3.2.3 HiFPTuner

HiFPTuner [25] falls again in the dynamic analysis tool group. Instead of treating the program as a black box, it analyzes both the source code and the runtime behavior of the program, in order to find dependencies between floating point variables in the program. In this way, the algorithm reduces the search cost by reducing the search

space, for the sake of scalability.

The tool, first of all, exploits the dependence analysis and Edge profiling techniques to build a graph where the nodes are the variables, and each arc represents the dependency of a value from another (i.e. the value of a variable is used to compute the value of another). The weights on the arcs represents how many times that dependence is found. In particular, the various weights are extracted at runtime, by a dynamic analysis.

After this step, the tools tries to identify viable variable groupings by applying a community detection problem algorithm, as used in networks. The sets will contain variables that interacts frequently and therefore are likely to have the same requirements in terms of precision. The graph is then collapsed by assembling the variables in the same set into a single node, and the community detection algorithm is applied again. In this way an hierarchy can be established, represented by the various run of the algorithm.

Once the hierarchy is built, the tool performs the actual precision tuning. The algorithm starts from the top most level — the one where the variables are most condensed — and searches for the best configuration by treating all the variables in the same community as having the same data type. The results found are propagated to the next level downwards, in order to reduce the search space and avoid looking for more precise alternatives, if a lower precision configuration is found to be effective.

When a configuration is set, it must be verified in order to check if it keeps the error in the required bound, and if the final program has a speedup.

The tool is heavily based on Precimonius, as it shares with it the program transformation LLVM pass. Doing so makes HiFPTuner independent both from the source language and the destination architecture. Still, the quality of the output depends on the test data, and the program o be optimized need to be executed many times for profiling purposes.

3.2.4 CAMPARY

Campary [24] is an auto tuning tool that aims to increment performances in application requiring high precision libraries by generating mixed precision version.

This process is composed by two phases.

The first phase determines a tuning plan, which is a collection of tuning iterations. In each one, a specified set of variables whose precision can be reduced.

The tuning strategy prefers to reduce the precision of operations that probably impacts less the result of the program. In their work, they defined different groups of operations that require special consideration.

First of all, loops with high number of iterations may propagate larger errors

when variables are implemented with lower precision. Therefore, precedence is given to instruction outside loops, then the loops are sorted in order of increasing number of iterations. The number of iterations can be inferred either from static analysis or by dynamically profiling the algorithm.

Accumulation patterns also play a role in error propagation. Tuning down short sequences of instructions will likely introduce less error with respect to longer sequences. Then, precedence is given to the first one.

Finally it is considered that different operations have different costs even when done with the same precision. Therefore, operations that are costly in higher precision are scheduled to be reduced in earlier iterations.

During the second phase, for each tuning iteration, the tool modifies the original program by converting instructions to the data types requested. Then the modified program is run, so that it can be verified for error bound constraint specified. When the program violates some precision constraints, the tuner stops the tuning plan. The optimization of the program starts from on a higher precision version of it, having all the floating point variables converted to CAMPARY 256-bit data types [31]. The implementation of the proposed auto tuning algorithm is done as passes of the LLVM compiler infrastructure.

Like the previous proposal, this approach does not provide guarantees about the quality of the precision mix found, and the tuning still requires multiple run of the original program. Moreover, the evaluation of the quality of the new mix depends on user provided data and is not guaranteed to be appropriate for different data.

3.2.5 Daisy

A different approach to mixed precision tuning is given by *Daisy* [16]. It 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 shall be written in a real-valued language, but it is not limited to it, and usually a single function at a time is processed. Preconditions and post conditions of the computation are specified using the keywords *require* and *ensuring*; the compiler, knowing the preconditions, will find a precision mix that will respect the condition on the outputs. The body of the function can be a combination of all the common operation, transcendental functions and local variables, but does not support loops. After the tuning process, Daisy will produce a Scala or C source code with all the required casts and declarations.

The data types currently supported by Daisy are fixed-point (16 or 32 bits) and IEEE754 single, double and quad precision. It can be easily extended to support other representations, possibly custom ones.

The tuning procedure is split up into several steps.

During the first step named *Rewriting*, the tool rewrites the expression into equivalent ones that has a small roundoff error. A genetic algorithm is used to generate the new expressions to evaluate; the error is computed as if all variables had a uniform precision. This procedure is based on the Xfp tool, which has been extended to perform more effective transformation at each iteration of the algorithm. The algorithm, moreover, optimize both for performance and for uniform precision. Therefore the resulting expression will never be slower in computation than the original one, it will evaluate with a lower error than the original one in a finite precision environment.

During the second phase, Daisy rewrites the expressions in a sort of static single assignment form.

In the third phase, the range analysis is carried out, in order to perform bounds computation in the tuning phase. The analysis can be done using interval arithmetic, affine arithmetic or a more expensive alternative, using a mix of interval arithmetic and an SMT solver.

During the fourth phase, the actual mixed precision tuning is performed. A variation of the delta-debugging algorithm implemented by Precimonius is used to reduce the search space. Unlike other tools presented previously, Daisy performs a static error analysis, without running the original program and without any input/output examples. Because of this, the error bounds computed by Daisy are proven to be valid, and will always be correct, disregarding the specific input values. The error computation is derived from a precedent work of the same author, Rosa [17].

Daisy not only focus on variable data type tuning, but also on intermediate results. The running time for the algorithm is also computed statically using a prediction model, based on static costs.

In the last fifth phase, the best mix found in the previous phases is materialized into a human readable programming language (C or Scala), together with all the necessary casts and declarations.

The tool provides guarantees on the output program in terms of errors and execution time. However, Daisy does not accept programs written in a general purpose programming language. In addition, it lacks supports for conditional execution (as example, the *if* statement) or loops.

Chapter 4

Compilation frameworks

A compiler is a program that takes another program, written in an human intelligible language, and performs a conversion into a language understandable by machines, called *machine code*.

The firs 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. In these language each statement was converted into a machine opcode.

With the growth in popularity of higher level languages, such as C and FORTRAN, which allowed to build more complex projects that were easy to maintain and understand by multiple programmers, compilers became even more important, in order to let programmers obtain a final program that was as quick as if it was written directly in assembly.

Nowadays, compilers are more complex and can perform optimizations of any kind, in order to exploit peculiarities of the target machines. For example, taking advantage of a larger number of registers available or exploiting the presence of particular instructions that can process more data in one shot (SIMD instructions). Other optimizations are not linked to a particular target architecture, but can be made by exploiting properties of the program, such as strength reduction in loops or constant propagation.

Modern compilers rarely are monolithic programs, but they usually are a set of tools that transform source code into highly optimized machine code. In the following section, the analysis of common structure of the modern compiler will is proposed, with particular focus on the LLVM toolchain [36], on which the later presented work is based. It follows an introduction to TAFFO, which is the specific toolchain this work is based on.

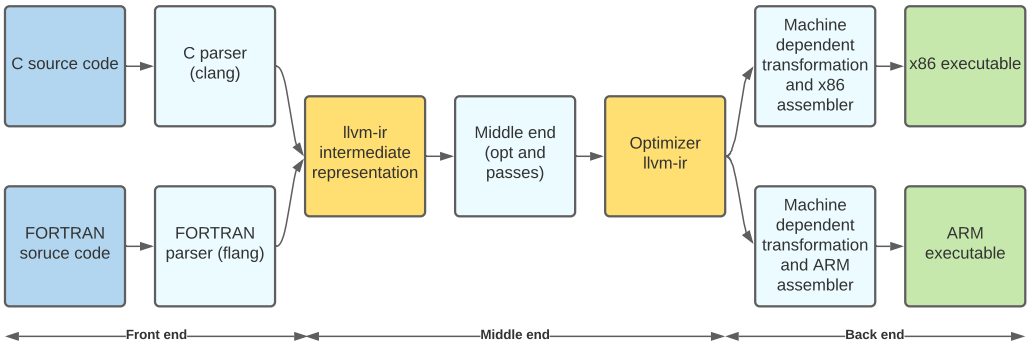

Figure 4.1: Internal structure of LLVM. Each pass is heavily decoupled from others.

4.1 Structure of a Compiler

Due to the fact that compilers are complex tools, most compiler designs are structured as a three-stage pipeline. This division is mainly required for abstracting the destination architecture from the source code language, in order to specialize each part of the compiler to better achieve its scope.

The LLVM Compiler Framework is an example of modern compiler following the best practices in the state of the art, and it has gained consideration in the last years both from academical research and the industry. Its main strength is its modularity which allows to share wide part of the compiler across multiple source language and architecture.

4.1.1 Front-end

The *front-end* is the component of the compiler which is in charge of analyzing the source code, given as input in a high level language, in order to lower it into a low level representation used internally by the compiler, called *intermediate representation* (IR).

The program is usually specified in a sufficiently expressive language, written using formal rules, such as Context Free grammars. The front-end performs several analysis on the input program.

The *lexical analysis* takes the input program, in form of text, and transforms it into a series of *tokens* that cannot be split anymore. All comments and white spaces of the input file are removed and only atoms are left, such as keywords, constants, identifiers and so on.

The tokens generated by the lexer are then analyzed by the *syntax analyzer* which compute which rule of the programming language grammar has been used to generate

a particular statement. At the end of this procedure, a parse tree is produced, containing all the rules used. All the leaves of the tree will contain terminal symbols, which are part of text that cannot be split further more using the grammar rules. If no rule can match a particular statement an error is produced.

Finally, the *semantic analyzer* checks that the parse tree is not only correct from a syntactic point of view, but that the meaning of the rules used is also sensical. A common error that could happen is when there is a type mismatch between a declaration of a variable and its successive use, such as assigning a string to an integer variable.

The front-end is independent from the target architecture. It depends only on the source language.

The IR is usually as low-level as the assembly language, however it is independent from the destination machine. Like in the assembly language, each statement describes a primitive operation, such as mathematical operations, function calls, memory accesses and so on. However the memory is still viewed as a set of logical variables, rather than a set of bytes referenced by an address. Most IR languages in use have an unlimited number of registers available and are in Single Static Assignment Form (SSA) [51]. In the Single Static Assignment form, each register is assigned exactly once in the program. This is done mainly to simplify further code transformations. The source code is transformed to the intermediate representation through a process called lowering, where high-level constructs are converted into simpler objects.

The process of lowering does not preclude the IR from carrying additional information about the original program, in the form of metadata, which may allow further optimizations in later stages of the compilation.

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".

LLVM-IR represents the program as a collection of global variables and functions. In turns, functions are composed by basic blocks, and basic blocks contain a list of instructions.

The number of instructions in the IR is low due to the fact that many opcodes are overloaded thanks to the type system used. The LLVM-IR type system is fully statically and strongly typed. Thanks to this choice, all type-checking operations can be offloaded to the frontend.

The most important data types are arbitrarily-sized integers, floating point types, pointers to such data types and vectors of them. The majority of operators are in three-address form, taking two operands and producing one results.

Being in SSA form, each value is declared and assigned exactly once.

In Listing 4.2 we present the resulting LLVM-IR from the compilation of the simple program shown in Listing 4.1. While complex instructions have been converted into simpler ones, it is still easy to identify and understand the semantics of the program. For example the *alloca* instruction is used to allocate space on the stack to store variables, while *load* and *store* instruction are used to access the memory.

```
#include <stdio.h>
int main(int argc, char* argv[]){
    double a, b, c=2.0, d, e;
    scanf("%lf %lf", &a, &b);
    d=(a+b)*c;
    e = d / 3.14159265358979323846264338;
    printf("%lf", e);
}
```

Listing 4.1: A simple C program using floating point computation.

```
define dso_local i32 @main() {
  %a = alloca double, align 8
  %b = alloca double, align 8
  %c = alloca double, align 8
  %d = alloca double, align 8
  %e = alloca double, align 8
  store double 2.000000e+00, double* %c, align 8
  %call = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([8 x
    i8], [8 x i8]* @.str, i32 0, i32 0), double* %a, double* %b)
  %0 = load double, double* %a, align 8
  %1 = load double, double* %b, align 8
  %add = fadd double %0, %1
  %2 = load double, double* %c, align 8
  %mul = fmul double %add, %2
  store double %mul, double* %d, align 8
  %3 = load double, double* %d, align 8
  %div = fdiv double %3, 0x400921FB54442D18
  store double %div, double* %e, align 8
  %4 = load double, double* %e, align 8
  %call1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4
    x i8], [4 x i8]* @.str.1, i32 0, i32 0), double %4)
  ret i32 0
}
```

Listing 4.2: The LLVM-IR generated from the program shown in Listing 4.1.

4.1.2 Middle-end

The middle-end is the part of the compiler that performs optimizations that are independent from the target architecture. It executes units called *passes* on the

IR of the source program. There are two types of passes in LLVM: *analysis passes* gain information about a program and have as a product of the computation these information; on the other hand *optimization passes* modify the code in order to change some property of the program, producing as output new IR code.

Optimization passes can be very expensive 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.

Examples of target machine independent optimizations are constant propagation and folding, which tries to simplify expressions dealing with constants values. Other optimizations deal with dead or unreachable code elimination, to reduce the size of the final program. A special group is composed by all transformations dealing with loops, such as unrolling and skewing.

Sometimes, there is a dependence between passes — in other words a pass needs to be performed before another one. For example a loop must be in canonical form in order to be optimized, thus the pass responsible for the transformation to canonical for must be performed in advance to the actual optimization. The pass manager is the component that is in charge of scheduling the passes correctly.

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 internal passes, LLVM supports plugins for adding other custom passes. These plugins are compiled as external objects that are loaded at runtime.

Listing 4.3 has been generated as an output of the optimization of the program shown in Listing 4.2. The effect of the optimizations is For example, the optimizer promoted some variables to registers. The variable *d*, for example, has a limited *liveness* inside the program, and has no meaning outside the *main* function. Therefore, it is not necessary to store it in a memory location, but can remain in a register. On the contrary, as the *a* variable is referenced from the outside (its address is passed to the *scanf* function), it must be preserved as a normal memory variable.

Another optimization that can be easily spotted is constant propagation. As the *c* variable is used to only store a constant, it is removed from the program and its use is replaced by an immediate containing the constant value.

```
1 @.str = private unnamed_addr constant [8 x i8] c"%lf %lf\00", align 1,
      !taffo.info !0
2 @.str.1 = private unnamed_addr constant [4 x i8] c"%lf\00", align 1, !
    taffo.info !0
3 define dso_local i32 @main(){
4 entry:
5   %a = alloca double, align 8
6   %b = alloca double, align 8
7   %call = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([8 x
```

```
    i8], [8 x i8]* @.str, i32 0, i32 0), double* %a, double* %b), !
    taffo.constinfo !6
  %tmp = load double, double* %a, align 8
  %tmp1 = load double, double* %b, align 8
  %add = fadd double %tmp, %tmp1
  %mul = fmul double %add, 2.000000e+00, !taffo.constinfo !7
  %div = fdiv double %mul, 0x400921FB54442D18, !taffo.constinfo !10
  %call1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4
    x i8], [4 x i8]* @.str.1, i32 0, i32 0), double %div), !taffo.
    constinfo !13
  ret i32 0
}
```

Listing 4.3: The LLVM-IR of the program shown in Listing 4.2 after optimization.

4.1.3 Back-end

The *back-end* of the compiler is the component that is invoked in order to produce the final machine code. It takes as an input the IR code, directly from the front-end or modified by the middle-end, and emits machine code.

The most important change the compiler must perform is to perform the transformation from the IR — which has an unlimited number of registers to a concrete architecture with finite registers. During this step the back-end must determine in which target register to store each value and, in case there are not enough registers, which values to store in memory. This phase is called *register allocation*, and it strongly depend on the *liveness interference analysis*. This analysis compute whether two variables will be live at the same time in a specific portion of code and thus both must be present in a register. If so, these variables *interfere* and can not be stored in the same register. The problem is similar to the graph coloring algorithm, which is intractable due to the exponential complexity. Compilers usually implements heuristic to speedup the execution of the algorithm.

During these steps machine dependent optimizations are also performed. For example, loops can be converted into SIMD instructions if the target architecture supports them, or a peephole optimization converts sets of instructions into more efficient ones.

As each architecture has its own particularities, a different back-end is needed not only for different architectures (ex. ARM vs x86), but also for different revisions of these architectures. This is necessary because some instructions may not be available on older implementations, or a particular target may lack — for example — the floating point unit. Moreover, 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*.

LLVM supports a wide number of target architectures, due to the fact that 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* [40].

As the acronym suggests, the LLVM-IR bytecode can also be executed as is. In facts, the LLVM toolchain also supports being used as *just-in-time* compiler.

In Listing 4.4 we show a snippet of the final result of the compilation for an Intel CPU of the example program. The language used is quite difficult to understand and uses instructions specific to the target architecture. For example, it is possible to note the use of some instruction linked to the *mmx* extension, a family of SIMD instructions. Moreover, each function call has been transformed to follow the calling convention of the destination architecture. While still not understandable by the machine as it is, this is the lowest-level human-readable programming language.

```
1  pushq %rbp
2  .cfi_def_cfa_offset 16
3  .cfi_offset %rbp, -16
4  movq  %rsp, %rbp
5  .cfi_def_cfa_register %rbp
6  subq  $48, %rsp
7  movabsq $.L.str, %rax
8  movl  %edi, -20(%rbp)          # 4-byte Spill
9  movq  %rax, %rdi
10 leaq  -8(%rbp), %rax
11 movq  %rsi, -32(%rbp)          # 8-byte Spill
12 movq  %rax, %rsi
13 leaq  -16(%rbp), %rdx
14 movb  $0, %al
15 callq scanf
16 movsd .LCPI0_0(%rip), %xmm0    # xmm0 = mem[0],zero
17 movsd .LCPI0_1(%rip), %xmm1    # xmm1 = mem[0],zero
18 movsd -8(%rbp), %xmm2          # xmm2 = mem[0],zero
19 addsd -16(%rbp), %xmm2
20 mulsd %xmm1, %xmm2
21 divsd %xmm0, %xmm2
22 movabsq $.L.str.1, %rdi
23 movaps  %xmm2, %xmm0
24 movl  %eax, -36(%rbp)          # 4-byte Spill
25 movb  $1, %al
26 callq printf
27 xorl  %ecx, %ecx
28 movl  %eax, -40(%rbp)          # 4-byte Spill
29 movl  %ecx, %eax
30 addq  $48, %rsp
```

Figure 4.2: The logo of the TAFFO project.

```
31 popq  %rbp
32 .cfi_def_cfa %rsp, 8
33 retq
```

Listing 4.4: x86 Assembly code generated from the program shown in 4.3.

4.2 TAFFO

The Tuning Assistant for Floating Point to Fixed point Optimization framework [12] [6], also known as TAFFO, is an optimization tool built on LLVM whose aim is to help developers to automatically change the program precision mix [10] in order to optimize the execution time while preserving its correctness [18] [4]. A subset of variables is selected to be transformed by the tool into fixed point. As this selection process is completely automated, it fells in the *auto-tuning* framework category.

The tool is based on the LLVM 8.0 compiler toolchain, and is implemented as a collection of multiple passes which are run in sequence. The majority of the passes are *analysis passes*, which means that they do not modify the code of the program in any way, but they only parse it to infer information useful to later passes. Thus, TAFFO operates within the middle end stage of the compiler.

Since TAFFO operates at the LLVM-IR code level, it is both source language agnostic and destination machine independent, thanks to the complete decoupling between the front end and the back end in the LLVM toolchain. Moreover each pass is completely decoupled from the successive one. Thus, different version of each pass employing different algorithm can be used without the need to modify other parts of the tool.

The programmer is only required to share some knowledge with the tool about the variables that will be converted. In particular, the variables that the programmer wants to enable for conversion must be *annotated* with a special syntax that will let TAFFO know which ones can be modified and which ones must be left as they are.

TAFFO supports any kind of floating point variable: it is not limited to scalar values, but also supports arrays and structures. Annotations are expressed as a text string, and contains information about the range that a variable may assume at runtime, and an initial seed value for the error propagation analysis.

In Listing 4.5 an example of annotation is shown. The array `set` will be converted to fixed point. Each element of the array will contain, at the begin of the analysis, elements between -256 and 255. The quantization error assumed for the variable is considered to be 10^{-100}. These ranges are hints given to TAFFO, if necessary, TAFFO will enlarge them as needed. This behavior can be disabled by including the parameter *final*.

```
double __attribute__((annotate("scalar(range(-256, 255) error(1e-100))
    "))) set[100];
```

Listing 4.5: Annotation example

In Listing 4.6 an example of annotation of a structure is given. The annotation is simply a set of *scalar* annotations corresponding to each element of the structure, in the same order as they appear in it. If the structure contains some non floating point variable which should be ignored, the keyword *void* may be used as a placeholder. No value is given for the range as the TAFFO is able to compute it by analyzing the program.

```
typedef struct {
   float r;
   float g;
   float b;
   int i;
   float distance;
} RgbPixel;

RgbPixel pixels __attribute__(("struct[scalar(),scalar(),scalar(),void
    ,scalar()]"));
```

Listing 4.6: Annotation example for a structure

4.2.1 Pass overview

We will give a general overview of each pass of TAFFO, focusing on the internal working of each step.

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 user provided annotations and transforms

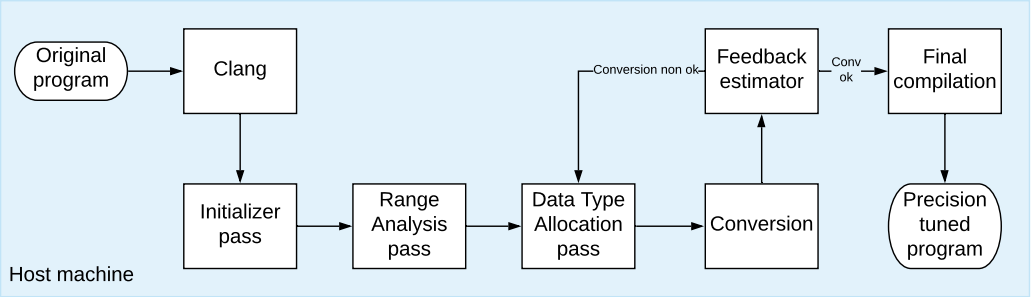

Figure 4.3: TAFFO architecture. All the various steps are decoupled one from another.

them into metadata, a data structure easier to handle in the LLVM framework.

This pass is also in charge of creating a different copy of each function for every different call to it. For example, if the same function is called across the program in three different places, after the execution of this pass there will be three identical copies of the same function. This is useful later on in the auto tuning process, as the arguments of each call may have different behavior in different program regions.

The metadata also provides information on whether a particular value has been selected for precision tuning or not, allowing to exclude critical section of the program to this process.

Value Range Analysis

The *Value Range Analysis pass* allows the successive passes to know the range of every value in the program. This is a very important and complex operation, because the more precise it is, the more accurate the selection of a data type will be. By exploiting range arithmetic and by inferring additional information from the source code, a final range for the variable is settled.

A multitude of different approaches can be used to acquire information on value ranges. The approach exploiting range arithmetic is the fastest one, but also the most inaccurate, as the predicted range can be very pessimistic and distant from the real program behavior. A new technique relying on symbolic execution is in development. This add analysis simulates the program behavior at compile time in order to acquire an increased amount of useful data. Unfortunately, this new approach may rise the compilation time.

Data Type Allocation

The goal of the *Data Type Allocation pass* (DTA) is to select an appropriate data type for each value in the program. The selection process is led primarily by the range of

each value. Indeed, a suitable fixed point representation to fit the whole computed range must be chosen. In current version of TAFFO, the algorithm is able to select only the position of the point.

The exploitation of fixed point data types can generate a very heterogeneous precision mix. Due to the fact that the conversion from a type to another has a cost, the more heterogeneous the mix, the slower the final program will be. The DTA step tries to limit excessive casts by merging similar fixed point types when used in the same computation. In order to preserve the original semantics, the data type selected as a replacement is the one with the smallest fractional part.

As multiple definitions of each function were generated during the initialization, the DTA pass will collapse all function with the same type assignment of the arguments, to reduce the final code size.

Conversion

The *Conversion pass* is the only pass that modifies the LLVM-IR code of the program. 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, such as call to an external library functions not available with fixed point parameters, a fallback code sequence is inserted, which converts the values back to their original types, using it and then converts the result back to a suitable fixed point type after the unsupported instruction.

Feedback Estimator

Finally, the *Feedback Estimator pass* analyzes the final code produced by *Conversion* and 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.

4.2.2 TAFFO strengths and limits

While being a mature and complete framework, TAFFO has some limits that, currently, have not been solved.

Single output data type

Even if the use of annotations allows a certain degree of selectivity, as opposed to other tools TAFFO does not allow converting kernels into different floating point types than the original. Thus, the only mix achievable will contain fixed point types and

the original data type. This can generate some issues during the transformation due to the fact that some portions of the code may be slower than the original ones, because of the usage of precise and expensive data types, even if not needed. In fact, even if fixed point are unsuitable for portion of the program, other floating point types may speedup the computation, while keeping the error into the required bounds.

No guarantees on execution time

The final generated program does not take into account the target architecture where the code will run, as all the annotated code is converted into fixed point. For some architectures, the transformation to fixed point may slow down the code with respect to the original one.

No guarantees on error

Although the final program is proven to be correct, as all the intermediate registers are of a fixed point data type with enough integer bits to contain all the possible ranges, no guarantees are given on the error in the computation results. Therefore, even if the transformation provides a reduced version of the program, the results may not be accurate enough for the specific application.

Non convertible code

Particular regions of code, such as calls to external mathematical function, must be handled carefully, as they cannot be transformed to take into account the use of fixed point types. As previously described, before such a call, the fixed point argument must be converted back to its original data type, and the return value of the function must be converted into fixed point once again if necessary. This can generate slowdowns because of unnecessary conversions.

Chapter 5

ILP for Mixed Precision tuning

In its current state of development, TAFFO only supports changing floating point types to fixed point. Furthermore, it is currently not capable of exploiting information on the target architecture during the Data Type Allocation process.

The objective of the proposed solution is to allow TAFFO to output an heterogeneous mix of data types, ranging from fixed points to standard floating points, and eventually more exotic data types. The selection of a data type for each original program variable and instruction should be carried out according to two fundamental parameters: the desired ratio between *precision* the *speed* of the program after the optimization. Usually, there is a trade-off between these two parameters: the more a program is precise, the slower it is expected to. Unfortunately, this intuitive statement is not always correct on modern architectures. Indeed, benchmarks show that double precision floating point operations are not significantly more time consuming than integer and single precision floating point operations. Therefore, particular attention is required, because a lower precision version of a program that is also slower than the original one should be avoided.

Ensuring that precision tuning poses a benefit is a complex task, mostly because the transformation procedure has to take into account the real instructions timings on the target hardware and the side effects of mixed precision, such as the casting overhead.

In this chapter, we present a technique to auto tune a generic program. In particular, we discuss a procedure to automatically build a integer linear programming (ILP) model. This model keeps into account both the execution time factor and the error propagation factor of the program. The generation of the model and its usage to perform an informed code conversion will be presented.

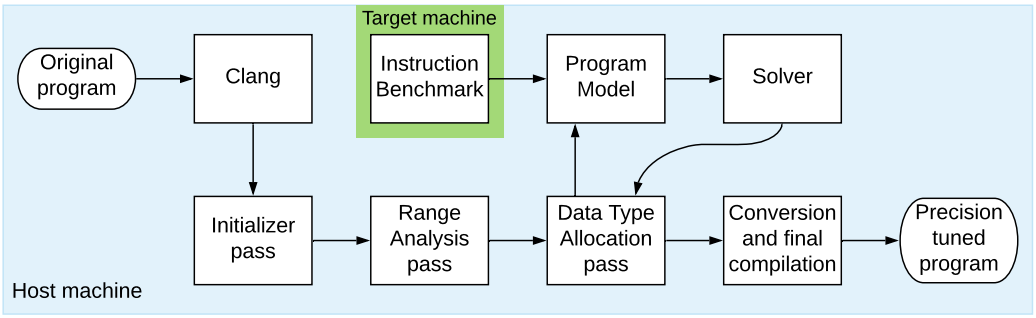

Figure 5.1: New TAFFO architecture. The steps with a green background must be done on the target machine. The steps with a blue background may be performed on a different machine via cross-compilation techniques.

5.1 Overview of the approach

The proposed solution heavily builds upon and extends the TAFFO framework. More precisely, the Data Type Allocation and Conversion passes have been greatly modified to support mixed precision with multiple data types.

In particular, the proposed implementation is composed of the following parts.

5.1.1 Instruction microbenchmarks

In principle, some metrics about the *target machine* are needed, in particular the relative execution time of each basic mathematical instruction or conversion between available data types. As information on the specific internal working of each architecture is not available, we chose to obtain this type of information via a dedicated ahead-of-time profiling stage. With the help of a benchmark, the selected target is profiled while executing a set of standard tests.

The benchmark collects the execution time of every possible mathematical instruction for every data type and of every possible cast operation. Since the exact absolute execution time for every instruction may vary depending on the specific machine, they are all normalized with respect to the fastest one.

Once the required information for the target machine has been collected, all the remaining compilation steps can be performed on a different machine, the *host* machine, whenever cross compilation needs to be applied.

Only this step requires the availability of a specimen of the target architecture during the compilation. It is intended to be executed only once per each different target architecture, independently from the number of compilations that will be done. There is no need to run the final compiled program on the target to get information regarding the new optimized kernel version.

5.1.2 New DTA algorithm

The TAFFO framework is left unchanged until the Data Type Allocation pass. In this stage, a linear model of the program being compiled is built, using both information from the source code itself and architecture specific timing information gained from the previous step. More details on the process involved in the generation of the model will be given later on. The model exploits the *or-tools* [22] integer linear optimization library to solve the optimization problem. Once the solution algorithm has found the optimal variable assignment, the result is loaded back into the DTA pass which finally applies the data type recommended by the solver to each instruction and variable allocation.

5.1.3 Enhanced Conversion

The Conversion pass is then executed, converting each instruction and variable to the data type recommended by the model. This pass has been extended to support as destination conversion data type every floating point type supported by LLVM, besides fixed point types which were already supported.

5.2 Comparing heterogeneous data types: the IEBW

When dealing with floating point data types and other real number representations with a finite number of bits, it is challenging to find a parameter to compare them, mainly because different types are defined in radically different way.

5.2.1 The *ulp*

A convenient method to compare two representation is to use the unit in the last place, or *ulp* [20]. This parameter defines the largest error that can be committed when representing a specific real number with that kind of representation.

ulp in floating point data types

The absolute error when using floating point types can be at most of 1/2 *ulp*, for every number represented. This is due to the fact that the point can be moved, and therefore the *ulp* is not fixed, but depends on the number that is being represented. In particular, let us assume we have a floating point number with base β, which can be considered without loss of generality equal to 2 and p digits in the fractional part. Therefore, the represented number will be:

$$d_0.d_1d_2...d_{p-1} \cdot \beta^e$$

that is

$$(d_0 + d_1\beta^{-1} + d_2\beta^{-2} + ... + d_{p-1}\beta^{-(p-1)}) \cdot \beta^e \tag{5.1}$$

Looking at (5.1), the absolute error can at most be

$$\frac{\beta}{2}\beta^{-p} \cdot \beta^e \tag{5.2}$$

While for a fixed exponent e the absolute error is the same for all the number representable with that exponent (i.e. the absolute error is only function of the exponent), the relative error can vary. From (5.1) it can be noted that, with a specific e the number that can be represented ranges from β^e to $\beta \cdot \beta^e$, excluded. Therefore, considering (5.2) the relative error ranges between:

$$\left(\frac{\beta}{2}\beta^{-p} \cdot \beta^e\right) \cdot \frac{1}{\beta \cdot \beta^e} \leq rerr(e) \leq \left(\frac{\beta}{2}\beta^{-p} \cdot \beta^e\right) \cdot \frac{1}{\beta^e}$$

which simplifies to

$$\frac{1}{2}\beta^{-p} \leq rerr(e) \leq \frac{\beta}{2}\beta^{-p} \tag{5.3}$$

As denoted by (5.3), the relative error bound does not depend on the specific number represented, but only on the number of bits p used for the fractional part.

ulp in fixed point data types

For fixed point data types, the problem can be formulated in a similar way. Therefore, considering a fixed point number with n total bits and $p-1$ fractional bits, with β as base, the number represented will be:

$$d_{n-p-1}\beta^{n-p-1} + ... + d_1\beta^1 + d_0\beta^0 + d_{-1}\beta^{-1} + ... + d_{p-1}\beta^{-(p-1)} \tag{5.4}$$

The absolute error, similarly to what occurs for floating point numbers, is still

$$\frac{\beta}{2}\beta^{-p} \tag{5.5}$$

but in this case it cannot be related to an exponent. Therefore the relative error of a number – with the exception of the representation of the 0 – can be as high as 50% with respect to the smallest number that the fixed point can represent.

$$\frac{\beta}{2}\beta^{-p} \cdot \frac{1}{\beta^{-(p-1)}} = \frac{\beta}{2}\beta^{-p} \cdot \beta^{(p-1)} = 0.5 = 50\%$$

In other words, when representing a number as a fixed point format, the error committed can be as high as half the represented number.

5.2.2 The IEBW metric

As a general rule, the floating point representation is better at representing sets of numbers where the values are very heterogeneous (i.e. the range of possible values is large), while fixed point formats are better at representing tighter ranges.

However, when exploiting mixed precision we need a metric which allows us to understand how better or worse it is to represent a generic variable with a specific data type. In the following discussion we introduce such metric, which we call IEBW.

Let us consider a fixed point type with an unlimited number of bits ($n \to +\infty$) but a finite number of fractional digits $p-1$. This allows the data type to contain any number in $(-\infty; +\infty)$ with a fixed absolute error, as expressed by Equation (5.5). This representation takes the name of *unrestricted fixed-point*.

Now, consider a generic data type t, which we call IEBW of t representing x, defined as the minimum number of fractional bits an *unrestricted fixed-point* should have to represent the same number x with a relative error lesser than or equal to the relative error of t.

In other words, when representing x as an unrestricted fixed-point with a number of fractional bits equal to the IEBW computed, it will have a relative similar error to the original one.

Finally, the notation IEBW(*data type*, x) represents the IEBW computed for the specific *data type* which is representing the number x. This is handy for data types that do not have a fixed absolute error for every value that they can represent. As an example, in this set falls all the floating point types.

While being very similar to the absolute error, the proposed metric provides some benefits. In practice, this is a linear function with respect to the number of fractional bits in a fixed point number. This is important and will be useful when dealing with linear problem as it is usually impossible to express other errors — like absolute and relative error — using only linear operators.

The IEBW can also be negative. In this particular case, it means that less than 0 fractional bits are needed in the unrestricted fixed-type to achieve the same error when representing a number in the specified data type. This is common when the original absolute error was greater or equal to the unit.

IEBW for fixed point

The IEBW computed for any fixed point with $p-1$ number of fractional bits is trivially equal to $p - 1$.

It is important to note that the IEBW does not depend on the number being

represented, and therefore

$$\text{IEBW}(fixp, x) = \text{IEBW}(fixp) = (p - 1) \tag{5.6}$$

IEBW for Floating point

As already seen, the absolute error while representing a number in a floating point data type varies with the value itself. To compute the IEBW it is useful to look at the way a floating point value is converted back into a decimal.

More in detail, referring to (5.1), the fractional part is allocated to exactly $p - 1$ bits. Equating (5.2) (considering p and e fixed) with (5.5), which is a correct equation for fixed point numbers as well, and assuming the same base β will result in:

$$\frac{\beta}{2}\beta^{-p_{float}} \cdot \beta^{e_{float}} = \frac{\beta}{2}\beta^{-p_{fix}} \tag{5.7}$$

which can be simplified to

$$\beta^{-p_{float}} \cdot \beta^{e_{float}} = \beta^{-p_{fix}} \tag{5.8}$$

which finally gives

$$p_{fix} = p_{float} - e_{float} \tag{5.9}$$

Therefore, remembering that (5.5) was computed for a fixed point having $p - 1$ fractional bits, the resulting unrestricted fixed-point will have a total number of fractional bits equals to $(p - e - 1)$.

e still needs to be computed and depends on the particular x represented. When considering only normalized numbers, where d_0 equals to 1, e can be computed as

$$e_{float} = min(\lfloor log_b|x|\rfloor, e_{max}) \tag{5.10}$$

It is important to make a note for the IEEE-754 representation. For the purpose of these formulas, p_{float} should be set to the number of effective bits. In fact, in the IEEE-754 specification, the number of stored bits is one less than the number of effective bits, as the number is always normalized and d_0 is therefore always considered to be 1.

5.2.3 The problem of fair comparison

Dealing with mixed precision, it is important to make decisions regarding which data type is the best for storing a particular value. In particular, thanks to the definition of IEBW a comparison between fixed point numbers and real number representation

is possible.

The main problem is to understand how to compute the IEBW and in particular, which x to choose as a representative number for the computation.

Before proceeding, it is essential to understand how a real number is represented as a fixed point number. In principle, the total number of bits n is fixed, and in general coincides with the number of bits available in a machine word of the target architecture. Indeed, a typical CPU takes a comparable amount of time in computing integer operations between a full word integer and smaller integer data types, when ignoring cache pressure and SIMD instructions. Then, considering the maximum representable number, inferred from the range of number that will be contained in it, a sufficient number of bits are reserved for the integer part, and the remaining digits are left for the fractional part. An additional digit is allocated to the integer part if the register may contain negative numbers, to store the sign bit in accordance to the two's complement representation.

There are, in practice, two values for which the IEBW can be computed. The first one is the maximum absolute representable number, or N_{max}, and the other one is the minimum absolute representable number that can be stored in a particular variable, or N_{min}.

Formally, these two parameters can be computed, starting from a range $[a, b]$, as follows:

$$N_{min} = \begin{cases} 0, & \text{if } sign(a) \neq sign(b) \\ a, & \text{if } a \geq 0 \\ b, & \text{otherwise} \end{cases}$$

$$N_{max} = \max(|a|, |b|)$$

When comparing two data types, in particular fixed point and floating point types with the same number of bits, the maximum absolute representable number is the least pessimistic case, as long as fixed point numbers are concerned. Indeed, in this case, all the bits in the fixed point are eventually exploited, while in the floating point case some bits (belonging to the exponent) are "wasted". On the other hand, when representing N_{min}, a large number of digits of the fixed point representation are put to zero and only a small amount of bits carry information. Conversely, in the floating point all the bits are actually used to carry information.

This consideration is reflected as a trade off on where to compute the IEBW: in general purpose programs, the IEBW should always be computed on the worst case, or in the minimum absolute relative number, thus making this approach conservative

and sound. If other features about the input data can be inferred, such as the distribution of values and so on, a representative sample of data (the average value, for example) can be processed to compute a more tight IEBW. In general these features are not known during a static analysis by the compiler. Indeed they cannot be exploited.

5.3 A cost model for mixed precision tuning

During the conversion of a program into fixed point, the best fixed point format that suiting the whole range is assigned to each register and variable under conversion. Thus, different instructions operates on fixed point with different fractional bits, even if they depend on each other.

This may lead to a very heterogeneous mix of fixed point data types, which can generate a program slowdown during execution due to the large number of casts that are required. Indeed, fixed point numbers with different position of the point are and should be considered as different data types.

A basic solution to this problem, which has been implemented in TAFFO, is to use a greedy algorithm that tries to merge the types of two similar data types. If the difference between the number of decimal digits is below a predefined *Q factor*, the fixed point type of a value and the type of the result of its use are merged [12]. By doing this, a moderate speedup can be achieved with relation to the baseline conversion, while some registers may not be kept at the maximum precision allowable by their range, thus introducing some noise in the computation [5].

Moreover, the problem grows in complexity when more than a single data type is considered, as when allowing heterogeneous data types in the final program. The algorithm described before, in fact, is hardly applicable because it assumes all types are fixed point. In addition, the number of possible assignments grows exponentially with the number of registers and instructions to be converted in the program.

The problem becomes even more complex when the output error of the program must be considered. Lowering the precision of a variable in a section of a program may make the high precision of other parts useless.

We propose a solution to this problem by exploiting a Linear Integer Programming problem. The only assumption made is that the program is expressed as LLVM-IR code, the intermediate representation used by LLVM.

5.3.1 Minimizing the number of type casts

Each time a register is given as input to an operation with a different data type than its own, a cast is needed before the operation is performed. Due to the fact that a

casts are operations computed by the processor, it will slow down the computation. In this first section, a model which aims to handle the casts in a program will be described. As an extension, a version of the same problem is provided for mixed precision programs, where more than one data type for fractional number representation is used, with particular attention to floating point numbers, both in single and double precision.

Problem definition

Considering the following snippet of code:

```
1 %a = OP1(...)
2 %b = OP1(...)
3 ...
4 %res = OP2(%a, %b)
```

Listing 5.1: Simple declaration and use example

Registers `%a` and `%b` are used in the operation `OP2` in order to compute a result to store in register `%res`.

Some constraints may be imposed between the two registers, for example the operator can require that both operands are of the same type. Moreover, additional constraints may be imposed to the result, for example in a sum the result is of the same type of the operands. These additional information can be exploited in order to build a model of the program to be optimized to use a more efficient data type assignment.

The procedure starts by assigning a set of variables to each register, for example x_a is assigned to `%a`. This variable will contain the number of binary fractional digits to allocate to a specific register. By exploiting the information produced by the VRA, the possible values for each variable can be limited as follow:

$$x_a \leq a_a$$
$$x_a \geq 0$$

where a_i is a constant representing the maximum number of binary digits that can be allocated to the register while still preserving the correct range. The number of total bits for each variable is constant, and a_i should take into account the sign bit.

An extension to this approach can enable the use of different final data types. In particular, we add one variable for each data type: $x_{a-float}$, $x_{a-double}$ and x_{a-fix}, which signifies whether a particular intermediate will use single-precision floating point, double-precision floating point or fixed point. These will be binary variables, i.e. integer variables in $\{0, 1\}$ range, where 1 indicates that the associated data type will be used, and 0 the opposite.

Since the final data type to be assigned to a register should be unique, the solver must be forced to select only one of the possible data type available. To achieve this behavior, the following constraint is inserted in the model:

$$x_{a-float} + x_{a-double} + x_{a-fix} = 1$$

which means that exactly one data type should be selected, at any time.

When a floating point data type is selected, x_a, which contains the number of fractional bits, can still vary in an uncontrolled way. We introduce the following constraint to solve this problem:

$$x_a \leq M \cdot x_{a-fix}$$

If x is selected to be a fixed point ($x_{a-fix} = 1$) then x_a will be constrained; on the other hand, when $x_{a-fix} = 0$, x_a is forced to 0. M is a big enough positive number.

Casts minimization

Each use of a register, when building the model, is interleaved with a "virtual" cast after its declaration. For example, the code analyzed in Listing 5.1 becomes:

```
%a = OP1(...)
%b = OP1(...)
...
%a_res_op2 = cast(%a)
%res = OP2(%a_res_op2, %b)
```

Listing 5.2: Virtual cast operator example

The variable $x_{a-res-op2}$ (and other variables for each data type) will be associated to the register `%a_res_op2` as explained before. The constraint, in term of maximum number of bits, associated to this new variable are the same associated with the original variable. Cast operations have a non negligible cost only if the type of `%a` will be different from the type of `%a_res_op2`. Otherwise, no casting operations will be inserted during the conversion pass and there will be no cost paid in term of computation time.

To model such a behavior is necessary to introduce two more binary variables that check if two data types are equal or not. If not, the cost of the casting should be taken into account when computing the total cost. To model this, $C1_{a-to-a-res-op2}$ and $C2_{a-to-a-res-op2}$ are defined as binary variables $C \in \{0, 1\}$. These variables should behave as follow:

$$(x_a > x_{a-res-op2}) \rightarrow (C1_{a-to-a-res-op2} = 1)$$
$$(x_a < x_{a-res-op2}) \rightarrow (C2_{a-to-a-res-op2} = 1)$$

which, translated into linear constraint, becomes:

$$(x_a - x_{a-res-op2}) < M \cdot C1_{a-to-a-res-op2}$$
$$(x_{a-res-op2} - x_a) < M \cdot C2_{a-to-a-res-op2}$$

where M is a very big positive number.

Binary variable Ci will be the first component of the objective function, representing the casting cost:

$$min[... + k_{cost} \cdot I_{cost} \cdot (C1_{a-to-a-res-op2} + C2_{a-to-a-res-op2}) + ...]$$

where k_{cost} is the cost for the specific conversion, which can be obtained by benchmarking the target architecture. We call the sum of all these costs C_c. On the other hand, I_{cost} is a constant that should be proportional to the number of times the cast will be executed at runtime. Usually loop analysis or dynamic analysis tools can infer an approximation of this metric, in order to provide a better estimation of the real cost. For our initial implementation, this factor has not been taken into account.

Note that $(C1_{a-to-a-res-op2} + C2_{a-to-a-res-op2})$ will be 0 if no conversion is necessary.

The sum of all the casting costs will be called C_c.

If mixed precision mode is enabled, we also define two conversion cost variable for each pair of data types. The total amount of conversion variable will be the cardinality of the cartesian product of the set of possible variable types, excluding where the two types are equal, plus two variables for the fix-to-fix cast described before. Hence, the total number of new binary cost variables introduced are $O(n^2)$, where n is the number of data types.

Each of these variables will have the following constraint:

$$(x_{a-fix} + x_{b-float}) \leq Ci_{a-to-a-res-op2} + 1$$

thus, when both variables on the left are equal to 1 (and therefore there is a type mismatch, as we define these variable only for different data type pairs), the conversion cost Ci will be enabled. Ci will be kept to zero by the minimization, if not required.

These new Ci variables are intended to be inserted into the minimization formula as well.

Even though other modelizations of the problem with less constraints may be found, this formulation describes the problem in plain terms, taking into account the estimated cost for each casting. Moreover, adopting a simple formulation eases the maintainability of the model generator, if a new data type should ever be implemented.

Constraints on instruction arguments

Some operations have very specific requirements on their input and output values. Considering the same example as Listing 5.1, let's assume that $OP2$ is a sum. Therefore:

```
1 %a_res_op2 = cast(%a)
2 %res = sum(%a_res_op2, %b)
```

Listing 5.3: Sum example

The sum is an example of a special operation, since it requires the two operand to have the same number of fractional digits in order to carry out a correct computation, at least when using fixed point data types. Therefore the analysis should take this into account, by inserting an explicit constraint into the model:

$$x_{a-res-op2} = x_b$$

Moreover, the output format of the sum will be equal to the input, so the following constraint should be taken into account:

$$x_{res} = x_b$$
$$x_{a-res-op2} = x_{res}$$

It is important to remember that some operations, like the multiplication, always requires a cast; therefore the output type is considered "free", as the conversion cost is usually present. The modeling of this cost can then be avoided.

When using mixed precision mode, more constraint must be taken into account. Specifically, LLVM requires the input data type to be equal for every floating point operation, the operation itself produce a result of the same data type. For example if a double precision floating point data type is used as first operand, the other operand must be a double precision type and the same type will be found as output of the computation.

To enforce this behavior, these constraints must be inserted into the model:

$$x_{res-float} = x_{b-float}$$
$$x_{a-res-op2-float} = x_{res-float}$$

and so on.

Constraints for other operations are omitted, as they are similar.

An important additional note must be done for external library call. In fact, these functions are not usually part of the compilation unit, and the source code is not available at compile time. Thus, the data type of every floating point argument must be converted back to its original type. This behavior can be modeled by forcing the data type after a "virtual" cast to the required one, and inserting it in the model as a casting cost.

This behavior must be emulated because external functions cannot be instrumented or modified by TAFFO. A new technique has been developed to enable TAFFO to modify calls to mathematical functions in order to achieve a better final execution time [7]. It works by replacing the original function, which uses the floating point data type, with a custom fixed point version of the function. As this feature is still under development and it is not available for every function of the C math library, it has not been exploited and therefore the model does not take advantage from it.

5.3.2 Minimizing the introduced error

The Integer Linear Programming problem as described up to now takes into account only the execution time of casts, and does not perform any prediction about the precision of the final program. In fact the less digits are given to the decimal part, the less precise the calculation will be. The same applies to floating point representation: the smaller the data type is, the less precise it will become.

A general solution is to consider the propagation of significant figures in the program.

Penalizing the introduced error

We compute the error that each data type can introduce inside the model.

In particular, IEBW_{orig-x} should be computed for every register and variable. It represents the IEBW of the original data type assigned. This is the equivalent number of binary fractional digits assigned to the smallest absolute number that the register can possibly store at runtime.

Therefore, in the cost formula, a new component for every variable and register can be introduced:

$$min[... + C_i \cdot (\text{IEBW}_{orig-x} - x_n) + ...]$$

where x_n is the IEBW computed on the selected data type.

Usually this addend is negative if starting from a program using only double precision floating point. Moreover, this expression can be simplified, as constant can

be eliminated in the objective function.

Therefore, the above expression will become:

$$min[... + C_i \cdot (-x_n) + ...]$$

C_i is very similar to I_{cost} discussed before, however this time it is greater than 1 if and only if the operation is used an iterative way, in order to model the loss of precision caused by such operations. An example of use case where C_i might be used is the reduction of an array (summing every element of an array to get one value). With this approach the cumulative error has more impact than, for example, in other cases such as summing a constant to a value in an array and storing it back in the same place. In the first implementation of the model, this cost is left to 1.

We call the sum of all the components of this cost E_c.

Mixed precision and IEBW

When dealing with the floating point representation, the number of fractional bits available in the data type cannot be directly used as x_n. In fact, doing this will give a false indication of the precision achieved by the representation.

To compare heterogeneous data types the IEBW can be used, as proposed in section 5.2. As the concept of IEBW is valid both for fixed and floating point data type, the equation for floating point variables becomes:

$$-x_{n-float} \cdot (\text{IEBW}_{n-float})$$

While this approximation is correct from a static point of view, it is unaware of the precision lost during the program execution. For example, storing a low precision result in an high precision variable does not restore the lost precision during computation. Therefore, an expansion to this technique will be explained in a further section.

5.3.3 Optimizing execution time

When formulating the problem in this way is quite intuitive that the solver will select, in most cases, uniform floating point double precision data types, because of their nature of having more precision than their smaller counterpart, producing a useless result.

However, this result is undesirable, because the main aim of precision tuning is to change the data types in the program to smaller ones in order to speed up the computation or reduce power consumption, while maintaining a sufficient level of

precision to give useful results.

Therefore it becomes necessary to add a cost associated to every (mathematical) operation, which proportional to the execution time — or the energy consumption if applicable. Summing these costs in the minimization formula makes the model aware of this trade off and therefore produces a precision mix that takes care of both precision and execution speed.

The formal definition is indeed very similar to the one previously defined to model the cost of casting:

$$min[... + k_{cost-op-float} \cdot I_{cost} \cdot (x_{a-float}) + ...]$$

Such addend must be repeated for every possible different data type, i.e. for fixed point, for single precision floating point and for double precision floating point. I_{cost} has the same meaning as before, while $k_{cost-op-float}$ is the cost of the single instruction in the desired precision.

Each cost is enabled only if its type is selected as the operand type. This part of the cost is called Ex_c. An estimated cost for each operation can be inferred using a benchmark, as explained before.

5.3.4 Useful IEBW propagation inside code regions

While the discussed model is formally correct, there is still a problem to be solved. In fact, the model is not aware of precision propagation inside the program. Let us suppose we have registers a, b, and the operation *op*. To gain precision, a and b could be recasted to a higher precision type, and then processed through *op*. Unfortunately, this will not help to increase precision, as the operation of casting the variables to the original type does not cancel the error introduced by the previous cast.

In other word a variable does not gain precision by the use of a cast of itself, but can gain precision only if other operations on which it depends are changed to use an higher precision data type too. This phenomenon needs to be appropriately modeled.

In details, we introduce a variable in the model which keeps track of the *useful* IEBW assigned to each variable and/or register. This variable can be interpreted as the number of figures that are meaningful in a specific variable.

For example, let us consider a variable a, with range $[1, 10]$ stored as a fixed point with IEBW $iebw_a = 6$ that is used in the operation $sum(a, 1)$. Considering that the operation is a sum, performing it as a double precision floating point instruction does not increase the IEBW of the result. In fact, even considering the precision of the constant infinite (IEBW $= \infty$), converting a to a double precision floating point will

not restore the lost precision due to previous operations and castings.

For each program variable, beside the already declared model variable, we define a new integer unrestricted variable, x_{IEBW} which represents the *useful* IEBW— which is different from the IEBW computed before, which represented the maximum IEBW achievable by the data type. This variable will replace the various elements in the objective function linked to the IEBW and therefore will be maximized to let the program have the maximum IEBW (and therefore precision) possible.

This variable should be limited. In particular the useful IEBW cannot exceed the static IEBW, computed starting from the range, as it would be impossible to store the exceeding digits. To model this, a constraint for every data type is needed. In detail:

$$\begin{aligned} x_{a-iebw} &\leq x_{a-n} + M \cdot (1 - x_{a-fix}) \\ x_{a-iebw} &\leq \text{IEBW}_{float}(a) + M \cdot (1 - x_{a-float}) \\ x_{a-iebw} &\leq \text{IEBW}_{double}(a) + M \cdot (1 - x_{a-double}) \end{aligned}$$

If no specific data type is selected ($x_{a-datatype} = 0$) M is summed to the left-hand side, a large number, in order to disable the constraint. As only one type variable can be active at any time, at most one of the above constraints will be active simultaneously at the same time.

As each mathematical operation may introduce different modifications to the IEBW, in the following sections we describe how it is propagated for several relevant operations.

Addition and subtraction IEBW propagation in the sum is trivial. The result is not allowed to have greater useful IEBW than the operands. So, for example, if the IEBW of the variable a is IEBW_a and there is the following computation:

$$c = sum(a, b)$$

c cannot have a IEBW greater than the input variables, similarly to the absolute error propagation, i.e.

$$\begin{aligned} \text{IEBW}_c &\leq \text{IEBW}_a \\ \text{IEBW}_c &\leq \text{IEBW}_b \end{aligned}$$

This property can be exploited both with addition and subtraction in order to build a model that takes into account the precision of the computation correctly.

Product When computing the IEBW resulting from a product operation, the computation becomes more complex. During the multiplication $c = mul(a, b)$, the worst case scenario is when both factors are considered with the minimum possible num-

ber in the range. In this case, to represent the first figure in the result that may be erroneous, at least $\text{IEBW}_a + \text{IEBW}_b$ binary fractional digits are needed. Therefore, representing the result with more digits is, in general, not useful. Hence, we can compute the maximum useful IEBW of the computation to:

$$\text{IEBW}_c \leq \text{IEBW}_a + \text{IEBW}_b$$

If information about the range of the variables is known, the IEBW can be tightened to a more realistic value. Indeed, when representing the smallest value, some bits on the left of the LSB can be different from zero. In this way they can carry over the error to a different digit, therefore amplifying it. In order to compute the new IEBW the following equation can be used:

$$\text{IEBW}_c \leq \text{IEBW}_a + \text{IEBW}_b - \min(bit_a, bit_b) \tag{5.11}$$

where bit_a and bit_b contains the number of digits needed to represent the minimum allowed value using an unrestricted fixed point with such an IEBW, i.e. the position of the most significant non-zero bit. This number can be computed as

$$bit_a = int\_bit_a + \text{IEBW}_a$$

where int_bit_a is the number of integer digits needed to represent the minimum value in a (known at compile time) and can be less than zero if some digits after the point are left to zero while representing the number (e.g. less than zero).

As $\min(bit_a, bit_b)$ is not a linear operator and cannot appear as a constraint in a linear problem, Equation 5.11 can be translated into the following constraints:

$$\begin{gathered}\text{IEBW}_c \leq \text{IEBW}_a + \text{IEBW}_b - bit_a + M \cdot y_1 \\ \text{IEBW}_c \leq \text{IEBW}_a + \text{IEBW}_b - bit_b + M \cdot y_2 \\ y_1 + y_2 = 1\end{gathered}$$

where y_i is a binary variable, constrained in $y_i \in [0, 1]$. In this way, only one of the two constraint will be active at any time, and, because IEBW_c will be maximized, only the less restrictive one.

Division The division can be reduced to a multiplication, when done in the real number set. In fact, a/b can be expressed as $a \cdot (1/b)$. As the problem of IEBW propagation in multiplication has already been tackled, the following discussion will focus on the IEBW computation for $1/b$.

Assuming that the IEBW of b is already known, for the computation of $1/b$, 1

can be assumed as a fixed point number with unlimited fractional digits. Therefore, when computing the division with the highest number that can be contained in b, the unit number is "moved" to the right at least by as many place as the number of bits needed to contain the integer part of b. Then, from here, the last digit that can be known is on the right of this digit of an amount equal to the IEBW of b. Here the IEBW must be computed on the worst case scenario that is on the maximum number that can be contained in b.

Hence, the total IEBW of $1/b$ is

$$\text{IEBW}_{inverse} = int\_bits_b + int\_bits_b + \text{IEBW}_b$$

Phi nodes In a program, a phi node is usually faced when dealing with loops and conditional statements. A phi nodes signals that the contents of a variable may come from different places, and thus the variable may behave in different ways depending on the control flow trace followed at runtime.

For what concerns our analysis and model generation, the propagation of the variable with the greatest IEBW must be taken into account. This is, in fact, the worst case scenario, thus it is a conservative modeling choice.

Therefore the resulting IEBW in a phi node is:

$$\text{IEBW}_{phi} = \max(\text{IEBW}_a, \text{IEBW}_b)$$

A problem arises when, in the list of values, a constant appears. In this case the constant is not processed. In fact it is impossible to evaluate the eventual IEBW generated by a constant in a precise way, while considering the full range IEBW may give a too pessimistic prediction.

Moreover, as the program is scanned top to bottom, some values for the phi nodes may not be available at the time of visit, as they could be computed in a basic block that is yet to be visited. This generates a dependency loop between the phi node result and the values themselves. To overcome the dependency loop, the IEBW variables are preallocated when a phi node is encountered. When the actual value is then reached, the correct variable is assigned to the previous computation and the dependency loop is closed.

Load/Store node When loading and storing into memory locations, it is important to deduce, in some way, the IEBW of the last instruction that performed a write at that memory address. In fact, the precision of a value stored in a variable does not only depend on the variable itself, but also on the results of previous operations, which in general are unknown. Therefore, the model construction takes advantage

of the LLVM *MemorySSA* [44] analysis which can be used to get the set of operations that could have written an area of memory before a particular load. More in detail, the goal of *MemorySSA* is to define a set of *may-def* instruction, or instruction that can write the memory just before the current use. Looking at this set, the *MemoryPhi* instructions can be synthesized, which contain the complete set of memory writers. These nodes are equivalent to *Phi Nodes* and can be processed in the same way.

Non-linear operations When considering non linear operation, it become more complex to propagate the IEBW through the computation. Therefore we find an approximation to propagate the IEBW while still using linear operations.

In general, the behavior a function near a given point can be approximated by the following formula

$$f(x_0 + dx) \approx f(x_0) + f'(x_0)dx$$

From there it follows that:

$$\Delta f = f(x_0 + dx) - f(x_0) \approx f(x_0) + f'(x_0)dx - f(x_0) = f'(x_0)dx \tag{5.12}$$

This formula, that is a first order Taylor series, can be used to compute the minimum error bound produced by a function as follows. Given the incoming IEBW, we want to know the IEBW that allows to represent correctly the resulting number without committing more error than it is already present in the input.

Let us consider dx as equal to the equivalent error associated with the IEBW, or — in other words — the maximum uncertainty associated with that variable:

$$dx = 2^{-\text{IEBW}}/2$$

The error on the output can be then computed using the Equation 5.12, and therefore becomes:

$$\Delta f = f'(x_0)dx = f'(x_0) \cdot 2^{-\text{IEBW}}/2$$

Reinterpreting this results in terms of IEBW,

$$\text{IEBW}_{result} = -\log_2(\left|f'(x_0)\right| \cdot 2^{-\text{IEBW}}/2) = -\log_2 \left|f'(x_0)\right| - \log_2(2^{-\text{IEBW}}/2)$$

Hence

$$\text{IEBW}_{result} = -\log_2 \left|f'(x_0)\right| + (\text{IEBW} + 1) - 1$$

The IEBW variation introduced by the non linear function can be seen as

$$\Delta_{\text{IEBW}} = -\log_2 \left| f'(x_0) \right|$$

From the previous equation it is clear that to evaluate the IEBW we need to establish a value for x_0. To keep the analysis conservative, the worst-case IEBW should be picked, therefore x_0 should be a point where a variation on the input produces the smallest variation on the output:

$$x_0 = \max_{x \in [a,b]} \Delta_{\text{IEBW}} = \max_{x \in [a,b]} -\log_2 \left| f'(x) \right|$$

where $[a, b]$ is the range of possible inputs to the function.

It is important to note that if the function has a minimum (or maximum) in the range considered, Δ_{IEBW} will be infinite. This may happen in functions like sin and cos.

5.3.5 The objective function

As only a single target function may be the subject of optimization, all the costs must be put together, in order to create a single expression.

As all the costs have to be minimized, we can simply sum C_c, E_c and Ex_c. Despite being distinct errors, there is a trade off between them: the lower the selected data type's IEBW, the more the intermediate results will be precise, but the final program will have a lot of costs in terms of execution time. On the other hand, a very homogeneous mix will lead to reduced execution times, but the precision will be reduced in some operations.

Therefore we need three parameters that signal which minimization to prioritize. In particular we call these parameters W_1, W_2, W_3. The final formulation will be:

$$\min[W_1 \cdot C_c + W_3 \cdot Ex_c + W_2 \cdot E_c]$$

As the values of C_c, Ex_c, and E_c are in general uncorrelated and on different scale, to obtain comparable results with the same parameters when compiling different programs and for different architectures, the costs are normalized to always be between 0 and 1. A normalization factor is therefore introduced, which is equal to the maximum possible value for each of the three component. The value can be computed during the creation of the model. Therefore the complete formula becomes:

$$\min \left[W_1 \cdot \frac{C_c}{N_c} + W_3 \cdot \frac{Ex_c}{N_{Ex}} + W_2 \cdot \frac{E_c}{N_e} \right]$$

Chapter 6

Experimental evaluation

We tested the proposed analyses and transformations, and theirs implementations to understand their performance. The testing scenarios have been chosen as similar as the intended common use.

As the solution is based on TAFFO, each test has been edited to insert the required annotations. The correct toolchain for every target architecture was also setup,most importantly for embedded systems.

Finally, all tests have been run on the selected targets, in order to collect data about the runtime behavior of the tuned programs, comprising the execution time and the error that was introduced.

In this section we will present the results obtained such benchmarks, together with the steps we followed in order to setup the test environment.

6.1 Experimental setup

In this section we outline the environment in which the experiments were conducted, with respect to four different variables involving our proposal. First, we describe how the target hardware was profiled ahead-of-time in order to collect the architecture-specific data required by the model. Then, we enter into detail regarding the benchmarks which were chosen, and the software implementation of our solution. Finally, the hardware used for the tests is outlined.

6.1.1 Ahead-of-time profiling

Before running any test, each platform must be instrumented, to understand how much each conversion will impact the final tuned program. The specific benchmark used to profile each architecture is a customized version of `time_arit` [39]. It works by allocating a memory buffer and then executing the same operation in a loop,

Instruction	Stm32	Raspberry	Intel	AMD
add_fix	1.2	1.3	1.1	1.3
add_float	2.3	1.8	1.0	1.3
add_double	2.7	2.2	1.4	2.6
sub_fix	1.2	1.3	1.1	1.3
sub_float	2.3	1.8	1.0	1.3
sub_double	2.7	2.2	1.4	2.6
mul_fix	1.6	2.0	1.4	2.6
mul_float	2.6	3.3	1.8	4.4
mul_double	4.0	4.1	1.6	4.6
div_fix	5.3	3.5	4.0	15.1
div_float	5.6	4.1	2.0	6.2
div_double	18.3	5.7	2.2	6.6
rem_fix	1.4	2.2	1.6	9.5
rem_float	27.0	15.2	54.0	13.6
rem_double	152.3	92.2	387.1	74.3
cast_fix_float	7.6	5.3	3.1	7.4
cast_fix_double	20.9	6.8	3.4	8.4
cast_float_fix	4.3	4.5	2.9	5.4
cast_float_double	1.6	1	1.2	1.7
cast_double_fix	5.6	5.5	2.7	6.1
cast_double_float	1.8	5.9	1.2	1.6
cast_fix_fix	1	1.1	1	1

Table 6.1: Results of elementary operation benchmarks on the architectures used during the experiments. Each time may be composed by the sum of multiple measurements.

for each data type over the whole buffer, in order to minimize the impact of the caching system. The run is repeated multiple times, recording the CPU time used by each execution and the median value across all times is returned. The CPU time is measured using the *clock_gettime* POSIX API, called with the parameter `CLOCK_PROCESS_CPUTIME_ID`. The benchmark must run on the target machine in order to acquire the correct parameters; however this process must be performed only once regardless of the number of compilations.

In Table 6.1 we summarized the parameters obtained from the test architectures. Each table of the row contains the weight for every elementary mathematical instruction and possible cast between data types. Each weight has been normalized with respect to the quickest operation. The instructions `rem_float` and `rem_double` have very high costs because they involve a call to the mathematical library function, as these operations usually are not implemented as hardware instructions.

6.1.2 Benchmark setup

The benchmark used to verify the quality of the proposed TAFFO extension is the POLIBENCH [60] test suite version 4.2.1. This benchmark suite is a set of different programs written in C containing different kernels used in various scientific research application. For example, some of the algorithm implemented are used to manipulate multimedia resources (such as the *fdtd-2d* Fourier transform, the *deriche* filter and so on), others are used in linear algebra computations, found in different fields, (*3mm* for matrix multiplication, *gramschmidt* for matrix decomposition, etc.) and data mining fields (*covariance* and *correlation*).

The test are written in the simplest possible way in order to to enable the verification of experimental research compilers and source code analyses. It is also possible to tune the amount of memory to allocate for each single test, in order to be able to test multiple target, even the most memory constrained ones, such as embedded systems.

The kernels have been altered to include the correct annotation required by TAFFO.

The initial data type selected was the maximum precision floating point type supported by all the hardware targets, the *double* data type. This version is used as a baseline for evaluating the error in the output of the optimized versions.

6.1.3 Software setup

The host machine, that is, the machine that will compile the program for the target, requires the installation of the LLVM toolchain, in particular version 8.0.1, on which TAFFO is based. While the tests have been performed with a debug LLVM build, it is advised to use a *release* version, because of the compilation speed improvements it enables.

TAFFO should also be present on the host machine, and compiled against the correct LLVM version in use.

When compiling for different architectures than the host machine, it is required that the host system also has available the *sysroot* and/or a *toolchain* for the target. Due to the use of the Miosix [58] kernel as a platform to run the test on the embedded target, its toolchain was also downloaded on the host system.

Finally, the host system must also have a working *python3* [52] installation, including the library *ortools* [22]. During the tests we used *python* version 3.8.5 and *ortools* version 7.6.7691. This is the only external dependency added to TAFFO by the solution proposed. In detail, python and the mentioned library are necessary to solve the linear optimization problem generated by the new version of DTA. Python

Parameter	Type	Behavior
`mixedmode`	boolean	Whether to enable or not the new DTA algorithm. If false, the old algorithm is used instead.
`costmodelfilename`	string	The name of the model file for operation timing, generated by the profiling.
`mixedtuningiebw`	decimal number	The weight to give to the program precision when solving the model.
`mixedtuningtime`	decimal number	The weight to give to the program execution speed when solving the model.
`mixedtuningcastingtime`	decimal number	The weight to give to the new cast introduced by mixed precision when solving the model.
`mixeddoubleenabled`	boolean	Whether to allow or not the inclusion of double operations in the final model. If false, some operation may still be done in double precision (e.g. where required by external functions).

Table 6.2: List of all the possible parameters of the new DTA

can be installed via the distribution's package manager, while the library can be obtained through *pip* [49], the Python package installer.

Since a TAFFO compilation is composed of several steps which are performed in a particular order, an helper script has been written to automatically execute each LLVM pass in the correct order and with the correct parameters. The DTA step has been changed and more parameters, listed in Table 6.2, have been introduced to allow a fine tuning of the underlying solver. The parameters *mixedtuningtime* and *mixedtuningcastingtime* have the same meaning in the model, as they are correlated with the execution time of the final generated program. Thus, they should always be equal.

Name	Architecture	Processor	Clock speed	Memory	FPU
Stm32	ARMv7-M	CortexM3	120 MHz	2 MB	no
Raspberry	ARMv6	ARM11	800 MHz	512 MB	yes
AMD	x86_64	AMD Opteron	2.6 GHz	218 GB	yes
Intel	x86_64	Core2	2.6 GHz	4 GB	yes

Table 6.3: Hardware specifications for each platform used.

6.1.4 Hardware setup

The tests have been run on different architecture. The main characteristics of each platform are summarized in Table 6.3.

As each architecture needs a specific setup in order to be tested, they are discussed distinctly.

Stm32

The STM3220G-EVAL is an evaluation board equipped with a 120 MHz CortexM3 ARM processor. It also features 1 MB of Flash, 128 kB of internal RAM and additionally 2 MB of SRAM, together with a series of different peripherals, such as the Serial interface, an LCD and a MicroSD reader.

The board can be used bare metal or with the help of an operating system. All the tests were run by relying on the Miosix real time operating system, as it offers a complete hardware abstraction layer and allows to execute the benchmarks with minimal modifications. The benchmarks were cross compiled and linked from an host system using LLVM together with the GCC toolchain provided by Miosix. The board was programmed before every test with the help of the Olimex ARM-USB-OCD JTAG [46] debugger tool. All the output was collected using the on board serial port.

To collect the kernel running time, the program was instrumented using the special debug register `CYCCNT`. When enabled, this 32 bit register increments by one on each cycle of the processor clock, except when in debug mode [2]. When it overflows, it simply wraps back to zero; on the board used, this event happens approximately every 35 seconds, which is a sufficiently long time to run the kernel of the benchmark.

Raspberry

The Raspberry Pi Model B Rev 2 is a single board computer featuring a BCM2835 ARMv6 processor supporting single precision floating point hardware instructions, running at 800 MHz (overclocked). It also features 512 MB of RAM, a small part

of which is reserved to the graphical processor. Despite being often compared to a complete desktop system, from the performance point of view it is more similar to an embedded system. The system was running a minimal installation of Arch Linux for ARM [1] updated to the 5.4 kernel revision.

Due to its limited computational power, all tests were cross compiled from an host system, with LLVM and the official cross compiler toolchain offered by the Raspberry Foundation [50]. All the tests were then uploaded to the Raspberry and executed. The profiling was made using the POSIX *clock_gettime* API. To make the results reproducible, the load of the board was reduced to the minimum and the CPU governor was set to *performance*.

AMD

This server is a NUMA node with four AMD Opteron 8435 CPUs, based on the K10 microarchitecture, running at 2.6 GHz and with a total of 128 GB DDR2 RAM. It has been selected for the tests as its architecture is similar to those used in high-performance computing. The operating system is Linux Ubuntu server 18.04.

The compilation was made on the server, using the release version of LLVM 8 available in the package manager.

The profiling was made in the same way as the Raspberry. No other services or superfluous processes were running on the server during the benchmarks, to better estimate the kernel running time.

Intel

A personal computer featuring a Core 2 dual-core processor running at 2.6 GHz, with 4 GB of RAM. The operating system installed is a recent release of Arch Linux, running the 5.4 kernel version. It has been selected to evaluate the performance of TAFFO in a general personal computer architecture.

The compilation was made on the computer itself. The evaluation of the speedup was made using the same API used for AMD and Raspberry. The load of the PC was reduced to the minimum and the CPU governor was set to *performance* to prevent variable frequency adjustments from affecting test results.

6.1.5 Model parameters

Each kernel has been tested on the destination platform in different configuration settings. The parameters passed to the solver are summarized in Table 6.4. As the parameters regulating the importance given to the execution time and the cast time are always equal, only one has been reported.

Mode	IEBW	Time
Quick	1	1000
Imprecise	20	80
Medium	50	50
Precise	1000	1

Table 6.4: Model parameters chosen for each configuration.

6.1.6 Evaluation metrics

To evaluate the performance of the proposed analyses and transformations, each benchmark has been instrumented to collect both the execution time and the results of the processing. We built an helper script to collect these parameters. In details, we computed for each benchmark the following information.

Speedup

The speedup represents how fast the new version generated by the optimization is with respect to the original version. It is expressed in form of a percentage; positive values represent a version that is quicker than the old one while a negative value represents a slowdown. A value of zero means that the kernel time has not changed between the two versions.

Formally, the speedup can be expressed as

$$S_{\%} = \left(\frac{t_{orig}}{t_{\text{TAFFO}}} - 1 \right) \cdot 100$$

where t_{orig} and t_{TAFFO} represent respectively the execution time of the original kernel version and the optimized one, in seconds or others units of measure directly proportional to the execution time.

Output precision

As a way to quantify the precision of the error, the *mean percentage error* (MPE) has been used as the metric of output quality. The MPE is defined as

$$E_{\text{MPE}} = \frac{100\%}{n} \sum_{i}^{n} \left| \frac{o_{orig,i} - o_{\text{TAFFO},i}}{o_{orig,i}} \right|$$

where $o_{orig,i}$ and $o_{\text{TAFFO},i}$ are the output of the original program and the tuned version respectively, and n represents the total number of elements found in the output.

Metrics of the solution of the model

Besides the speed and the error, we also collected some metrics about the solution of the model. In particular, after every compilation, we collected the number of instructions converted into a specific output type. These values can give an estimation of the final *precision mix*. In addition, the compilation time has been profiled, both when using the old TAFFO algorithm and when using newly proposed solution.

6.2 Result analysis

In this section we present the results obtained processing the data generated from the benchmark.

6.2.1 Speedup

In Tables A.1, A.2, A.3, A.4 and in Figures 6.1, 6.2, 6.3, 6.4 we show the speedup achieved by the kernels on the various target platforms. The label *Fix* denotes the same kernel compiled with the precedent TAFFO algorithm, which is capable of using only fixed point types as the final conversion data type.

Referring to the charts in Figure 6.1, it is possible to note that, apart from a restricted number of outliers, the algorithm generates multiple versions of the same kernel with an increasing speedup in relation to the more weight given to the execution time by the model parameters. In particular, speedup of over 800% where achieved by using the proposed algorithm.

In the *Quick* mode, the new linear-programming-based model was able to find a type assignment that produced faster programs than the original type allocation algorithm did. We believe that this happens because the new algorithm has a global view of the program being optimized, whilst the old algorithm was a peephole optimization. In particular, these speedups are produced by a more homogeneous fixed point assignment, requiring a lesser amount of casts in the final program to perform the computation.

The near to zero speedup found for the majority of the *Precise* versions on all the target architectures is generally produced by the inability of the model to find better type assignments to the variables, while when keeping the precision at the maximum. Even if this was achievable, the costs introduced by the casts would be too high and they would not allow any speedup. Therefore the model leaves all the variables as they are.

On the Raspberry target, no large speedup were achievable with the exploitation of mixed precision. In facts, all the versions are as time consuming as the original

double version, or as the fixed point version. The principal cause of this phenomenon is that the platform only supports to only single precision floating point operations, while the double precision is not hardware assisted, and therefore mixing single and double precision operation does not cause a speedup, contrary to the result on the Stm32 platform. The slowdown that can be seen on some *Precise* kernel versions is originated from the fact that the conversion back end is not perfect and sometimes generates redundant code.

Different results are obtained, instead, on targets Intel and AMD. On the Intel platform, it is possible to note that in the majority of kernels where the original TAFFO algorithm did produce a slower version then the original one, the new proposed analysis instead did find some opportunity to speed up the execution time. Moreover, if it could not find any faster version, it produced a version with similar execution times.

The slowdown outliers are probably generated by the platform complexities not modelled in the solution. For example, it is not feasible to model the superscalar architecture found on the Core 2 processor using a simple linear model cost. In these cases, the original code is faster due to a different allocation of the resources in the processor, or by the exploitation of different SIMD-style instructions to increase the data throughput when using specific data types.

The same reasoning applies to the results obtained on the AMD architecture. While the model is overall able to find more performant alternatives where the original DTA algorithm was not effective, sometimes it produces slower code, and the execution time of the final kernels does not monotonically follow the execution time parameters.

6.2.2 Error

In Figures 6.5, 6.6, 6.7 and 6.8 and Tables B.1, B.2, B.3 and B.4 we present the error computed as previously described.

When recompiling the same kernel with a lower requirement in terms of precision, the precision of the optimized version goes indeed down. The last column (*Fix*) is shown as a reference to compare the results with the precedent algorithm. Sometimes the error in the *Quick* column may be worse than the result in the column *Fix*. This is due to the use of different fixed point formats in the final mix.

There are some peculiarities in the table. First of all, in some kernels the error stays equal to zero. The analysis of such kernels showed that, in the case of `heat-3d` the error is always zero because the computation converges to the same values even for reduced precision kernels, while for `nussinov` and `floyd-warshall` the output of the computation is not a numeric value but a classification. Despite the approximation

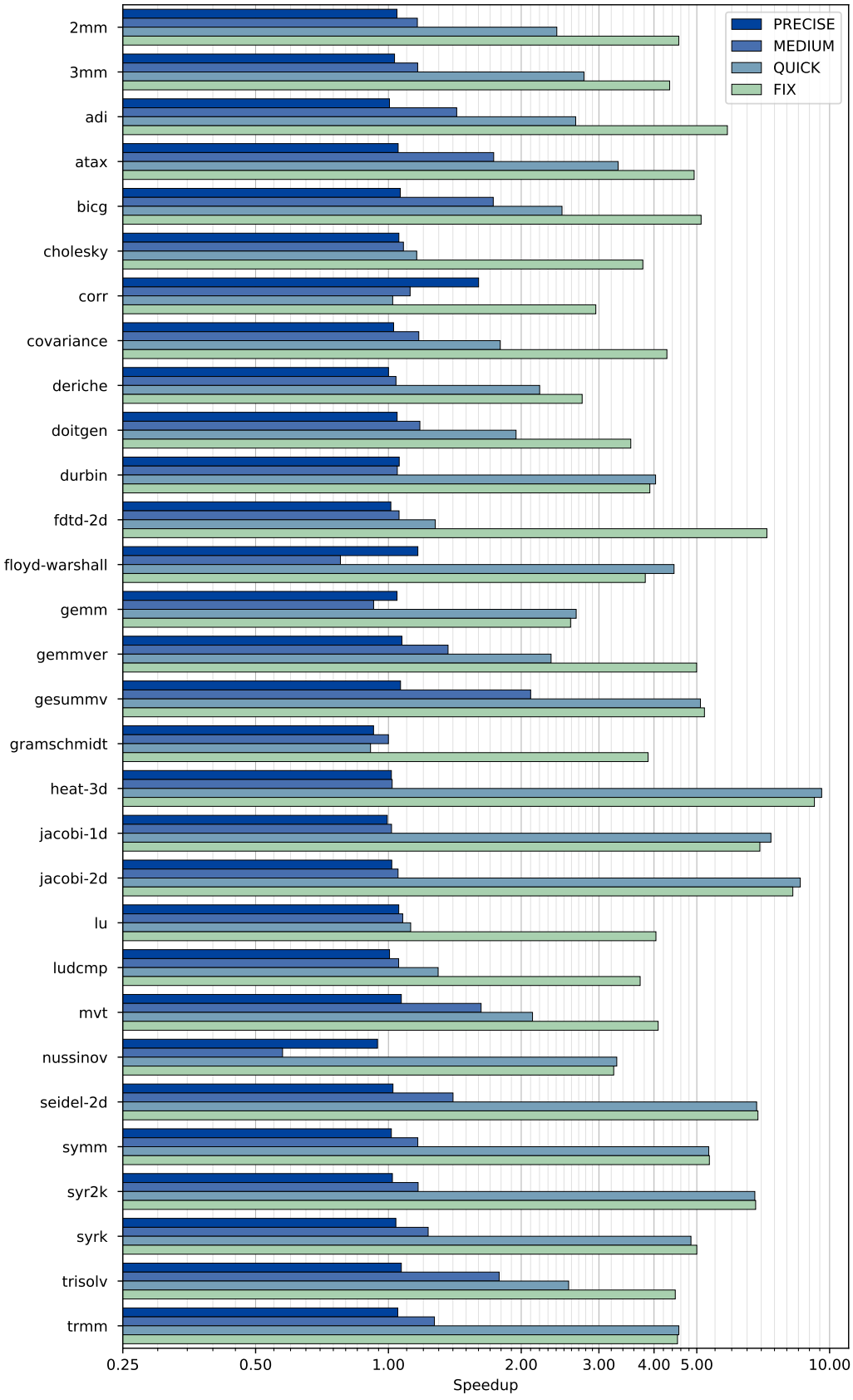

Figure 6.1: Stm32 speedup chart.

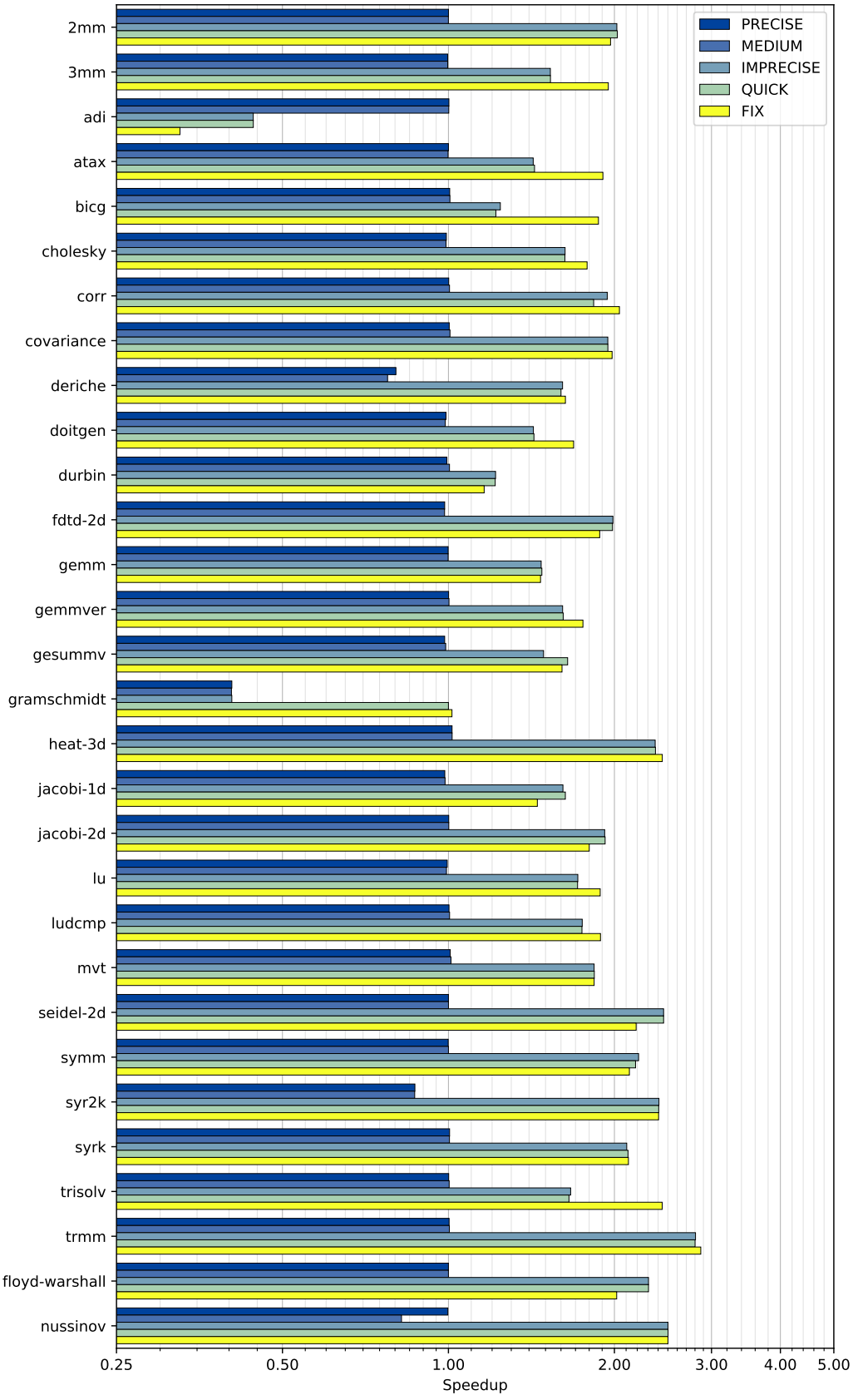

Figure 6.2: Raspberry speedup chart.

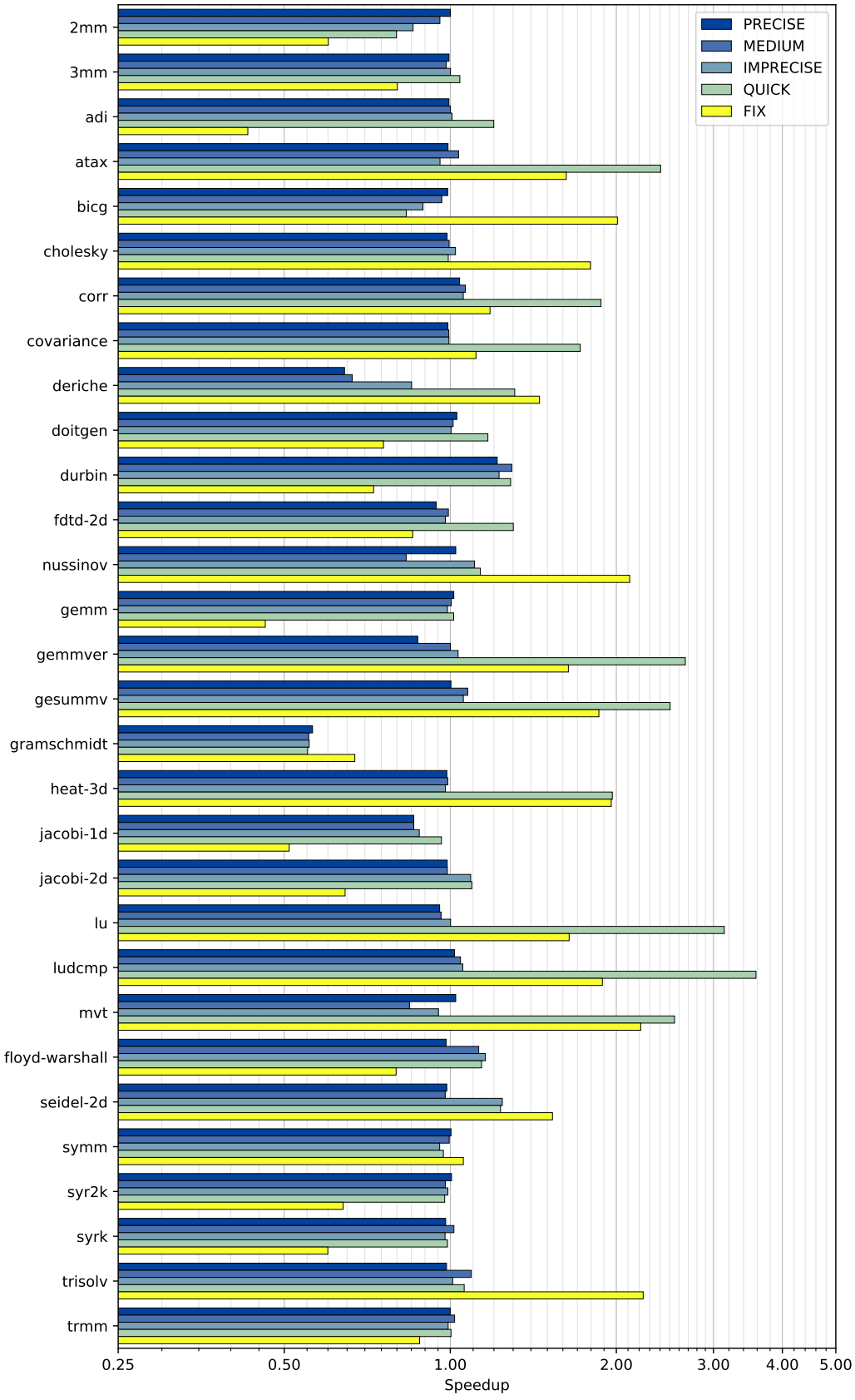

Figure 6.3: Intel speedup chart.

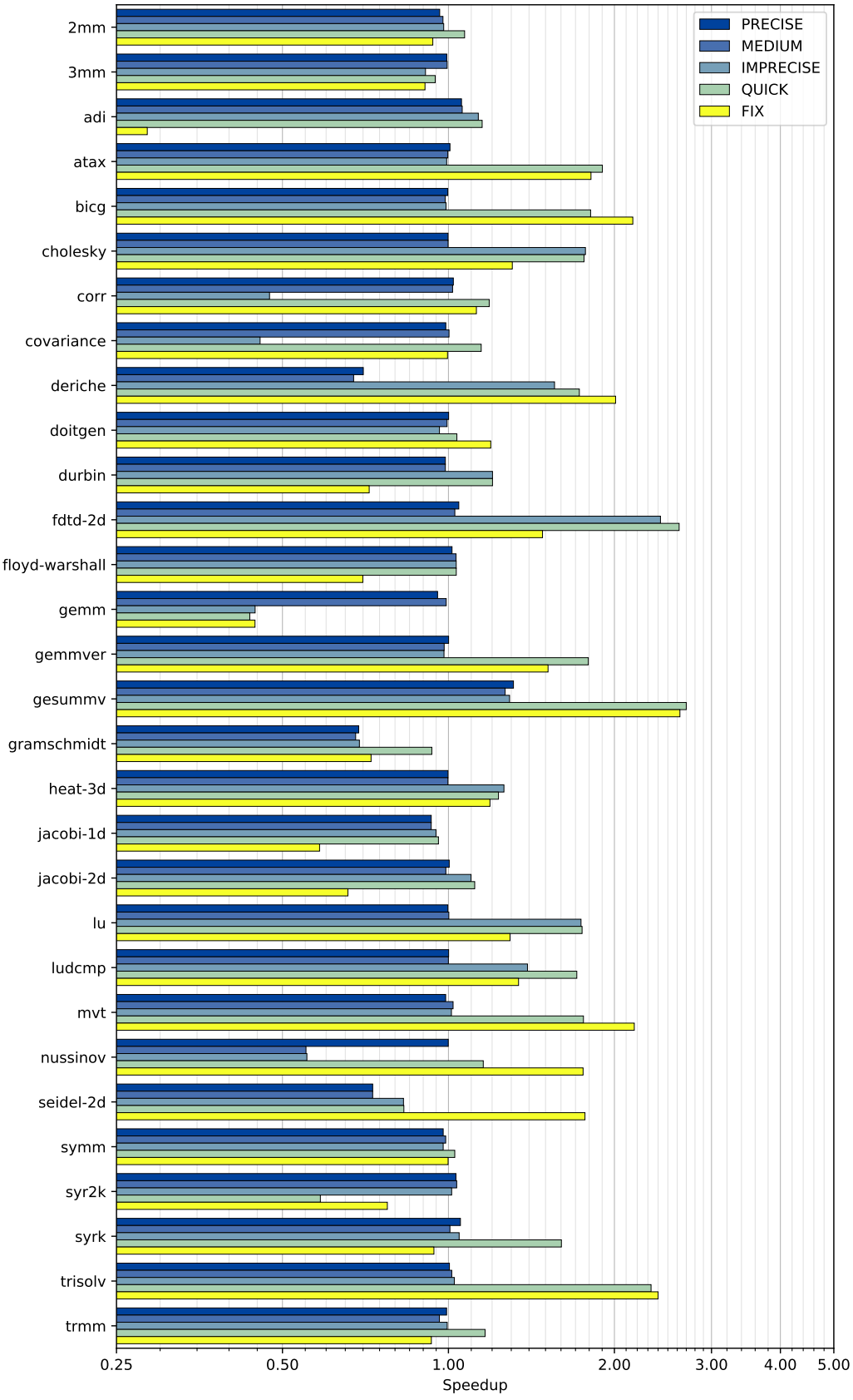

Figure 6.4: AMD speedup chart.

introduced in the computation, the values are still assigned to the correct label. Therefore the total error computed is 0.

Finally, some columns contain a lot of zeros, in particular the left most ones. This is mainly because the algorithm did not find assignment with better execution time than the original one. Therefore the output kernel still uses the same data type, thus producing the same error as the original, especially on architecture with full floating point support.

The test `gramschmidt` represents an exception, as all the versions produced shows error of above 100%. The algorithm, in fact, is very susceptible to noise. As the precision tuning process introduces a lot of such noise by switching to less precise variables, the test present high error in output data. Moreover, the model is not able to handle correctly this case because of the use of nonlinear operation (square root) and divisions inside the computational kernel.

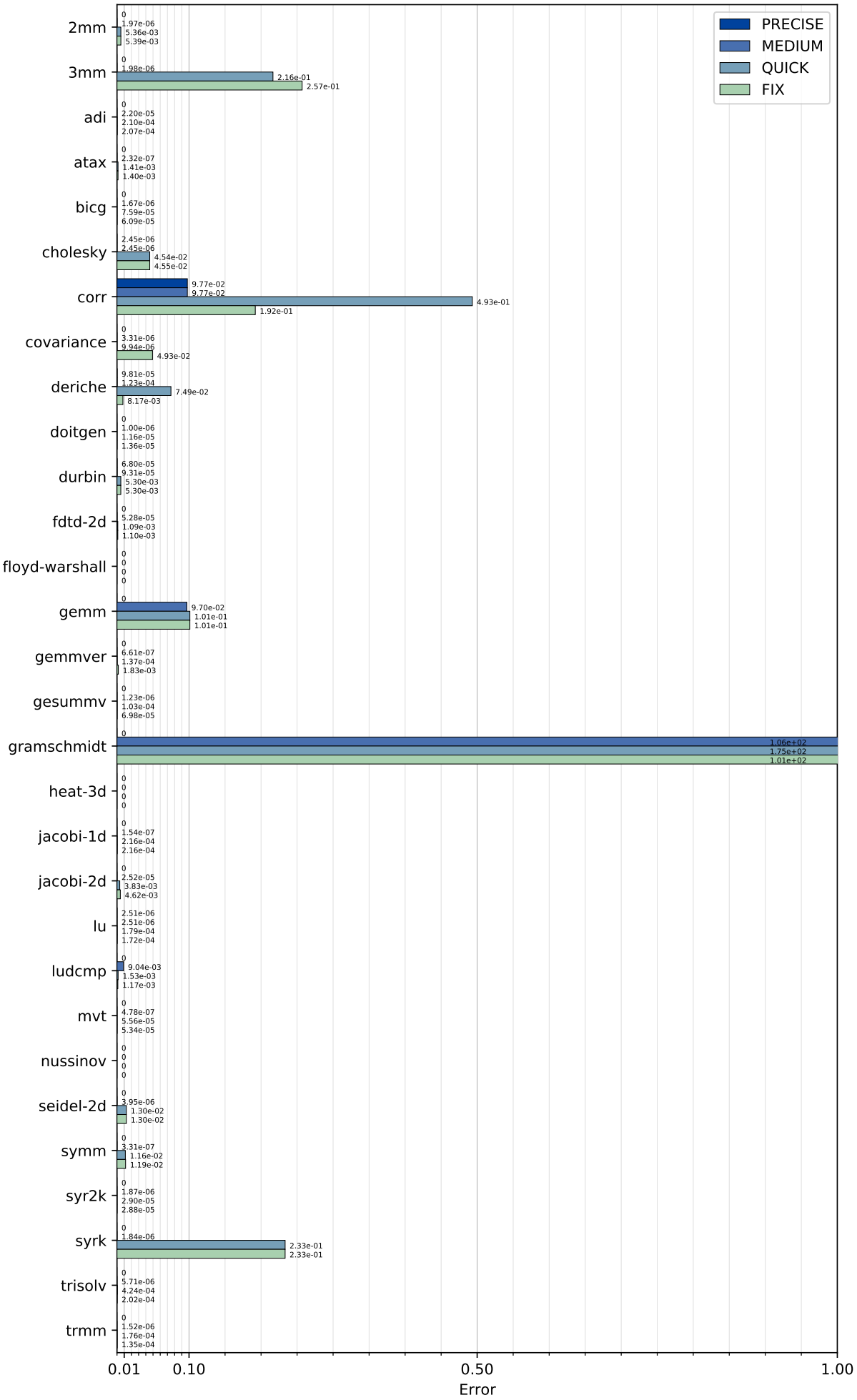

Figure 6.5: Stm32 error chart.

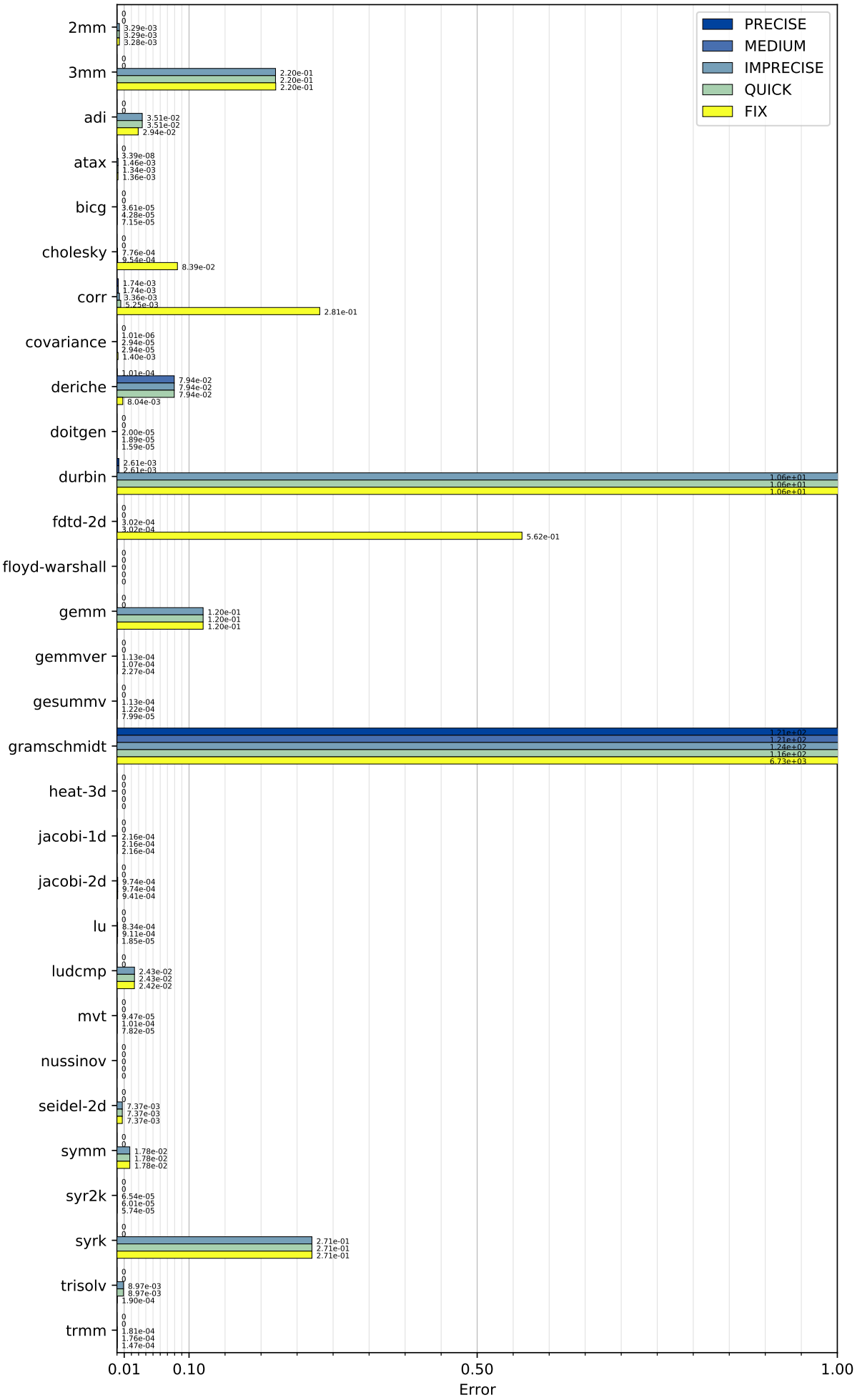

Figure 6.6: Raspberry error chart.

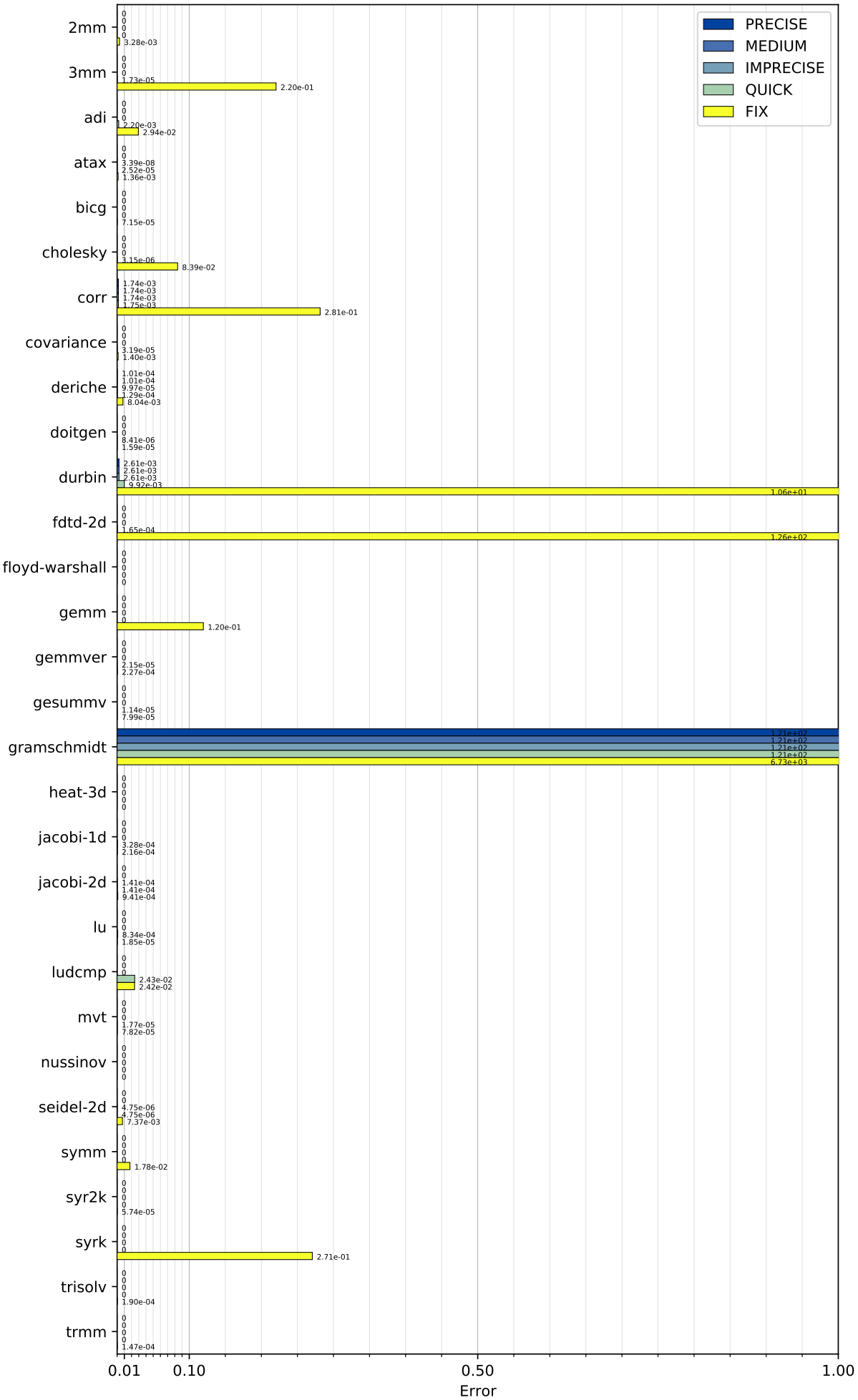

Figure 6.7: Intel error chart.

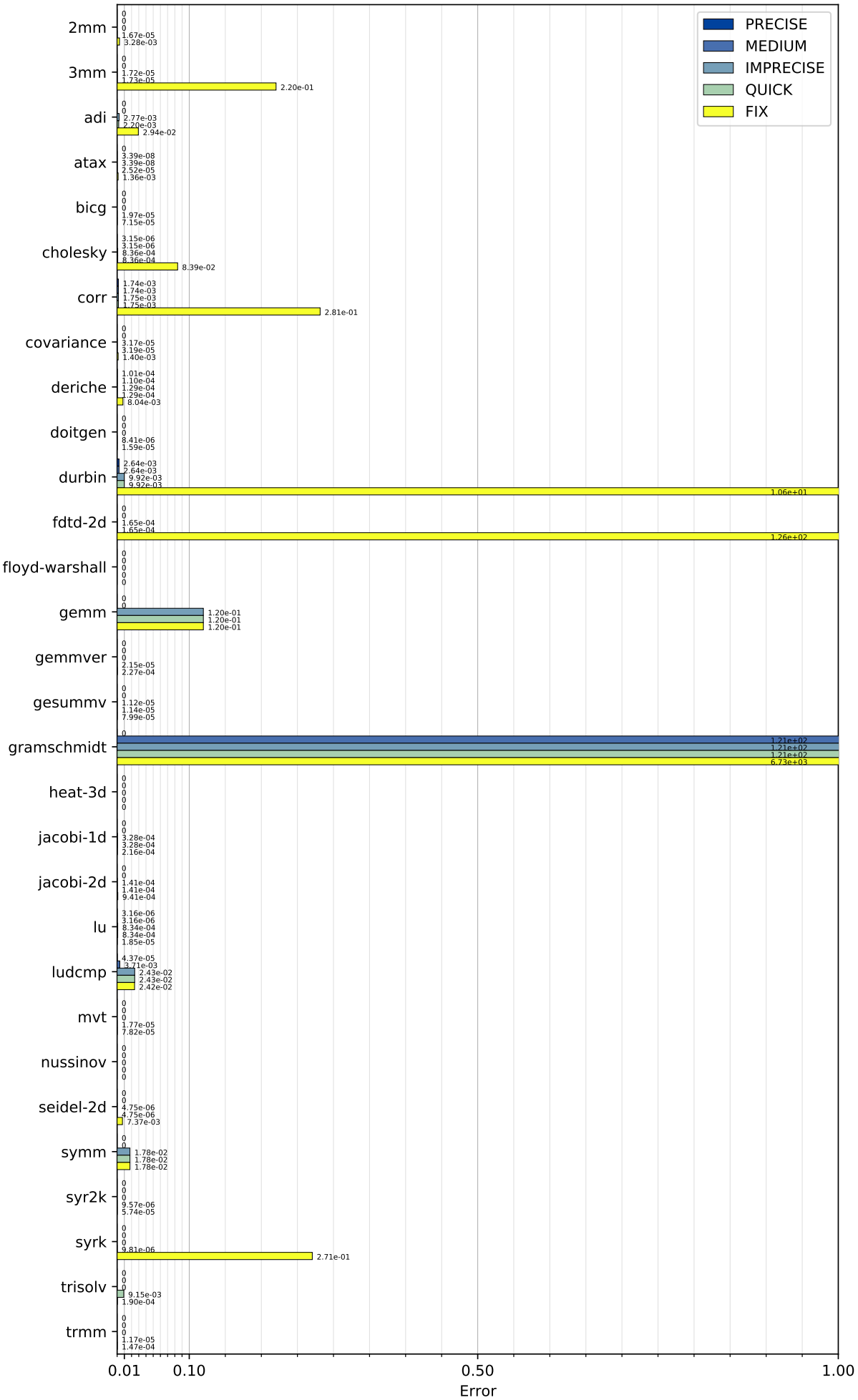

Figure 6.8: AMD error chart.

Target	Time		Error	
	Permissive	Strict	Permissive	Strict
STM32	96.7	83.3	100.0	96.7
Raspberry	96.7	90.0	100.0	83.3
Core2	93.3	76.7	100.0	100.0
AMD	100.0	80.0	96.7	93.3

Table 6.5: Correctly handled test cases for each platform.

6.2.3 Summary of compilation results

In Table 6.5 we report a summary of the results, as an aggregate metric meant to reflect whether the model was successful or not in achieving the required goals with respect to precision and execution time.

The aggregate metric is the percentage of tests that behave correctly to changes in the speed and precision parameters of the model. A test behaves correctly with respect to speed if increasing the speed parameter also determines a reduction of the execution time. In the same way, test behaves correctly with respect to the error if increasing the precision also determines a reduction of the error. In the Permissive column, we show the percentage of tests that behave correctly only considering the Precise and Quick versions. For example, if the Quick version is faster than the Precise version, it behaves correctly with respect to speed. In the Strict column, we show the percentage of tests that behave correctly in all possible combinations of versions.

On all the architectures, the model behaves correctly in the great majority of the cases, showing that speed and precision have been controlled as expected in more than 75% of the benchmarks.

Similarly, with respect to the error, the model generates correct results in more than the 80% of the benchmarks.

6.2.4 Precision mix

In Table 6.6 we show the instruction mix for the Stm32 target architecture, the one that produced the most heterogeneous mixes. The instruction mix measure, as a percentage, how many instruction have been converted to a specific data type, when processed by the model. The instructions are reported as a percentage on the total amount of convertible instructions.

These metrics are obtained directly from output of the model solver.

The analysis of the data shows that when requesting increased precision in the final program, most of the instructions are left as double precision floating point.

Kernel	Precise			Medium			Quick		
	Fix	Float	Double	Fix	Float	Double	Fix	Float	Double
2mm	0	0	100	0	24.6	75.4	70.8	27.7	1.5
3mm	0	2.5	97.5	0	22.8	77.2	82.3	16.5	1.3
adi	0	2.6	97.4	0	14.7	85.3	58.0	39.8	2.2
atax	0	2.3	97.7	0	22.7	77.3	79.5	18.2	2.3
bicg	0	2.0	98.0	0	25.5	74.5	68.6	29.4	2.0
cholesky	0	5.1	94.9	0	11.5	88.5	30.8	65.4	3.8
corr	0	3.8	96.2	0	32.7	67.3	26.0	69.2	4.8
covariance	0	1.7	98.3	0	18.3	81.7	8.3	90	1.7
deriche	0	0	100	7.0	3.5	89.5	76.7	23.3	0
doitgen	0	2.5	97.5	0	22.5	77.5	50	50	0
durbin	0	6.4	93.6	0	12.8	87.2	95.7	4.3	0
fdtd-2d	0	0	100	0	9.2	90.8	100	0	0
floyd-warshall	0	2.7	97.3	8.1	91.9	0	97.3	2.7	0
gemm	0	8.3	91.7	11.1	47.2	41.7	83.3	8.3	8.3
gemmver	0	0	100	0	28	72	71	27	2
gesummv	0	1.7	98.3	0	22.0	78.0	96.6	1.7	1.7
gramschmidt	0	4.7	95.3	0	10.6	89.4	16.5	77.6	5.9
heat-3d	0	0	100	0	2.9	97.1	100	0	0
jacobi-1d	0	10.5	89.5	10.5	15.8	73.7	100	0	0
jacobi-2d	0	2.9	97.1	0	14.5	85.5	100	0	0
lu	0	5.6	94.4	0	12.7	87.3	33.8	64.8	1.4
ludcmp	0	3.5	96.5	0	15.0	85.0	35.4	62.8	1.8
mvt	0	0	100	0	35.1	64.9	61.4	38.6	0
nussinov	6.2	0	93.8	9.2	0	90.8	98.5	0	1.5
seidel-2d	0	0	100	0	3.8	96.2	100	0	0
symm	0	1.5	98.5	0	19.4	80.6	98.5	1.5	0
syr2k	0	0	100	0	23.5	76.5	100	0	0
syrk	0	0	100	0	22.9	77.1	100	0	0
trisolv	0	2.6	97.4	0	15.4	84.6	12.8	87.2	0
trmm	0	2.6	97.4	0	23.1	76.9	97.4	2.6	0

Table 6.6: Instruction mix of the versions for the Stm32 platform. The values are in percentage.

When requesting less precise programs, double precision floating point instructions are mixed with less precise counterparts, like single precision floating point and fixed point instructions.

6.2.5 Compilation times

The compilation times has also been collected for a specific set of tests, in particular for the AMD version. From this data, the compilation slowdown has been computed with respect to using a release version of LLVM. These results are reported in Table C.1 and visible in Figure 6.9.

In the table, the first two columns represent the compilation time with the precedent algorithm and with the new algorithm for mixed precision, respectively. The

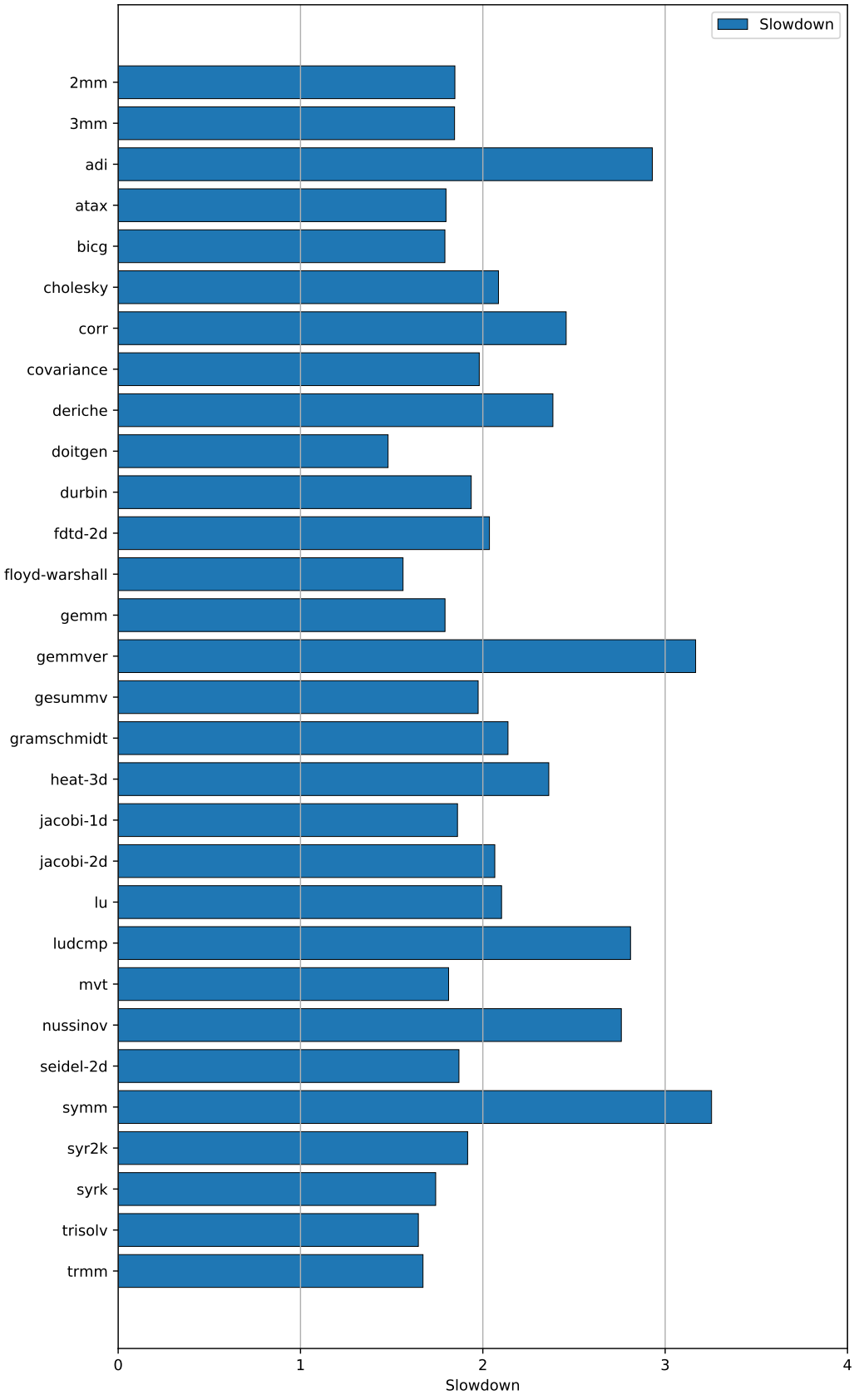

Figure 6.9: Compilation time slowdown for each kernel.

compilation time of the new algorithm is a geometric mean across the four configuration tested.

Finally, in the last column, the slowdown is presented in the form of a ratio between the times. More precisely, the last column is computed as:

$$R_{slowdown} = \frac{t_{new}}{t_{old}}$$

The greatest slowdown achieved was for test `symm`, where the compilation process lasted about thrice the time the original algorithm did, whereas the slowest kernel to compile in absolute terms was `heat-3d` which took about 6 seconds.

This table is important in order to understand the applicability of the methodology proposed. In fact, due to the exploitation of an integer linear programming solver, whose complexity is in the order of $O(2^n)$ — where n is the number of registers and variables allocated in the programs — the compilation time may increase dramatically on larger programs.

Nevertheless, the constants involved are very small and therefore no excessive penalty is introduced during the compilation step for this type of benchmark. Moreover, the code for the new algorithm still contains a great number of debugging statement which partially contribute to the slowdown.

6.2.6 Number of tests

The exploitation of this new analysis method to optimize computational kernels does not give any guarantee on the error of the output or on execution time improvements, but generates different versions of the program with partial guarantees on the monotonicity of the results with relation to the input parameters. Nevertheless, it is possible to note that the number of possible versions that the model generates are limited. Small variation on the input parameters do not produce changes in the output of the model. This remarkably reduces the possible search space generated from the input parameters, limiting the number of kernels that are required to be executes for performing a correctness check.

In order to show this property, we compiled a specific test (`gesummv`) with several fine-grained variations of the parameter, as shown in Table D.1 and in Figures 6.10 and 6.11. The granularity of the parameter variations is in steps of 5. Chosen the time parameter x_{time}, the IEBW parameter is selected as $101 - x_{time}$.

The only changes in the program are present after test 7, 15, 16 and 17. The last 4 tests have the same data type assignments except for the fix point decimal places, for a total of 4 fundamentally different programs.

It is clear that there is no need to finely tune the parameters to the model

because, at least on small programs, nothing change. More precisely, for all but a small fraction of sets of possible parameters there are no variation in execution time, error or number of instructions attributed to the data type. Therefore, we deem the number of versions we chose to generate during the tests, for this type of kernels, as sufficient to characterize the model.

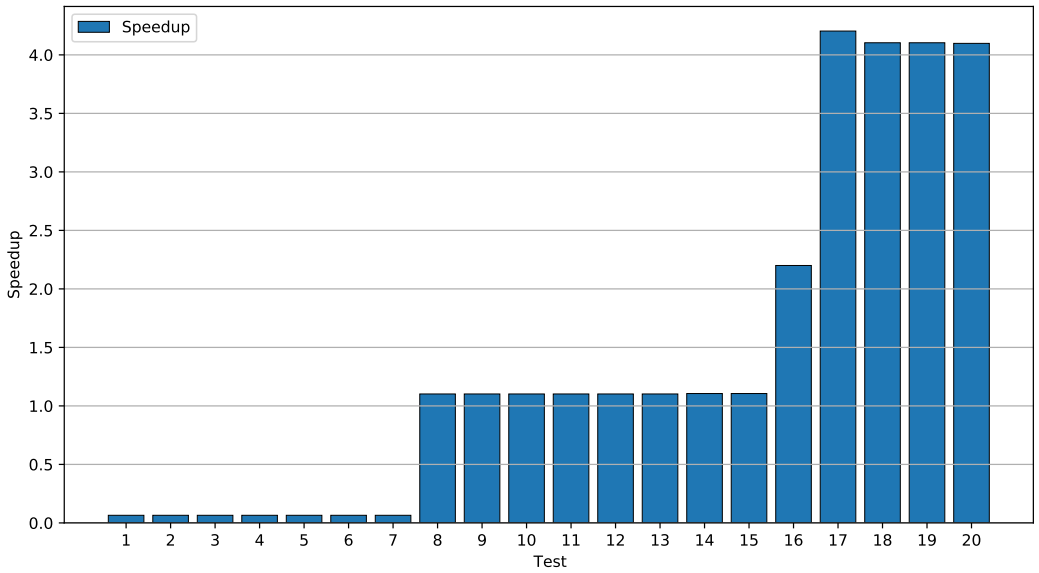

Figure 6.10: `gesummv` speedup.

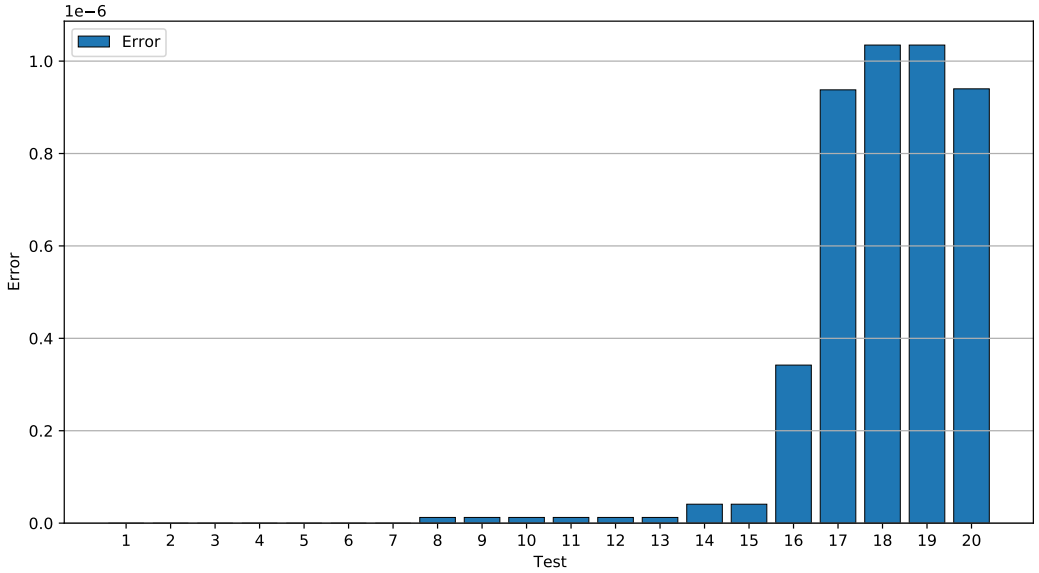

Figure 6.11: `gesummv` error.

Chapter 7

Conclusions

In this thesis, we presented a new technique to enable precision tuning via static analysis and exploitation of integer linear programming. This technique has also been implemented as an extension to a state of the art precision tuning framework, TAFFO, which is based on the well known LLVM compiler toolchain.

This work has been evaluated on a set of standard benchmarks, proving its effectiveness on a wide variety of workloads. The experimental campaign conducted on the proposed implementation reports an overall positive feedback. This solution proved to be particularly well-suited for simple microarchitectures. The model proposed handles successfully more than 75% of the benchmarks tested on every platform with respect to execution time. The model precision goes up to 80% when dealing with the error parameter.

On more complex architectures, the prediction results can be improved in accuracy. On such hardware, the compiler can apply disruptive optimizations – such as automatic vectorization – which can easily invalidate the original model. More work needs to be carried out to correctly predict these systems.

Nevertheless, we expect for this approach to pave the way to the use of static analysis techniques in the precision tuning field. The vast application area of performance modelling for precision tuning still needs to be investigated more before this approach can be applied on a generic microarchitecture.

In practice, a better way to model complex architectures has still to be implemented, together with the support for the conversion of different mathematical functions. Afterwards, the approach can be expanded to support more exotic data types, and then adapted to custom hardware support.

Finally, this solution can also be combined with some other techniques like dynamic recompilation [10] to build different versions of the same kernel which automatically adapt to different input properties.

Bibliography

[1] *Arch Linux ARM*. URL: https://archlinuxarm.org/.

[2] ARM. *ARMv7-M ArchitectureReference Manual*. 2014. URL: https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf.

[3] Robert E Bixby. "A brief history of linear and mixed-integer programming computation". In: *Documenta Math*. 2012.

[4] Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo and Giovanni Agosta. "Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018, pp. 172–176.

[5] Daniele Cattaneo, Michele Chiari, Stefano Cherubin, Antonio Di Bello and Giovanni Agosta. "Feedback-Driven Performance and Precision Tuning for Automatic Fixed Point Exploitation". In: *Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing, PARCO 2019, Prague, Czech Republic, September 10-13, 2019*. Ed. by Ian T. Foster, Gerhard R. Joubert, Ludek Kucera, Wolfgang E. Nagel and Frans J. Peters. Vol. 36. Advances in Parallel Computing. IOS Press, 2019, pp. 299–308. DOI: 10.3233/APC200054. URL: https://doi.org/10.3233/APC200054.

[6] Daniele Cattaneo, Antonio Di Di Bello, Michele Chiari, Stefano Cherubin and Giovanni Agosta. "Fixed Point Exploitation via Compiler Analyses and Transformations: POSTER". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. CF '19. Alghero, Italy: Association for Computing Machinery, 2019, pp. 292–294. ISBN: 9781450366854. DOI: 10.1145/3310273.3323424. URL: https://doi.org/10.1145/3310273.3323424.

[7] 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* (2021). (to appear). ISSN: 2210-5379.

[8] Stefano Cherubin. *fixedpoint - Fixed Point C++ support for approximate computing*. 2018. URL: https://github.com/skeru/fixedpoint.

[9] 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.

[10] Stefano Cherubin, Daniele Cattaneo, Michele Chiari and Giovanni Agosta. "Dynamic Precision Autotuning with TAFFO". In: *ACM Trans. Archit. Code Optim.* 17.2 (May 2020). ISSN: 1544-3566. DOI: 10.1145/3388785. URL: https://doi.org/10.1145/3388785.

[11] Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou and Olivier Sentieys. "Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error". In: *International Conference on Parallel Computing (ParCo)*. Bologna, Italy, Sept. 2017. URL: https://hal.inria.fr/hal-01633790.

[12] 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.

[13] J. N. Coleman, E. I. Chester, C. I. Softley and J. Kadlec. "Arithmetic on the European logarithmic microprocessor". In: *IEEE Transactions on Computers* 49.7 (2000), pp. 702–715.

[14] William Cook. "Markowitz and Manne+ Eastman+ Land and Doig= branch and bound". In: *Optimization Stories* (2012), pp. 227–238.

[15] G. Dantzig. *Programming in a linear structure*. U.S. Air Force Comptroller, 1948.

[16] Eva Darulova, Einar Horn and Saksham Sharma. "Sound Mixed-precision Optimization with Rewriting". In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '18. Porto, Portugal, 2018, pp. 208–219. ISBN: 978-1-5386-5301-2. DOI: 10.1109/ICCPS.2018.00028.

[17] Eva Darulova and Viktor Kuncak. "Sound Compilation of Reals". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 235–248. ISSN: 0362-1340. DOI: 10.1145/2578855.2535874. URL: https://doi.org/10.1145/2578855.2535874.

[18] Nicola Fossati, Daniele Cattaneo, Michele Chiari, Stefano Cherubin and Giovanni Agosta. "Automated Precision Tuning in Activity Classification Systems: A Case Study". In: *Proceedings of the 11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures / 9th*

Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms. PARMA-DITAM'2020. Bologna, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375450. DOI: 10.1145/3381427.3381432. URL: https://doi.org/10.1145/3381427.3381432.

[19] Free Software Foundation. *GMP*. 2019. URL: https://gmplib.org/.

[20] David Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163.

[21] Ralph Gomory. "Early Integer Programming". In: *Operations Research* 50 (Feb. 2002), pp. 78–81. DOI: 10.1287/opre.50.1.78.17793.

[22] Google. *Google OR-Tools*. URL: https://developers.google.com/optimization.

[23] Thomas Gross. "Software implementation of floating-point arithmetic on a reduced-instruction-set processor". In: *Journal of Parallel and Distributed Computing* 2.4 (1985), pp. 362–375. ISSN: 0743-7315. DOI: https://doi.org/10.1016/0743-7315(85)90020-6. URL: http://www.sciencedirect.com/science/article/pii/0743731585900206.

[24] R. Gu, P. Beata and M. Becchi. "Characterizing the Performance/Accuracy Tradeoff of High-Precision Applications via Auto-tuning*". In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 2019, pp. 268–272.

[25] 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*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 333–343. ISBN: 9781450356992. DOI: 10.1145/3213846.3213862. URL: https://doi.org/10.1145/3213846.3213862.

[26] 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.

[27] IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754-1985* (1985), pp. 1–14. DOI: 10.1109/IEEESTD.1985.82928.

[28] IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[29] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1. May 2018. Chap. 8.

[30] Fredrik Johansson. *mpmath*. 2019. URL: http://mpmath.org/.

[31] Mioara Joldes, Jean-Michel Muller, Valentina Popescu and Warwick Tucker. "CAMPARY: Cuda multiple precision arithmetic library and applications." English. In: *Mathematical software – ICMS 2016. 5th international conference, Berlin, Germany, July 11–14, 2016. Proceedings*. Cham: Springer, 2016, pp. 232–240. ISBN: 978-3-319-42431-6/pbk; 978-3-319-42432-3/ebook.

[32] N. Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Combinatorica* 4.4 (Dec. 1984), pp. 373–395. ISSN: 1439-6912. DOI: 10.1007/BF02579150. URL: https://doi.org/10.1007/BF02579150.

[33] Israel Koren. *Computer arithmetic algorithms*. CRC Press, 2018. ISBN: 9781568811604.

[34] Ki-Il Kum, Jiyang Kang and Wonyong Sung. "AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors". In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47.9 (Sept. 2000), pp. 840–848. ISSN: 1057-7130. DOI: 10.1109/82.868453.

[35] 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*. ICS '13. Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 369–378. ISBN: 9781450321303. DOI: 10.1145/2464996.2465018. URL: https://doi.org/10.1145/2464996.2465018.

[36] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[37] Hong Q. Le, J. A. Van Norstrand, B. W. Thompto, J. E. Moreira, D. Q. Nguyen, D. Hrusecky, M. J. Genden and M. Kroener. "IBM POWER9 processor core". In: *IBM Journal of Research and Development* 62.4/5 (July 2018), 2:1–2:12. ISSN: 0018-8646. DOI: 10.1147/JRD.2018.2854039.

[38] C. E. Lemke. "The dual method of solving the linear programming problem". In: *Naval Research Logistics Quarterly* 1.1 (1954), pp. 36–47. DOI: 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.

[39] Nicolas Limare. *time-arit-mat benchmark tool*. 2014. URL: http://nicolas.limare.net/pro/notes/2014/time_arit_math.tgz.

[40] *LLVM feature list*. URL: https://llvm.org/Features.html.

[41] 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: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 190–200. ISBN: 1595930566. DOI: 10.1145/1065010.1065034. URL: https://doi.org/10.1145/1065010.1065034.

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

[43] David Monniaux. "The Pitfalls of Verifying Floating-point Computations". In: *ACM Transactions on Programming Languages and Systems* 30.3 (May 2008), 12:1–12:41. ISSN: 0164-0925. DOI: 10.1145/1353445.1353446.

[44] Diego Novillo et al. "Memory SSA-a unified approach for sparsely representing memory operations". In: *Proceedings of the GCC Developers' Summit*. Citeseer. 2007, pp. 97–110.

[45] Johan Nyström. *Fixmath - Fixed Point Library*. 2012. URL: http://savannah.nongnu.org/projects/fixmath/.

[46] *Olimex ARM-USB-OCD JTAG debugger*. URL: https://www.olimex.com/Products/ARM/JTAG/ARM-USB-OCD/.

[47] Christos H Papadimitriou. "On the complexity of integer programming". In: *Journal of the ACM (JACM)* 28.4 (1981), pp. 765–768.

[48] Antonio Pullini. "Design of Energy Efficient Microcontrollers". PhD thesis. ETH Zurich, 2019.

[49] PyPA. *pip package installer for Python*. URL: https://pip.pypa.io.

[50] *Raspberry Foundation cross-compile toolchain*. URL: https://www.raspberrypi.org/documentation/linux/kernel/building.md.

[51] Fabrice Rastello. *SSA-Based Compiler Design*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 1441962018.

[52] Guido van Rossum. *Python Programming Language*. URL: https://www.python.org/about/.

[53] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu and David Hough. "Precimonious: Tuning Assistant for Floating-Point Precision". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503296. URL: https://doi.org/10.1145/2503210.2503296.

[54] Cristina Silvano et al. "Autotuning and Adaptivity in Energy Efficient HPC Systems: The ANTAREX Toolbox". In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. CF '18. Ischia, Italy: Association for Computing Machinery, 2018, pp. 270–275. ISBN: 9781450357616. DOI: 10.1145/3203217.3205338. URL: https://doi.org/10.1145/3203217.3205338.

[55] Cristina Silvano et al. "The ANTAREX tool flow for monitoring and autotuning energy efficient HPC systems". In: *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2017, pp. 308–316.

[56] Michael A Soderstrand, W Kenneth Jenkins, Graham A Jullien and Fred J Taylor, eds. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*. IEEE Press, 1986. ISBN: 087942205X.

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

[58] Federico Terraneo. *Miosix embedded OS*. 2008. URL: https://miosix.org/.

[59] Vasek Chvatal. *Linear Programming*. 1983. Chap. 1.

[60] Tomofumi Yuki. *Understanding PolyBench/C 3.2 kernels*. 2014. URL: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

Appendix A

Speedup data

Kernel	Precise	Medium	Quick	Fix
2mm	4.57	16.31	140.83	355.19
3mm	3.25	16.51	177.56	333.96
adi	0.56	42.83	165.77	486.59
atax	5.21	73.23	231.65	393.16
bicg	6.38	72.84	147.45	411.41
cholesky	5.61	8.16	15.93	277.28
corr	60.09	11.99	2.26	195.19
covariance	2.77	17.18	79.22	327.81
deriche	0.05	4.02	120.20	174.87
doitgen	4.65	17.84	94.68	254.31
durbin	5.80	4.75	303.27	291.34
fdtd-2d	1.27	5.71	27.71	620.92
floyd-warshall	16.61	−22.16	344.05	282.14
gemm	4.59	−7.43	166.46	158.77
gemmver	7.30	36.51	133.73	399.61
gesummv	6.53	110.10	409.68	420.42
gramschmidt	0.93	1.00	0.91	3.88
heat-3d	1.56	1.98	859.51	823.81
jacobi-1d	−0.62	1.63	636.33	594.91
jacobi-2d	1.76	5.16	758.23	725.43
lu	5.55	7.75	12.31	304.02
ludcmp	0.58	5.49	29.62	272.07
mvt	6.96	62.22	112.16	308.69
nussinov	−5.52	−42.45	229.32	224.09
seidel-2d	2.40	40.09	583.43	588.03
symm	1.54	16.58	432.26	434.00
syr2k	2.10	16.76	576.94	579.82
syrk	3.97	23.06	385.04	400.06
trisolv	6.97	78.26	156.16	346.91
trmm	5.02	27.13	355.32	352.13

Table A.1: Speedup for Stm32 target platform. Results are shown in Figure 6.1.

Kernel	Precise	Medium	Imprecise	Quick	Fix
2mm	0.02	−0.01	102.04	102.43	96.80
3mm	−0.32	−0.30	53.06	53.12	94.96
adi	0.20	0.15	−55.74	−55.73	−67.41
atax	0.00	−0.17	42.48	43.26	90.61
bicg	0.53	0.62	24.20	21.91	87.09
cholesky	−0.86	−1.00	62.77	62.63	78.44
corr	0.12	0.45	94.11	83.40	104.24
covariance	0.33	0.64	94.55	94.61	98.07
deriche	−19.69	−22.43	61.05	60.00	62.99
doitgen	−1.00	−1.31	42.59	42.92	68.66
durbin	−0.64	0.44	21.75	21.57	16.09
fdtd-2d	−1.56	−1.60	98.75	98.41	88.08
floyd-warshall	−0.01	−0.00	130.60	130.59	101.91
gemm	−0.13	−0.13	47.29	47.73	46.90
gemmver	0.07	0.24	61.09	61.49	75.37
gesummv	−1.56	−1.14	48.76	64.50	60.69
gramschmidt	0.40	0.40	0.40	1.00	1.01
heat-3d	1.52	1.56	137.12	137.39	144.25
jacobi-1d	−1.51	−1.39	61.31	62.94	44.96
jacobi-2d	0.12	0.15	92.04	92.38	80.04
lu	−0.51	−0.81	71.65	71.47	88.47
ludcmp	0.24	0.48	74.84	74.65	88.70
mvt	0.76	1.07	83.74	83.89	83.75
nussinov	−0.25	−17.79	150.11	150.22	150.12
seidel-2d	0.00	0.01	145.61	145.61	118.99
symm	−0.10	0.00	121.24	118.45	112.91
syr2k	−13.07	−13.14	140.88	140.64	140.52
syrk	0.48	0.43	110.56	111.69	112.05
trisolv	0.11	0.31	66.59	65.44	144.17
trmm	0.28	0.39	180.62	180.09	186.85

Table A.2: Speedup for Raspberry target platform. Results are shown in Figure 6.2.

Kernel	Precise	Medium	Imprecise	Quick	Fix
2mm	0.03	−4.24	−14.38	−20.12	−39.90
3mm	−0.53	−1.65	0.09	4.09	−19.82
adi	−0.57	0.09	0.72	19.90	−57.05
atax	−0.94	3.52	−4.20	140.65	62.30
bicg	−1.07	−3.48	−10.76	−16.79	100.92
cholesky	−1.35	−0.48	2.20	−0.83	79.53
corr	3.98	6.52	5.57	87.61	18.05
covariance	−0.98	−0.63	−0.56	72.04	11.32
deriche	−35.68	−33.61	−14.91	30.88	45.13
doitgen	2.76	1.19	0.33	17.00	−24.32
durbin	21.64	29.27	22.59	28.65	−27.39
fdtd-2d	−5.71	−0.79	−2.04	30.10	−14.53
floyd-warshall	−1.72	12.60	15.78	13.94	−20.24
gemm	1.47	0.39	−1.16	1.44	−53.80
gemmver	−12.69	0.06	3.32	166.75	63.72
gesummv	0.27	7.56	5.61	150.36	85.96
gramschmidt	−43.73	−44.63	−44.51	−44.87	−32.87
heat-3d	−1.44	−1.05	−1.99	96.69	95.82
jacobi-1d	−14.13	−14.13	−12.14	−3.62	−48.96
jacobi-2d	−1.28	−1.21	8.90	9.38	−35.52
lu	−4.33	−3.76	0.13	213.89	64.26
ludcmp	1.80	4.36	5.36	258.26	88.70
mvt	2.31	−15.64	−4.81	155.01	121.46
nussinov	2.32	−16.81	10.61	13.40	111.47
seidel-2d	−1.45	−2.08	24.26	23.33	53.22
symm	0.30	−0.43	−4.33	−2.83	5.61
syr2k	0.52	−2.01	−0.99	−2.31	−36.07
syrk	−1.90	1.53	−2.15	−1.17	−40.01
trisolv	−1.55	9.17	1.00	5.98	123.73
trmm	−0.03	1.83	−0.88	0.31	−12.00

Table A.3: Speedup for Intel target platform. Results are shown in Figure 6.3.

Kernel	Precise	Medium	Imprecise	Quick	Fix
2mm	−3.59	−2.26	−1.96	6.98	−6.32
3mm	−0.68	−0.53	−9.14	−5.36	−9.32
adi	5.61	5.93	13.29	15.12	−71.57
atax	0.64	−0.27	−0.73	90.18	81.36
bicg	−0.29	−1.32	−0.99	81.07	115.98
cholesky	−0.17	−0.15	77.21	76.16	30.57
corr	2.07	1.75	−52.63	18.56	12.39
covariance	−1.10	0.25	−54.46	14.66	−0.38
deriche	−29.92	−32.69	55.75	72.75	100.99
doitgen	0.08	−0.57	−3.68	3.59	19.33
durbin	−1.28	−1.28	20.21	20.21	−28.17
fdtd-2d	4.42	2.71	142.41	162.06	48.12
floyd-warshall	1.51	3.23	3.30	3.32	−30.06
gemm	−4.43	−0.93	−55.39	−56.38	−55.42
gemmver	0.07	−1.76	−1.84	79.34	51.59
gesummv	31.20	26.72	29.09	169.99	162.91
gramschmidt	−31.25	−32.12	−31.05	−6.70	−27.61
heat-3d	−0.20	−0.24	26.09	23.24	18.94
jacobi-1d	−6.93	−6.93	−5.05	−4.08	−41.61
jacobi-2d	0.33	−1.07	9.90	11.60	−34.28
lu	−0.25	0.17	73.96	74.73	29.32
ludcmp	0.08	−0.01	39.11	70.93	34.02
mvt	−1.15	1.96	1.26	75.74	117.30
nussinov	−0.07	−44.83	−44.63	15.66	75.47
seidel-2d	−27.12	−27.12	−17.11	−16.98	76.86
symm	−2.16	−1.14	−2.17	2.68	−0.14
syr2k	3.21	3.51	1.35	−41.42	−22.52
syrk	5.14	0.63	4.57	60.28	−5.88
trisolv	0.42	1.40	2.48	133.06	140.03
trmm	−0.77	−3.71	−0.51	16.54	−6.88

Table A.4: Speedup for AMD target platform. Results are shown in Figure 6.4.

Appendix B

Error data

Kernel	Precise	Medium	Quick	Fix
2mm	0	1.97×10^{-6}	5.36×10^{-3}	5.39×10^{-3}
3mm	0	1.98×10^{-6}	2.16×10^{-1}	2.57×10^{-1}
adi	0	2.20×10^{-5}	2.10×10^{-4}	2.07×10^{-4}
atax	0	2.32×10^{-7}	1.41×10^{-3}	1.40×10^{-3}
bicg	0	1.67×10^{-6}	7.59×10^{-5}	6.09×10^{-5}
cholesky	2.45×10^{-6}	2.45×10^{-6}	4.54×10^{-2}	4.55×10^{-2}
corr	9.77×10^{-2}	9.77×10^{-2}	4.93×10^{-1}	1.92×10^{-1}
covariance	0	3.31×10^{-6}	9.94×10^{-6}	4.93×10^{-2}
deriche	9.81×10^{-5}	1.23×10^{-4}	7.49×10^{-2}	8.17×10^{-3}
doitgen	0	1.00×10^{-6}	1.16×10^{-5}	1.36×10^{-5}
durbin	6.80×10^{-5}	9.31×10^{-5}	5.30×10^{-3}	5.30×10^{-3}
fdtd-2d	0	5.28×10^{-5}	1.09×10^{-3}	1.10×10^{-3}
floyd-warshall	0	0	0	0
gemm	0	9.70×10^{-2}	1.01×10^{-1}	1.01×10^{-1}
gemmver	0	6.61×10^{-7}	1.37×10^{-4}	1.83×10^{-3}
gesummv	0	1.23×10^{-6}	1.03×10^{-4}	6.98×10^{-5}
gramschmidt	0	1.06×10^{2}	1.75×10^{2}	1.01×10^{2}
heat-3d	0	0	0	0
jacobi-1d	0	1.54×10^{-7}	2.16×10^{-4}	2.16×10^{-4}
jacobi-2d	0	2.52×10^{-5}	3.83×10^{-3}	4.62×10^{-3}
lu	2.51×10^{-6}	2.51×10^{-6}	1.79×10^{-4}	1.72×10^{-4}
ludcmp	0	9.04×10^{-3}	1.53×10^{-3}	1.17×10^{-3}
mvt	0	4.78×10^{-7}	5.56×10^{-5}	5.34×10^{-5}
nussinov	0	0	0	0
seidel-2d	0	3.95×10^{-6}	1.30×10^{-2}	1.30×10^{-2}
symm	0	3.31×10^{-7}	1.16×10^{-2}	1.19×10^{-2}
syr2k	0	1.87×10^{-6}	2.90×10^{-5}	2.88×10^{-5}
syrk	0	1.84×10^{-6}	2.33×10^{-1}	2.33×10^{-1}
trisolv	0	5.71×10^{-6}	4.24×10^{-4}	2.02×10^{-4}
trmm	0	1.52×10^{-6}	1.76×10^{-4}	1.35×10^{-4}

Table B.1: MPE for Stm32 target platform. Results are shown in Figure 6.5

Kernel	Precise	Medium	Imprecise	Quick	Fix
2mm	0	0	3.29×10^{-3}	3.29×10^{-3}	3.28×10^{-3}
3mm	0	0	2.20×10^{-1}	2.20×10^{-1}	2.20×10^{-1}
adi	0	0	3.51×10^{-2}	3.51×10^{-2}	2.94×10^{-2}
atax	0	3.39×10^{-8}	1.46×10^{-3}	1.34×10^{-3}	1.36×10^{-3}
bicg	0	0	3.61×10^{-5}	4.28×10^{-5}	7.15×10^{-5}
cholesky	0	0	7.76×10^{-4}	9.54×10^{-4}	8.39×10^{-2}
corr	1.74×10^{-3}	1.74×10^{-3}	3.36×10^{-3}	5.25×10^{-3}	2.81×10^{-1}
covariance	0	1.01×10^{-6}	2.94×10^{-5}	2.94×10^{-5}	1.40×10^{-3}
deriche	1.01×10^{-4}	7.94×10^{-2}	7.94×10^{-2}	7.94×10^{-2}	8.04×10^{-3}
doitgen	0	0	2.00×10^{-5}	1.89×10^{-5}	1.59×10^{-5}
durbin	2.61×10^{-3}	2.61×10^{-3}	1.06×10^{1}	1.06×10^{1}	1.06×10^{1}
fdtd-2d	0	0	3.02×10^{-4}	3.02×10^{-4}	5.62×10^{-1}
floyd-warshall	0	0	0	0	0
gemm	0	0	1.20×10^{-1}	1.20×10^{-1}	1.20×10^{-1}
gemmver	0	0	1.13×10^{-4}	1.07×10^{-4}	2.27×10^{-4}
gesummv	0	0	1.13×10^{-4}	1.22×10^{-4}	7.99×10^{-5}
gramschmidt	1.21×10^{2}	1.21×10^{2}	1.24×10^{2}	1.16×10^{2}	6.73×10^{3}
heat-3d	0	0	0	0	0
jacobi-1d	0	0	2.16×10^{-4}	2.16×10^{-4}	2.16×10^{-4}
jacobi-2d	0	0	9.74×10^{-4}	9.74×10^{-4}	9.41×10^{-4}
lu	0	0	8.34×10^{-4}	9.11×10^{-4}	1.85×10^{-5}
ludcmp	0	0	2.43×10^{-2}	2.43×10^{-2}	2.42×10^{-2}
mvt	0	0	9.47×10^{-5}	1.01×10^{-4}	7.82×10^{-5}
nussinov	0	0	0	0	0
seidel-2d	0	0	7.37×10^{-3}	7.37×10^{-3}	7.37×10^{-3}
symm	0	0	1.78×10^{-2}	1.78×10^{-2}	1.78×10^{-2}
syr2k	0	0	6.54×10^{-5}	6.01×10^{-5}	5.74×10^{-5}
syrk	0	0	2.71×10^{-1}	2.71×10^{-1}	2.71×10^{-1}
trisolv	0	0	8.97×10^{-3}	8.97×10^{-3}	1.90×10^{-4}
trmm	0	0	1.81×10^{-4}	1.76×10^{-4}	1.47×10^{-4}

Table B.2: MPE for Raspberry target platform. Results are shown in Figure 6.6

Kernel	Precise	Medium	Imprecise	Quick	Fix
2mm	0	0	0	0	3.28×10^{-3}
3mm	0	0	0	1.73×10^{-5}	2.20×10^{-1}
adi	0	0	0	2.20×10^{-3}	2.94×10^{-2}
atax	0	0	3.39×10^{-8}	2.52×10^{-5}	1.36×10^{-3}
bicg	0	0	0	0	7.15×10^{-5}
cholesky	0	0	0	3.15×10^{-6}	8.39×10^{-2}
corr	1.74×10^{-3}	1.74×10^{-3}	1.74×10^{-3}	1.75×10^{-3}	2.81×10^{-1}
covariance	0	0	0	3.19×10^{-5}	1.40×10^{-3}
deriche	1.01×10^{-4}	1.01×10^{-4}	9.97×10^{-5}	1.29×10^{-4}	8.04×10^{-3}
doitgen	0	0	0	8.41×10^{-6}	1.59×10^{-5}
durbin	2.61×10^{-3}	2.61×10^{-3}	2.61×10^{-3}	9.92×10^{-3}	1.06×10^{1}
fdtd-2d	0	0	0	1.65×10^{-4}	1.26×10^{2}
floyd-warshall	0	0	0	0	0
gemm	0	0	0	0	1.20×10^{-1}
gemmver	0	0	0	2.15×10^{-5}	2.27×10^{-4}
gesummv	0	0	0	1.14×10^{-5}	7.99×10^{-5}
gramschmidt	1.21×10^{2}	1.21×10^{2}	1.21×10^{2}	1.21×10^{2}	6.73×10^{3}
heat-3d	0	0	0	0	0
jacobi-1d	0	0	0	3.28×10^{-4}	2.16×10^{-4}
jacobi-2d	0	0	1.41×10^{-4}	1.41×10^{-4}	9.41×10^{-4}
lu	0	0	0	8.34×10^{-4}	1.85×10^{-5}
ludcmp	0	0	0	2.43×10^{-2}	2.42×10^{-2}
mvt	0	0	0	1.77×10^{-5}	7.82×10^{-5}
nussinov	0	0	0	0	0
seidel-2d	0	0	4.75×10^{-6}	4.75×10^{-6}	7.37×10^{-3}
symm	0	0	0	0	1.78×10^{-2}
syr2k	0	0	0	0	5.74×10^{-5}
syrk	0	0	0	0	2.71×10^{-1}
trisolv	0	0	0	0	1.90×10^{-4}
trmm	0	0	0	0	1.47×10^{-4}

Table B.3: MPE for Intel target platform. Results are shown in Figure 6.7

Kernel	Precise	Medium	Imprecise	Quick	Fix
2mm	0	0	0	1.67×10^{-5}	3.28×10^{-3}
3mm	0	0	1.72×10^{-5}	1.73×10^{-5}	2.20×10^{-1}
adi	0	0	2.77×10^{-3}	2.20×10^{-3}	2.94×10^{-2}
atax	0	3.39×10^{-8}	3.39×10^{-8}	2.52×10^{-5}	1.36×10^{-3}
bicg	0	0	0	1.97×10^{-5}	7.15×10^{-5}
cholesky	3.15×10^{-6}	3.15×10^{-6}	8.36×10^{-4}	8.36×10^{-4}	8.39×10^{-2}
corr	1.74×10^{-3}	1.74×10^{-3}	1.75×10^{-3}	1.75×10^{-3}	2.81×10^{-1}
covariance	0	0	3.17×10^{-5}	3.19×10^{-5}	1.40×10^{-3}
deriche	1.01×10^{-4}	1.10×10^{-4}	1.29×10^{-4}	1.29×10^{-4}	8.04×10^{-3}
doitgen	0	0	0	8.41×10^{-6}	1.59×10^{-5}
durbin	2.64×10^{-3}	2.64×10^{-3}	9.92×10^{-3}	9.92×10^{-3}	1.06×10^{1}
fdtd-2d	0	0	1.65×10^{-4}	1.65×10^{-4}	1.26×10^{2}
floyd-warshall	0	0	0	0	0
gemm	0	0	1.20×10^{-1}	1.20×10^{-1}	1.20×10^{-1}
gemmver	0	0	0	2.15×10^{-5}	2.27×10^{-4}
gesummv	0	0	1.12×10^{-5}	1.14×10^{-5}	7.99×10^{-5}
gramschmidt	1.21×10^{2}	1.21×10^{2}	1.21×10^{2}	0	6.73×10^{3}
heat-3d	0	0	0	0	0
jacobi-1d	0	0	3.28×10^{-4}	3.28×10^{-4}	2.16×10^{-4}
jacobi-2d	0	0	1.41×10^{-4}	1.41×10^{-4}	9.41×10^{-4}
lu	3.16×10^{-6}	3.16×10^{-6}	8.34×10^{-4}	8.34×10^{-4}	1.85×10^{-5}
ludcmp	4.37×10^{-5}	3.71×10^{-3}	2.43×10^{-2}	2.43×10^{-2}	2.42×10^{-2}
mvt	0	0	0	1.77×10^{-5}	7.82×10^{-5}
nussinov	0	0	0	0	0
seidel-2d	0	0	4.75×10^{-6}	4.75×10^{-6}	7.37×10^{-3}
symm	0	0	1.78×10^{-2}	1.78×10^{-2}	1.78×10^{-2}
syr2k	0	0	0	9.57×10^{-6}	5.74×10^{-5}
syrk	0	0	0	9.81×10^{-6}	2.71×10^{-1}
trisolv	0	0	0	9.15×10^{-3}	1.90×10^{-4}
trmm	0	0	0	1.17×10^{-5}	1.47×10^{-4}

Table B.4: MPE for AMD target platform. Results are shown in Figure 6.8

Appendix C

Compilation slowdown data

Kernel	**Original** (s)	**New** (s)	**Slowdown**
`2mm`	0.86	1.59	1.85
`3mm`	1.09	2.00	1.84
`adi`	1.88	5.50	2.93
`atax`	0.51	0.91	1.80
`bicg`	0.53	0.94	1.79
`cholesky`	0.76	1.58	2.09
`corr`	0.72	1.77	2.46
`covariance`	0.58	1.15	1.98
`deriche`	0.92	2.20	2.38
`doitgen`	0.98	1.45	1.48
`durbin`	0.55	1.07	1.94
`fdtd-2d`	1.28	2.61	2.04
`floyd-warshall`	0.67	1.04	1.56
`gemm`	0.67	1.20	1.79
`gemmver`	0.66	2.09	3.17
`gesummv`	0.50	0.99	1.97
`gramschmidt`	0.82	1.75	2.14
`heat-3d`	2.52	5.95	2.36
`jacobi-1d`	0.47	0.87	1.86
`jacobi-2d`	0.77	1.59	2.07
`lu`	0.77	1.62	2.10
`ludcmp`	0.93	2.61	2.81
`mvt`	0.55	0.99	1.81
`nussinov`	0.64	1.75	2.76
`seidel-2d`	0.63	1.19	1.87
`symm`	0.67	2.16	3.25
`syr2k`	0.62	1.20	1.92
`syrk`	0.57	1.00	1.74
`trisolv`	0.48	0.79	1.65
`trmm`	0.55	0.92	1.67

Table C.1: Compilation time and slowdown for each kernel analyzed. These results are shown in Figure 6.9. Times are reported in seconds.

Appendix D

`gesummv` data

N	Config		Results		Mix		
	Time	IEBW	Error	Speedup	Fix	Float	Double
1	1	100	0	0.07	0	1	58
2	6	95	0	0.07	0	1	58
3	11	90	0	0.07	0	1	58
4	16	85	0	0.07	0	1	58
5	21	80	0	0.07	0	1	58
6	26	75	0	0.07	0	1	58
7	31	70	0	0.07	0	1	58
8	36	65	1.23×10^{-8}	1.10	0	13	46
9	41	60	1.23×10^{-8}	1.10	0	13	46
10	46	55	1.23×10^{-8}	1.10	0	13	46
11	51	50	1.23×10^{-8}	1.10	0	13	46
12	56	45	1.23×10^{-8}	1.10	0	13	46
13	61	40	1.23×10^{-8}	1.10	0	13	46
14	66	35	4.10×10^{-8}	1.11	0	17	42
15	71	30	4.10×10^{-8}	1.11	0	17	42
16	76	25	3.42×10^{-7}	2.20	24	34	1
17	81	20	9.38×10^{-7}	4.20	57	1	1
18	86	15	1.03×10^{-6}	4.10	57	1	1
19	91	10	1.03×10^{-6}	4.10	57	1	1
20	96	5	9.40×10^{-7}	4.10	57	1	1

Table D.1: Data collected for test `gesummv` with small changes in the parameters. These results are shown in Figures 6.10 and 6.11