Trade-offs of certified fixed-point code synthesis for linear algebra basic blocks

Matthieu Martel, Mohamed Amine Najahi, Guillaume Revy

Trade-offs of certified fixed-point code synthesis for linear algebra basic blocks

Matthieu Martel[1] Amine Najahi[2,3,4] Guillaume Revy[2,3,4]

[1] *Univ. Perpignan Via Domitia, Laboratoire LAMPS, F-66860, Perpignan, France*
[2] *Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France*
[3] *Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France*
[4] *CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France*

{*matthieu.martel, amine.najahi, guillaume.revy*}*@univ-perp.fr*

Abstract

In embedded systems, efficient implementations of numerical algorithms typically use the fixed-point arithmetic rather than the standardized and costly floating-point arithmetic. But, fixed-point programmers face two difficulties: First, writing fixed-point codes is tedious and error prone. Second, the low dynamic range of fixed-point numbers leads to the persistent belief that fixed-point computations are inherently inaccurate. In this article, we address these two limitations by introducing a methodology to design and implement tools that synthesize fixed-point programs. To strengthen the user's confidence in the synthesized code, analytic methods are presented to automatically assert its numerical quality. Furthermore, we use this framework to generate fixed-point code for linear algebra basic blocks such as matrix multiplication and inversion. For example, the former task involves trade-offs such as choosing to maximize the code's accuracy or minimize its size. For the two cases of matrix multiplication and inversion, we describe, implement, and experiment with several algorithms to find trade-offs between the conflicting goals.

Keywords: Fixed-point arithmetic, Code generation, Certified numerical accuracy, Numerical linear algebra.

1. Introduction

Fixed-point arithmetic is a lightweight alternative to floating-point arithmetic. It does not require dedicated hardware, namely a floating-point unit, and executes more efficiently. However, developing fixed-point implementations requires numerical expertise from the programmer, is time consuming, and error prone. Moreover, the correctness and the numerical quality of the produced codes are not guaranteed since they depend solely on the programmer.

In a typical design, DSP programmers prototype and simulate their algorithms in high level environments like MATLAB. These environments work with

the floating-point arithmetic [1] to ease and speedup the prototyping phase. However, when mapping the design to hardware, constraints on silicon area, power consumption, or throughput frequently force the implementer to convert this design to the more efficient fixed-point arithmetic [2]. This conversion is known to be a tedious and time consuming process [3] that may be split into two phases:

1. Range analysis: This phase allows to find the integer wordlength of each variable in the design. In a finite wordlength environment, minimizing the integer word length allows one to allocate more digits for the fractional part, thus obtaining more accuracy.
2. Precision analysis: In this phase, the number of bits to allocate to the fractional part is decided. This phase must take into account the precision requirements of the application.

Over the last years, authors have suggested different strategies to tackle these conversion phases. These contributions fit into two categories:

1. Simulation based strategies [4, 5]: The information that allows to estimate the required range and precision are inferred from intensive simulations carried out using an accurate arithmetic, typically floating-point arithmetic.
2. Analytic strategies [6, 7]: The information is obtained using formal methods such as interval arithmetic, affine arithmetic, and norm computation for digital filters. The precision analysis relies on optimization techniques.

In this work, we focus on the automated design of fixed-point programs for linear algebra basic blocks, like matrix multiplication and inversion. Although many work on this topic exist, to our knowledge, this work is the first one where an analytic approach based on interval arithmetic is used for large problems, in order to bound the range of the variables in the design and to give strict bounds on the rounding errors. Indeed [4] deals with the transformation from floating-point to fixed-point of matrix decomposition algorithms for DSPs and [8] with the implementation of matrix factorization algorithms for the particular C6x VLIW processor, while [9] and [10] discuss matrix inversion for the C64x+ DSP core and FPGAs, respectively. For the matrix multiplication, [11] presents a hardware implementation of a matrix multiplier optimized for a Virtex4 FPGA, which mainly relies on a large matrix-vector block to handle large matrices. Yet another FPGA architecture is presented in [12], that uses parallel DSP units and multiplies sub-matrices, whose size has been optimized so as to fully exploit the resources of the underlying architecture. In [13] a delay and resource efficient methodology is introduced to implement a FPGA architecture for matrix multiplication in integer/fixed-point arithmetic. However, in all these works, simulation based approaches are mainly used to decide the integer and fractional wordlengths, in order to treat small size problem without any guarantee on the accuracy of the result. For example, the methodology presented in [10] enables to treat inversion of size-8 matrices, while [4] is able to handle matrices of size up to 35, but without providing any certificate on the error bounds.

In this article, we present a framework for certified fixed-point code synthesis. Through this framework, our aim is threefold:

1. to shorten the development time by providing tools that generate efficient fixed-point code,
2. to reassure the users by certifying the numerical properties of the generated codes,
3. to propose a tool that scales up, i.e. able to synthesize code for large problems such as inverting a 80×80 matrix in fixed-point arithmetic.

This framework includes an arithmetic model, the CGPE[1] library that synthesizes code for fine-grained expressions (such as dot-products, sums, polynomial evaluations, ...), and the high level FPLA[2] tool to generate code for linear algebra basic blocks (such as matrix multiplications, Cholesky decompositions, and triangular matrix inversions).

We intend this framework to be a proof of concept that the development time of fixed-point codes can be dramatically reduced and that their numerical quality can be asserted. Furthermore, we use the framework to show that generating codes for matrix multiplication involves accuracy versus code size trade-offs and that generating codes for matrix inversion involves trade-offs between obtaining sharp error bounds and risking to have run-time overflows. For both cases, we describe, implement, and experiment with several algorithms to find trade-offs between the conflicting goals.

This article is organized as follows. Section 2 introduces background material concerning the fixed-point numbers followed by our arithmetic model. Section 3 is dedicated to matrix multiplication and to the trade-offs between code size and accuracy. Several techniques for matrix inversion are then introduced in Section 4, before a conclusion in Section 5.

2. Background on certifying fixed-point computations

In this section, we start by a presentation of our fixed-point arithmetic model. Then, we explicit a model based on the propagation of intervals to bound the range of fixed-point variables and the rounding errors entailed by fixed-point computations.

2.1. Fixed-point arithmetic model

Fixed-point number and variable. Unlike floating-point numbers, fixed-point numbers do not store any information about their exponent. Indeed, the exponent is implicit and known only to the programmer. And from the computer's perspective, a fixed-point number is similar to a computer integer. The machine integer that encodes the fixed-point number, denoted by X, is often a k-bit signed integer in two's complement notation. On the other hand, the

[1] See `http://cgpe.gforge.inria.fr/` and [14] for details.
[2] See `http://perso.univ-perp.fr/mohamedamine.najahi/fpla/` and [15, § 6] for details.

implicit information on the exponent is given by the scaling factor denoted by $f \in \mathbb{Z}$. Together, these integers define the fixed-point value x as:

$$x = X \cdot 2^{-f}.$$

In the sequel of this article, we shall denote $\mathbf{Q}_{i.f}$ the *format* of a given fixed-point variable v represented using a k-bit integer associated with a scaling factor f, with $k = i + f$. Here i and f denote the number of bits in the *integer* and *fraction* parts of v, respectively, while k represents its *wordlength*. Hence v is such that:

$$v \in \{V \cdot 2^{-f}\} \quad \text{with} \quad V \in \mathbb{Z} \cap [-2^{k-1}, 2^{k-1} - 1]. \tag{1}$$

Set of fixed-point variables. In practice, a fixed-point variable v may lie in a smaller range than the one in Equation (1). For instance, if $V \in \mathbb{Z} \cap [-2^{k-1} + 2^{k-2}, 2^{k-1} - 2^{k-2}]$ in Equation (1), then v is still in the $\mathbf{Q}_{i.f}$ format but with additional constraints on the runtime values it can take. For this reason, we shall denote by $\mathbb{F}\text{ix}$ the *set of fixed-point variables*, where each element has a fixed-point format and an interval that narrows its runtime values.

2.2. Interval arithmetic based error model

An arithmetic model describes the semantics of operations such as addition and multiplication, and gives the mean to estimate their accuracy. In the absence of standards to govern fixed-point implementations, it is customary for every research work on fixed-point arithmetic to present its underlying arithmetic model. Examples of such models include Fang *et al.*'s work [7] which is based on affine arithmetic and Didier *et al.*'s [16] which uses a probabilistic estimation of the propagation of noise.

Our arithmetic model is based on interval arithmetic and was influenced by typical DSP architectures. It keeps track of the three following intervals for each fixed-point variable v:

1. $\mathbf{Val}(v)$ enclosing the values of v computed at run-time with finite precision,
2. $\mathbf{Math}(v)$ enclosing the values of v had the computations been carried using infinite precision,
3. $\mathbf{Err}(v)$ enclosing the rounding errors occurring while computing v,

such that:

$$\mathbf{Err}(v) = \mathbf{Math}(v) - \mathbf{Val}(v).$$

Notice that the computations that involve $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ are carried using the interval arithmetic [17]. And thanks to the formula above, keeping track of $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ suffices to deduce $\mathbf{Math}(v)$.

Next, for each operator $\diamond \in \mathcal{O} = \{+, -, \times, \ll, \gg, \sqrt{}, /\}$, we shall explicit the basic rules to compute $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ from $\mathbf{Val}(v_1), \mathbf{Val}(v_2), \mathbf{Err}(v_1)$ and $\mathbf{Err}(v_2)$, where v_1 is the first operand of $\diamond$ and v_2 the second operand if $\diamond$ is binary. To show how $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ are computed, let us define the fixed-point formats of v, v_1, and v_2 to be $\mathbf{Q}_{i.f}$, $\mathbf{Q}_{i_1.f_1}$, and $\mathbf{Q}_{i_2.f_2}$, respectively. When $v = v_1 \diamond v_2$ with $\diamond \in \{+, -, \times\}$, we have:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) \diamond \mathbf{Val}(v_2) - \mathbf{Err}_{\diamond}.$$

2.2.1. Addition and subtraction

In our context, addition and subtraction are error-free. Hence for $\diamond \in \{+, -\}$ we have:

$$\mathbf{Err}(v) = \mathbf{Err}(v_1) \diamond \mathbf{Err}(v_2) \quad \text{and,}$$

$$i = \max(i_1, i_2) + 1 \quad \text{and} \quad f = \max(f_1, f_2).$$

Note that the most significant bit is here to prevent overflow issues. In absence of overflow, we can reduce the format of the result to $i = \max(i_1, i_2)$, which is actually the case considered in our experiments.

2.2.2. Multiplication

If $\diamond$ is a multiplication, we have:

$$\begin{aligned} \mathbf{Err}(v) &= \mathbf{Err}_\times + \mathbf{Err}(v_1) \cdot \mathbf{Err}(v_2) \\ &+ \mathbf{Err}(v_1) \cdot \mathbf{Val}(v_2) + \mathbf{Val}(v_1) \cdot \mathbf{Err}(v_2), \end{aligned}$$

where $\mathbf{Err}_\times$ is the error entailed by the multiplication itself. Remark that exact fixed-point multiplication results in a number having a fraction of $f_1 + f_2$ bits. If the fraction part f of the output is such that $f \geq f_1 + f_2$, then we have an exact multiplication, and $\mathbf{Err}_\times = [0, 0]$. However, this is costly and most DSP processors provide truncated multiplication operators. In this case, $\mathbf{Err}_\times$ accounts for the truncation of the exact result of the multiplication to fit in a smaller format with $f < f_1 + f_2$ fraction bits. Consequently it is defined as:

$$\mathbf{Err}_\times = [0, 2^{-f} - 2^{-(f_1+f_2)}].$$

In this work, we consider a 32×32 multiplier that returns the 32 most significant bits of the exact result. In this case,

$$i = i_1 + i_2 \quad \text{and} \quad f = 32 - i.$$

2.2.3. Left and right shift

If $\diamond \in \{\texttt{<<}, \texttt{>>}\}$, we have:

$$\mathbf{Err}(v) = \mathbf{Err}(v_1) + \mathbf{Err}_\diamond.$$

Left shifts of s bits entail no error but only a possible overflow: $\mathbf{Err}_{\texttt{<<}} = [0, 0]$ and $(i, f) = (i_1 - s, f_1 + s)$. However right shifts of s bits may be followed by a truncation to fit the result in a smaller format with $f < f_1$ fraction bits. Thus, we have:

$$(i, f) = (i_1 + s, f_1 - s) \quad \text{and} \quad \mathbf{Err}_{\texttt{>>}} = [0, 2^{-f_1+s} - 2^{-f_1}].$$

2.2.4. Square root
Assuming $v_1 \geq 0$, for $v = \sqrt{v_1}$, we have:

$$\mathbf{Val}(v) = \sqrt{\mathbf{Val}(v_1)} - \mathbf{Err}_{\sqrt{}}$$

since the computed value is truncated, while $\mathbf{Err}(v)$ is:

$$\mathbf{Err}(v) = \sqrt{\mathbf{Math}(v_1)} - \sqrt{\mathbf{Val}(v_1)} + \mathbf{Err}_{\sqrt{}},$$

where $\mathbf{Err}_{\sqrt{}}$ is the error entailed by the square root operation itself. The error term is given by the following formula:

$$\begin{aligned} \mathbf{Err}(v) &= \sqrt{\mathbf{Val}(v_1) + \mathbf{Err}(v_1)} - \sqrt{\mathbf{Val}(v_1)} + \mathbf{Err}_{\sqrt{}} \\ &= \sqrt{\mathbf{Val}(v_1)} \cdot \left(\sqrt{1 + \frac{\mathbf{Err}(v_1)}{\mathbf{Val}(v_1)}} - 1 \right) + \mathbf{Err}_{\sqrt{}}. \end{aligned} \tag{2}$$

The last factorization is used to remedy the *interval dependency* phenomenon inherent to interval arithmetic [18].

Notice that this formula does not yield tight error bounds as soon as $\mathbf{Val}(v_1)$ smallest elements are of the same order of magnitude than $\mathbf{Err}(v_1)$. To overcome this issue, we may use the subadditivity property of the square root function, which holds as long as x and $x + y$ are both positive:

$$\sqrt{x} - \sqrt{|y|} \leq \sqrt{x + y} \leq \sqrt{x} + \sqrt{|y|}.$$

Hence we deduce the following bounds on $\mathbf{Err}(v)$:

$$\mathbf{Err}_{\sqrt{}} - \sqrt{|\mathbf{Err}(v_1)|} \leq \mathbf{Err}(v) \leq \mathbf{Err}_{\sqrt{}} + \sqrt{|\mathbf{Err}(v_1)|}. \tag{3}$$

In practice, we compute the intersection of the enclosures (2) and (3).

As for $\mathbf{Err}_{\sqrt{}}$, it depends on the algorithm used to compute the square root. To explicit such an algorithm, let us remember that we have $v_1 = V_1 \cdot 2^{-f_1}$ which one can rewrite as

$$v_1 = 2^{\eta} \cdot V_1 \cdot 2^{-(f_1+\eta)}$$

with the integer η being a parameter of the algorithm chosen at synthesis-time such as $f_1 + \eta$ is even. Using this scaling factor, it follows that

$$\sqrt{v_1} = \sqrt{2^{\eta} \cdot V_1} \cdot 2^{\frac{-(f_1+\eta)}{2}}. \tag{4}$$

An algorithm that exploits (4) shifts the integer representation V_1 of v_1 by η bits to the left and computes its integer square root. The result of this algorithm is a fixed-point variable with $(f_1 + \eta)/2$ bits of fraction part. Hence using this approach, we conclude that

$$i = \lceil i_1/2 \rceil, \quad f = \frac{f_1 + \eta}{2}, \quad \text{and} \quad \sqrt{v_1} = \left\lfloor \sqrt{2^{\eta} \cdot V_1} \right\rfloor \cdot 2^{\frac{-(f_1+\eta)}{2}},$$

where $\left\lfloor \sqrt{2^{\eta} \cdot V_1} \right\rfloor$ is computed using an integer square root operation. It is clear now that with such an algorithm, we obtain the following bound on $\mathbf{Err}_{\surd}$:

$$\mathbf{Err}_{\surd} = \left[0, 2^{-\frac{(f_1+\eta)}{2}}\right].$$

Notice that it would not make sense to choose $\eta < 0$, since it would result in an increase of $\mathbf{Err}_{\surd}$. Hence in the following of the section, we assume $\eta \geq 0$.

Notice that this integer square root operator may be implemented in hardware or in software using multiple techniques such as digit-recurrence, and Newton-Raphson or Goldschmidt iteration [19, 20]. In this implementation the parameter η must be carefully chosen, to ensure that $f_1 + \eta$ is even, and that the wordlength of the result is at most k, otherwise an overflow may occur.

2.2.5. Division

Let $v = v_1/v_2$, when the quotient is defined, *i.e.* when $v_2 \neq 0$, that is, $0 \notin \mathbf{Val}(v_2)$, we have:

$$\mathbf{Val}(v) = \frac{\mathbf{Val}(v_1)}{\mathbf{Val}(v_2)} - \mathbf{Err}_{/}$$

while $\mathbf{Err}(v)$ is defined as:

$$\mathbf{Err}(v) = \frac{\mathbf{Math}(v_1)}{\mathbf{Math}(v_2)} - \frac{\mathbf{Val}(v_1)}{\mathbf{Val}(v_2)} + \mathbf{Err}_{/},$$

where $\mathbf{Err}_{/}$ is the error entailed by the division itself. It follows that the error term is defined as:

$$\mathbf{Err}(v) = \frac{\mathbf{Val}(v_2) \cdot \mathbf{Err}(v_1) - \mathbf{Val}(v_1) \cdot \mathbf{Err}(v_2)}{\mathbf{Val}(v_2) \cdot (\mathbf{Val}(v_2) + \mathbf{Err}(v_2))} + \mathbf{Err}_{/}.$$

From a theoretical point of view, the quotient v_1/v_2 when defined must be a fixed-point variable in the format $\mathbf{Q}_{i.f}$ with

$$i = i_1 + f_2 \quad \text{and} \quad f = f_1 + i_2$$

since:

1. The largest possible dividend is -2^{i_1-1} while the smallest divisor is 2^{-f_2}. Thus the quotient could be as large as $-2^{i_1+f_2-1}$.
2. As for the opposite case, the smallest dividend is 2^{-f_1} while the largest divisor is -2^{i_2-1}. To be precise, the fractional part must be of size f_1+i_2.

Remark that a special care must be taken when $0 \in \mathbf{Val}(v_2)$ to avoid division by zero. In this case, we first compute both error and value bounds twice, using the two intervals $\underline{\mathbf{Val}(v_2)}$ and $\overline{\mathbf{Val}(v_2)}$, such that:

$$\underline{\mathbf{Val}(v_2)} \cup \overline{\mathbf{Val}(v_2)} = \mathbf{Val}(v_2) \setminus \{0\}.$$

Then we compute the union of the resulting intervals.

As for the square root, the fixed-point format of v depends on the algorithm implemented. In the following, we suggest an algorithm to perform fixed-point division together with a piece of C code that implements it and we exhibit its error bound. To do this, let us remember that we have $v_1 = V_1 \cdot 2^{-f_1}$ and $v_2 = V_2 \cdot 2^{-f_2}$ and start from the following rewriting:

$$\frac{v_1}{v_2} = \frac{V_1 \cdot 2^{-f_1}}{V_2 \cdot 2^{-f_2}} = \frac{V_1 \cdot 2^{\eta}}{V_2} \cdot 2^{-(f_1 - f_2 + \eta)}.$$

An implementation based on this formula would compute

$$\frac{v_1}{v_2} = \texttt{trunc}\left(\frac{V_1 \cdot 2^{\eta}}{V_2}\right) \cdot 2^{-(f_1 - f_2 + \eta)}$$

which results in a variable having the fractional part

$$f = f_1 - f_2 + \eta.$$

In our context, since we consider that all the variables have the same wordlength, and in particular $i + f = i_1 + f_1$, it follows that the integer part of the result is:

$$i = i_1 + f_2 - \eta. \tag{5}$$

Then we deduce that the error $\mathbf{Err}_{/}$ is as follows:

$$\mathbf{Err}_{/} = [-2^{-(f_1 - f_2 + \eta)}, 2^{-(f_1 - f_2 + \eta)}].$$

Remark that even when v_1 and v_2 have the same format, $\mathbf{Err}_{/} = [-2^{-\eta}, 2^{-\eta}]$ remains tight as long as η is large enough. Also as for square root, it would not make sense to choose $\eta < 0$, since it would result in an increase of $\mathbf{Err}_{/}$.

Notice that an implementation of this method may use the C standard integer division. If this option is not available or is too costly, this operation may also be implemented in hardware or in software using digit-recurrence, and Newton-Raphson or Goldschmidt iteration. And again, the parameter η must be chosen carefully, since it greatly influences the result by impacting its integer part i. Indeed picking a large η leads to a smaller value i than the theoretical one and it minimizes the error bound and ensures more accuracy on the result. However, by doing so, we suppose that the result is not large enough. More precisely, this means that the largest values in magnitude eventually taken by the result are ignored and discarded. This approach is equivalent to discarding the smallest values eventually taken by the variable v_2 in $\mathbf{Val}(v_2)$, that is, the values around 0 in $\mathbf{Val}(v_2)$.

3. Code size versus accuracy trade-offs: matrix multiplication

As shown further in Section 4.1, the Cholesky decomposition based matrix inversion requires to be able to multiply the inverse of triangular matrices. Moreover, in practice, the matrix multiplication basic block can also be used for applying

linear transformations to input data derived from signal or image processing, or for solving linear systems by iterative methods [21].

In floating-point arithmetic, while asymptotically fast algorithms such as Strassen's [22] are usually used in highly optimized libraries, like the BLAS [23], this process can be simply implemented in a straightforward way, based on the following direct definition:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j},$$

where A, B, and C are three matrices. In fixed-point arithmetic, works that deal with matrix multiplications are scarce, and the existing ones mainly rely on Sung's technique [24] to convert floating-point designs into fixed-point, not well-adapted for large problems as shown in Section 1. For example, the size-64 matrix multiplication, which is considered as a large problem in fixed-point arithmetic, involves 8192 input variables. In this case simulation approaches do not scale and only an analytic approach is practical. To our knowledge, our work is the first to attempt to apply certified fixed-point techniques to such large problems [25]. And to provide small and accurate codes to multiply matrices in fixed-point arithmetic, trade-offs between code size and accuracy are thus proposed.

Section 3.1 gives a statement of the problem of matrix multiplication in fixed-point arithmetic, while this kind of straightforward algorithms is investigated in Section 3.2. Then, Section 3.3 suggests a strategy to find trade-offs between these straightforward algorithms. This strategy is implemented and experimental data are presented to show its effectiveness on a set of benchmarks in Section 3.4.

3.1. Problem statement of matrix multiplication

Let A and B be two matrices of fixed-point variables of size $m \times n$ and $n \times p$, respectively:

$$A \in \mathbb{F}\mathrm{ix}^{m \times n} \quad \text{and} \quad B \in \mathbb{F}\mathrm{ix}^{n \times p}.$$

Here and hereafter, we denote by $A_{i,:}$ and $A_{:,j}$ the i^{th} row and j^{th} column of A, respectively, and $A_{i,j}$ the element of the i^{th} row and j^{th} column of A.

Our goal is to generate fixed-point code to multiply A and B. And, since A and B are matrices of fixed-point variables, the generated code should be able to multiply at run-time any matrices A' and B', where A' and B' are two matrices that belong to A and B, that is, where the fixed-point numbers $A'_{i,k}$ and $B'_{k,j}$ belong to the fixed-point variables $A_{i,k}$ and $B_{k,j}$, respectively. This consists in writing a program for computing $C = A \cdot B$, where $C \in \mathbb{F}\mathrm{ix}^{m \times p}$. Therefore, $\forall i, j \in \{1, \cdots, m\} \times \{1, \cdots, p\}$, we have:

$$C_{i,j} = A_{i,:} \cdot B_{:,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}, \tag{6}$$

3.2. Straightforward approaches for the synthesis of matrix multiplication codes

Here we discuss two straightforward approaches to solve the problem of code synthesis for matrix multiplication.

3.2.1. Accurate and compact approaches

Following Equation (6), and assuming that a routine dedicated to the generation of code for dot-products is available, a first straightforward approach may consist in invoking this routine as many times as required to generate code for each dot-product. As explained in Section 1, this routine is provided by the CGPE software tool, and it is denoted by DPSynthesis in the sequel of this section. Algorithm 1 below implements this approach.

Algorithm 1 Accurate algorithm.

Input:

$A \in \mathbb{F}\mathrm{ix}^{m\times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n\times p}$

Output:

Code to compute $A' \cdot B'$

Algorithm:

1: **for** $1 \leq i \leq m$ **do**
2: **for** $1 \leq j \leq p$ **do**
3: DPSynthesis$(A_{i,:}, B_{:,j})$
4: **end for**
5: **end for**

Algorithm 2 Compact algorithm.

Input:

$A \in \mathbb{F}\mathrm{ix}^{m\times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n\times p}$

Output:

Code to compute $A \cdot B$

Algorithm:

1: $\mathcal{U} \leftarrow A_{1,:} \cup A_{2,:} \cup \cdots \cup A_{m,:}$, with $\mathcal{U} \in \mathbb{F}\mathrm{ix}^{1\times n}$
2: $\mathcal{V} \leftarrow B_{:,1} \cup B_{:,2} \cup \cdots \cup B_{:,p}$, with $\mathcal{V} \in \mathbb{F}\mathrm{ix}^{n\times 1}$
3: DPSynthesis$(\mathcal{U}, \mathcal{V})$

Notice that Algorithm 1 issues $m \times p$ queries to the DPSynthesis routine. And at runtime, only one call to each generated code will be issued, for a total of $m \times p$ calls.

To significantly reduce the number of dot-product codes generated, denoted by DPCodes in the following, some of them could be factored to evaluate more than one dot-product at run-time. Algorithm 2 pushes this idea to the limits by merging element by element the matrices A and B into a unique row $\mathcal{U}$ and column $\mathcal{V}$, respectively. Here *merging* two fixed-point matrices means computing their union. Particularly, let $(x, y) \in \mathbb{F}\mathrm{ix}^2$ be two fixed-point variables. In order to compute their union, we must determine $z \in \mathbb{F}\mathrm{ix}$ such that $\mathbf{Q}_{a_z.b_z}$ and $\mathcal{I}_Z$ are, respectively, the smallest format and enclosure that accommodate the values of x and y without overflow. (See [15, § 4.2.1] for details on the algorithm used to compute z.) This second approach issues a unique call to the DPSynthesis routine, while at run-time, $m \times p$ calls to this code are still needed to evaluate the matrix product.

3.2.2. Illustration example

Let us now illustrate the differences between these two algorithms by considering the code generation for the product of the following two fixed-point matrices:

$$A = \begin{pmatrix} [-1000, 1000] & [-3000, 3000] \\ [-1, 1] & [-1, 1] \end{pmatrix} \text{ and } B = \begin{pmatrix} [-2000, 2000] & [-2, 2] \\ [-4000, 4000] & [-10, 10] \end{pmatrix},$$

	Algorithm 1				Algorithm 2
Dot-product	$A_{1,:} \cdot B_{:,1}$	$A_{1,:} \cdot B_{:,2}$	$A_{2,:} \cdot B_{:,1}$	$A_{2,:} \cdot B_{:,2}$	All
Evaluated using	$\text{DPCode}_{1,1}$	$\text{DPCode}_{1,2}$	$\text{DPCode}_{2,1}$	$\text{DPCode}_{2,2}$	$\text{DPCode}_{\mathcal{U},\mathcal{V}}$
Output format	$Q_{26,6}$	$Q_{18,14}$	$Q_{15,17}$	$Q_{7,25}$	$Q_{26,6}$
Certified error	$\approx 2^{-5}$	$\approx 2^{-14}$	$\approx 2^{-16}$	$\approx 2^{-24}$	$\approx 2^{-5}$
Maximum error	$\approx 2^{-5}$				$\approx 2^{-5}$
Average error	$\approx 2^{-7}$				$\approx 2^{-5}$

Table 1: Numerical properties of the codes generated by Algorithms 1 and 2 for $A \cdot B$.

where $A_{1,1}$ and $B_{1,1}$ are in the format $\mathbf{Q}_{11.21}$, $A_{1,2}$ in $\mathbf{Q}_{12.20}$, $A_{2,1}$, $A_{2,2}$, $B_{2,1}$ in $\mathbf{Q}_{2.30}$, $B_{1,2}$ in $\mathbf{Q}_{3.29}$, and $B_{2,2}$ in $\mathbf{Q}_{5.27}$. Algorithm 1 produces 4 distinct codes, denoted by $\text{DPCode}_{1,1}$, $\text{DPCode}_{1,2}$, $\text{DPCode}_{2,1}$, and $\text{DPCode}_{2,2}$. On the other hand, Algorithm 2 first computes $\mathcal{U} = A_{1,:} \cup A_{2,:}$ and $\mathcal{V} = B_{:,1} \cup B_{:,2}$ as follows:

$$\mathcal{U} = ([-1000, 1000][-3000, 3000]) \text{ and } V = \begin{pmatrix} [-2000, 2000] \\ [-4000, 4000] \end{pmatrix}.$$

Then, $\text{DPCode}_{\mathcal{U},\mathcal{V}}$ is generated that evaluates the dot-product of $\mathcal{U}$ and $\mathcal{V}$. Table 1 summarizes the properties of the codes produced by Algorithms 1 and 2 on this example. On one hand, Algorithm 1 produces codes optimized for the range of their entries: it is clearly superior in terms of accuracy since a dedicated code evaluates each run-time dot-product. On the other hand, as expected, Algorithm 2 produces far less code: it is is optimal in terms of code size since a unique DPCode is generated, but remains a worst-case in terms of accuracy.

3.3. Dynamic closest pair algorithm for code size vs. accuracy trade-offs

Fixed-point arithmetic is primarily used in embedded systems where the execution environment is usually constrained. Hence even tools that produce codes with guaranteed error bounds would be useless if the generated code size is excessively large. In this section, we go further than Algorithms 1 and 2, and explore the possible means to achieve trade-offs between the two conflicting goals, through our new appraoch called *Dynamic Closest Pair algorithm*.

3.3.1. How to achieve trade-offs?

Once the accuracy and code size parameters are set, the programmer tries Algorithms 1 and 2.

1. When Algorithm 1 is not accurate enough. Since Algorithm 1 produces the most accurate codes, a potential solution is to adapt the fixed-point computation word-lengths to reach the required accuracy, as in [26].
2. When Algorithm 2 does not satisfy the code size constraint. Again, since this algorithm produces the most compact code, other solutions must be considered such as adding more hardware resources.

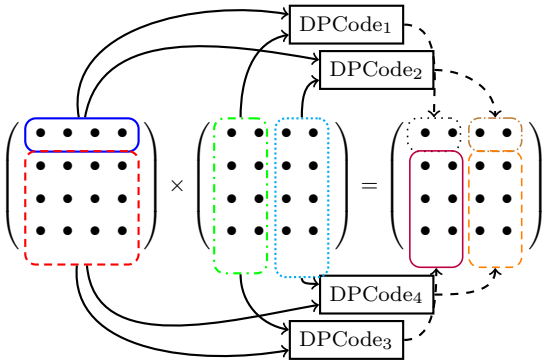


Figure 1: One merging strategy on a 4×4 matrix multiplication.

Finally, the only uncertainty that remains is when Algorithm 1 satisfies the accuracy constraint but has a large code size while Algorithm 2 satisfies the code size bound but is not accurate enough. This case appeals for code size versus accuracy trade-offs.

Since $m \times p$ dot-product calls are needed at runtime, reducing the number of DPCodes requires to factor some of them so that they would evaluate more than one run-time dot-product. This amounts to merging certain rows and/or columns of the input matrices together. Obviously, it is useless to go as far as compressing the left and right matrices into one row and column, respectively, since this corresponds to Algorithm 2. Our idea is illustrated by Figure 1 on a 4×4 matrix multiplication. In this example, the first matrix is compressed into a 2×4 matrix while the second matrix is compressed into a 4×2 matrix, as shown by the differently colored and shaped blocks. In this case, the number of required codes is reduced from 16 to 4. For example, DPCode_1 has been particularly optimized for the computation of $A_{1,:} \cdot B_{:,1}$ and $A_{1,:} \cdot B_{:,2}$, and will be used exclusively for these at run-time.

3.3.2. Combinatorial aspect of the merging strategy

Consider the two sets of vectors:

$$\mathcal{S}_A = \{A_{1,:}, \cdots, A_{m,:}\} \quad \text{and} \quad \mathcal{S}_B = \{B_{:,1}, \cdots, B_{:,p}\},$$

associated to the input matrices:

$$A \in \mathbb{F}\text{ix}^{m \times n} \quad \text{and} \quad B \in \mathbb{F}\text{ix}^{n \times p}.$$

In our case, the problem of finding an interesting code size versus accuracy trade-off reduces to finding partitions of the sets $\mathcal{S}_A$ and $\mathcal{S}_B$ into $k_A \leq m$ and $k_B \leq p$ subsets, respectively, such that both of the following conditions hold:

1. the code size bound σ is satisfied, that is:

$$(4n - 1) \cdot k_A \cdot k_B < \sigma, \tag{7}$$

where $4n-1$ is a worst case bound on the number of elementary operations (additions, multiplications, and shifts) needed to evaluate a size-n dot-product in fixed-pont arithemtic [25, § 3.2],

2. and the error bound ϵ is guaranteed, that is:

$$\epsilon_{\text{matrix}} < \epsilon, \tag{8}$$

where ϵ_{matrix} is either the minimal, the maximal, or the average computation error depending on the certification level required by the user.

Note that, given the partitions of $\mathcal{S}_A$ and $\mathcal{S}_B$, the first condition is easy to check. However in order to guarantee the error condition, we must synthesize the DPCodes and deduce their error bounds. This can be done using CGPE.

A benefit of formulating the refactoring strategy in terms of partitioning is the ability to give an upper bound on the number of possible dot-product mergings. Indeed, given a non-empty set $\mathcal{S}$ of k vectors, the number of different ways to partition $\mathcal{S}$ into $k' \leq k$ non-empty subsets of vectors is given by the *Stirling* number[3] of the second kind $\left\{ {k \atop k'} \right\}$, defined as follows:

$$\left\{ {k \atop k'} \right\} = \frac{1}{k'!} \sum_{j=0}^{k'} (-1)^{k'-j} \frac{k'!}{j!(k'-j)!} j^k.$$

However, k' is *a priori* unknown and can be $\in \{1, \cdots, k\}$. The total number of possible partitions of a set of k vectors is therefore given by the following sum, commonly referred to as the *Bell* number:[4]

$$B(k) = \sum_{k'=1}^{k} \left\{ {k \atop k'} \right\}.$$

Finally, in our case, the total number of partitionings is defined as follows:

$$\mathcal{P}(m,p) = B(m) \cdot B(p) - 2, \tag{9}$$

where $m \times p$ is the size of the resulting matrix. Notice that we exclude two partitions:

1. The partition of $\mathcal{S}_A$ and $\mathcal{S}_B$ into, respectively, m and p subsets which correspond to putting one and only one vector in each subset. This is the partitioning that leads to Algorithm 1.
2. The partition of $\mathcal{S}_A$ and $\mathcal{S}_B$ into one subset each. This partitioning leads to Algorithm 2.

Table 2 gives some values of $\mathcal{P}$ of Equation (9). Since this number is large, even for small matrix sizes, heuristics will be necessary to tackle this problem.

[3] See `http://oeis.org/A008277`.
[4] See `http://oeis.org/A000110`.

(m, p)	$(5,5)$	$(6,6)$	$(10,10)$	$(16,16)$	$(25,25)$	$(64,64)$
Number of algorithms $\mathcal{P}$	2704	41 209	$\approx 2^{34}$	$\approx 2^{66}$	$\approx 2^{124}$	$\approx 2^{433}$

Table 2: Some values of $\mathcal{P}$ for the multiplication of square matrices.

3.3.3. Dynamic Closest Pair Algorithm

A component-wise merging of two vectors $\mathcal{U}$ and $\mathcal{V}$ of fixed-point variables yields a vector whose ranges are larger than those of $\mathcal{U}$ and $\mathcal{V}$. This eventually leads to a degradation of the accuracy if the resulting vector is used to generate some DPCodes. In the extreme, this is illustrated by Algorithm 2 in Section 3.2.1. Therefore the underlying idea of our approach is that of putting together, in the same subset, row or column vectors that are close according to a given distance or criterion. Hence we ensure a reduction in code size while maintaining tight fixed-point formats, and thus guaranteeing a tight error bound.

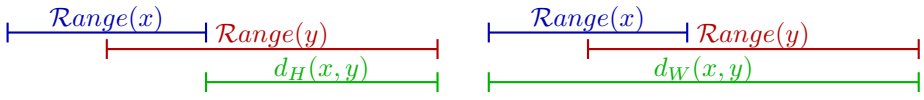
Many metrics can be used to compute the distance between two vectors. Below, we cite two mathematically rigorous distances that are suitable for fixed-point arithmetic: the Hausdorff distance and the fixed-point distance. However, as our method does not use the mathematical properties of distances, any criterion that may discriminate between pairs of vectors of fixed-point variables may be used, like the width criterion introduced below.

Hausdorff distance. The range of a fixed-point variable corresponds to a rational discrete interval. It follows that the Hausdorff distance [27], widely used as a metric in interval arithmetic, can be applied to fixed-point variables. Given two fixed-point variables x and y and their ranges $\mathcal{R}\text{ange}(x) = \left[\underline{r_x}, \overline{r_x}\right]$ and $\mathcal{R}\text{ange}(y) = \left[\underline{r_y}, \overline{r_y}\right]$, this distance $d_H(x, y)$ is defined as follows:

$$d_H : \mathbb{F}\text{ix} \times \mathbb{F}\text{ix} \to \mathbb{R}^+$$
$$d_H\left(x, y\right) = \max\left\{\left|\underline{r_x} - \underline{r_y}\right|, \left|\overline{r_x} - \overline{r_y}\right|\right\},$$

Roughly, this distance computes the maximum increase suffered by $\mathcal{R}\text{ange}(x)$ and $\mathcal{R}\text{ange}(y)$ when computing the union $x \cup y$, as illustrated on Figure 2(a).

This distance illustrates our heuristic: by trying to merge only vectors of variables that minimize the Hausdorff distance, we make sure that this merging minimally impacts their range.



(a) Hausdorff distance. (b) Width criterion.

Figure 2: Illustration of distances between two input variables $(x, y) \in \mathbb{F}\text{ix}^2$.

Fixed-point distance. Contrarily to the Hausdorff distance which reasons on the ranges defined by the fixed-point variables, the fixed-point distance uses only their fixed-point formats. As such, it is slightly faster to compute. Given two fixed-point variables x and y, this distance $d_F(x, y)$ is defined as follows:

$$d_F : \mathbb{F}\text{ix} \times \mathbb{F}\text{ix} \to \mathbb{N}$$
$$d_F\,(x, y) = |a_x - a_y|,$$

where $\mathbf{Q}_{a_x.b_x}$ and $\mathbf{Q}_{a_y.b_y}$ are the fixed-point formats of x and y, respectively. Analogously to Hausdorff distance, this distance computes the increase in the integer part suffered by x and y when computing their union $x \cup y$.

Width criterion. Let x, y, and z be three fixed-point variables such that $z = x \cup y$ where z is computed according to merging algorithm in [15, § 4.2.1]. Our third metric consists in considering the width of $\mathcal{R}\text{ange}(z) = \left[\underline{r_z}, \overline{r_z}\right]$ as illustrated on Figure 2(b). Formally, it is defined as follows:

$$d_W : \mathbb{F}\text{ix} \times \mathbb{F}\text{ix} \to \mathbb{R}^+$$
$$d_W\,(x, y) = \left(\overline{r_z} - \underline{r_z}\right).$$

Notice that although the metrics are introduced as functions of two fixed-point intervals, we generalized them to vectors of fixed-point variables by considering either the component-wise max or average value.

Given one of these metrics and a set $\mathcal{S}$ of vectors, it is straightforward to implement a `findClosestPair` routine that returns the pair of closest vectors in $\mathcal{S}$. A $\mathcal{O}(n^2)$ naive approach was implemented, that compare all the possible pairs of vectors. But, depending on the distance used, optimized implementations may rely on the well established *fast closest pair of points* algorithms [28], [29, §33]. Nevertheless, our contribution lies mainly in the design of Algorithm 3 which is based on a dynamic search of a code that satisfies both an accuracy bound $\mathcal{C}_1$ and a code size bound $\mathcal{C}_2$.

Here we assume that Algorithm 1 satisfies the accuracy bound $\mathcal{C}_1$, otherwise, no smaller code satisfying $\mathcal{C}_1$ could be found. Therefore, Algorithm 3 starts with two sets of m and p vectors, respectively, corresponding to the rows of A and the columns of B. As long as the bound $\mathcal{C}_1$ is satisfied, each step of the while loop merges together the closest pair of rows or columns, and thus decrements the total number of vectors by 1. At the end of Algorithm 3, if the size of the generated code satisfies the code size bound $\mathcal{C}_2$, a trade-off solution has been found. Otherwise, Algorithm 3 failed to find a code that satisfies both bounds $\mathcal{C}_1$ and $\mathcal{C}_2$. This algorithm was implemented in the FPLA tool.

3.4. Numerical experiments

In this section, we illustrate the efficiency of our heuristics, and the behaviour of Algorithm 3 as well as the impact of the distance and the matrix size through a set of numerical results.

Algorithm 3 Dynamic Closest Pair algorithm.

Input:

Two matrices $A \in \mathbb{F}\text{ix}^{m \times n}$ and $B \in \mathbb{F}\text{ix}^{n \times p}$
An accuracy bound $\mathcal{C}_1$ (ex. average error bound is $< \epsilon$)
A code size bound $\mathcal{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfied,
or no code otherwise

Algorithm:

$\mathcal{S}_A \leftarrow \{A_{1,:}, \ldots, A_{m,:}\}$
$\mathcal{S}_B \leftarrow \{B_{:,1}, \ldots, B_{:,p}\}$
while $\mathcal{C}_1$ is satisfied **do**
  $(u_A, v_A), d_A \leftarrow findClosestPair(\mathcal{S}_A, d)$
  $(u_B, v_B), d_B \leftarrow findClosestPair(\mathcal{S}_B, d)$
  if $d_A \leq d_B$ **then**
    $remove(u_A, v_A, \mathcal{S}_A)$
    $insert(u_A \cup v_A, \mathcal{S}_A)$
  else
    $remove(u_B, v_B, \mathcal{S}_B)$
    $insert(u_B \cup v_B, \mathcal{S}_B)$
  end if
  for $(A_i, B_j) \in \mathcal{S}_A \times \mathcal{S}_B$ **do**
    DPSynthesis(A_i, B_j)
  end for
end while
/* Revert the last merging step. */
/* Check the bound $\mathcal{C}_2$. */

3.4.1. Experimental environment

Experiments have been carried out with 32 bit fixed-point variables and using 3 structured and 1 unstructured benchmark. For structured benchmarks, the large coefficients distribution throughout the matrices follows different patterns. This is achieved through weight matrices, as shown in Table 3 where $W_{i,j}$ corresponds to the element of row i and column j of the considered weight matrix.

Name	$W_{i,j}$	Heat map
Center	$2^{\max(i,j,n-1-i,n-1-j)-\lfloor n/2 \rfloor}$	
Edges	$2^{\min(i,j,n-1-i,n-1-j)}$	
Rows / Columns	$2^{\lfloor i/2 \rfloor}$ $2^{\lfloor j/2 \rfloor}$	
Random	$2^{\text{rand}(0,\lfloor n/2 \rfloor-1)}$	

Table 3: Weight matrices considered for the benchmarks.

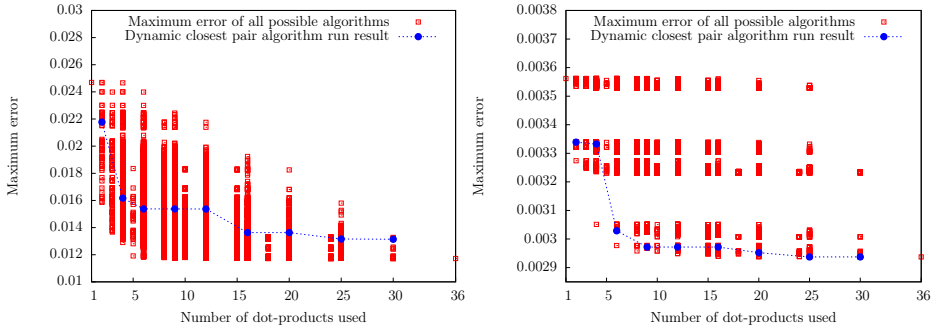
Notice, that the dynamic range defined as $max(W_{i,j})/min(W_{i,j})$ is the same for all benchmarks, and is equal to $2^{\lfloor n/2 \rfloor}$. The reason we did not directly use these matrices in our experiments is that the first three patterns correspond to structured matrices in the usual sense and that better algorithms to multiply structured matrices exist [30]. To obtain random matrices where the large coefficients are still distributed according to the pattern described by the weight matrices, we computed the Hadamard product of Table 3 matrices with normally distributed matrices generated using Matlab®'s `randn` function. Finally, notice that the matrices obtained this way have floating-point coefficients. In order to get fixed-point matrices, we first converted them to interval matrices by considering the radius 1 intervals centered at each coefficient. Next, the floating-point intervals are converted into fixed-point variables by considering the smallest fixed-point format that holds all the interval's values.

3.4.2. Efficiency of the distance based heuristic

As a first experiment, let us consider 2 of the benchmarks: *Center* and *Random* square matrices of size 6. For each, we build two matrices A and B, and observe the efficiency of our *closest pair* heuristic based approach by comparing the result of Algorithm 3 to all the possible codes. To do so, we compute all the possible row and column mergings: Equation (9) assures that there are 41 209 such mergings for size-6 matrices. For each of these, we synthesized the codes for computing $A \cdot B$, and determined the maximum and average errors. This exhaustive experiment took approximately 2h15min per benchmark on an Intel Core i7-870 desktop machine running at 2.93 GHz. Figures 3 and 4 show the maximum and average errors of the produced codes according to the number of DPCodes involved. Next, we ran our tool with Hausdorff's distance and with the accuracy bound $\mathcal{C}_1$ set to a large value so as to see the behavior of Algorithm 3 on all the intermediate steps. This took less than 10 seconds for each benchmark and corresponds to the dark blue dots in Figures 3 and 4. Notice on both sides the accurate algorithm which produces 36 DPCodes and the compact algorithm which produces only one DPCode.

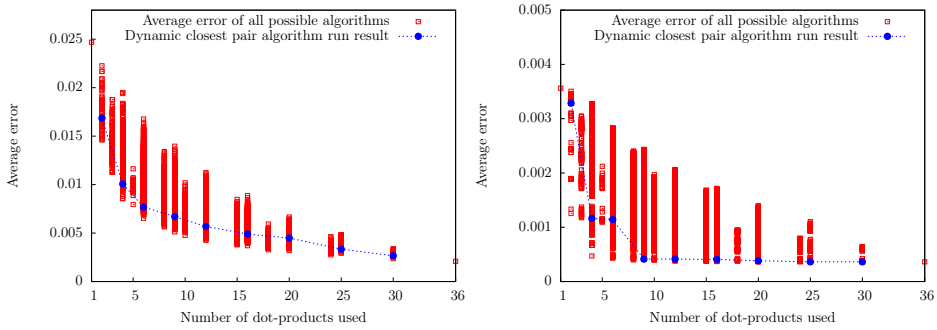
For the structured *Center* benchmark, Algorithm 3 behaves as expected. For both maximum and average error, it is able to drastically reduce the number of DPCodes without impacting the accuracy. Indeed, as shown in Figure 3(b), for a maximum error of $\approx 3 \cdot 10^{-3}$ which is close to the most accurate algorithm, our algorithm is able to reduce the number of DPCodes from 36 to 9. For average error in Figure 4, Algorithm 3 even finds a merging that produces 9 DPCodes and has almost the same accuracy as Algorithm 1 which is the most accurate.

For the *Random* benchmark, the behavior of Algorithm 3 is less predictable. Indeed, in this benchmark, the elements of high dynamic range are spread over the matrix and do not follow a particular pattern. In this case, it is less obvious for Algorithm 3 to find the best codes in terms of accuracy. Indeed, Algorithm 3 follows a greedy approach in making the local decision to merge two vectors. And, once it goes in a wrong branch of the result space, this may lead to a code having an average or maximum error slightly larger than the best case. This



(a) Random matrices. (b) Center matrices.

Figure 3: Maximum error according to the number of DPCodes.



(a) Random matrices. (b) Center matrices.

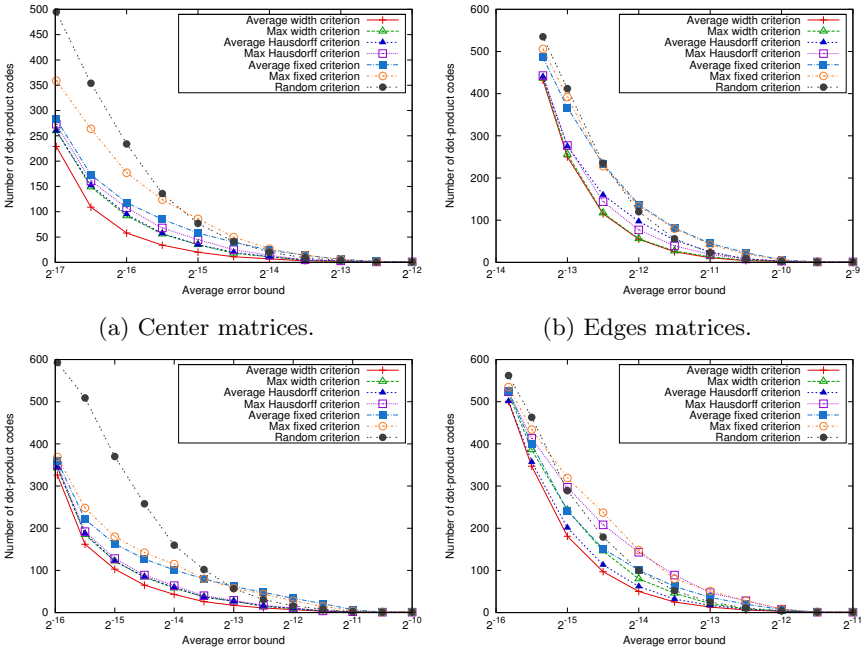
Figure 4: Average error versus the number of DPCodes.

can be observed on Figure 4(b): the first 6 steps produce code with very tight average error, but step 7 results in a code with an average error of $\approx 10^{-3}$ while the best code has an error of $\approx 5 \cdot 10^{-4}$. As a consequence, the following of the algorithm gives a code with an error of $\approx 3 \cdot 10^{-3}$ instead of $\approx 10^{-3}$ for the best case. The same phenomenon happens at step 1 Figure 3(a).

Despite this, these experiments show the interest of our approach. Indeed we may observe that, at each step, the heuristic merges together 2 rows of A or 2 columns of B to produce a code having in most cases an average error close to the best case. This is particularly the case on Figures 3(b) and 4(b) for *Center* benchmarks. Moreover, Algorithm 3 converges toward code having good numerical quality much faster than the exhaustive approach.

3.4.3. Impact of the metric on the trade-off strategy

In this second experiment, we consider 25×25 matrices. For each benchmark introduced above, 50 different matrix products are generated, and the results exhibited are computed as the average on these 50 products. To compare the



(a) Center matrices.

(b) Edges matrices.

(c) Rows/Columns matrices.

(d) Random matrices.

Figure 5: Number of dot-product codes generated by each algorithm for increasing average error bounds.

different distances, we consider the *average accuracy* bound: for each metric, we varied this bound and used Algorithm 3 to obtain the most compact codes that satisfy it. Here we ignored the code size bound $\mathcal{C}_2$ by setting it to a large enough value. Also, in order to show the efficiency of the *closest pair* strategy, we compare the codes generated using Algorithm 3 with those of an algorithm where the merging of rows and columns is carried out randomly. Figure 5 shows the results of running FPLA.

First notice that, as expected, large accuracy bounds yield the most compact codes. For instance, for all the benchmarks, no matter the distance used, if the target average accuracy is $> 2^{-9.5}$, one DPCode suffices to evaluate the matrix multiplication. This indeed amounts to using Algorithm 2. Also as expected and except for few values, when used with one of the distances above, our algorithm produces less DPCodes than with the random function as a distance. Using the average width criterion, our algorithm is by far better than the random algorithm and yields on the *Center* and *Rows/Columns* benchmarks a significant reduction in code size, as shown on Figures 5(a) and 5(c). For example, for the *Center* benchmark, when the average error bound is set to 2^{-16}, our algorithm satisfies it with only 58 DPCodes, while the random algorithm needs 234 DPCodes. This yields a code size reduction of up to 75%. Notice also that

globally, the *Center* benchmark is the most accurate. This is due to the fact that few *Rows/Columns* have a high dynamic range. On Figures 5(b) and 5(d), in the *Edges* as well as *Random* benchmarks, all of the rows and columns have a high dynamic range which explains in part why these benchmarks are less accurate than the *Center* benchmark. These experiments also suggest that average based distances yield tighter code than maximum based ones.

3.4.4. Impact of the matrix size

In this third experiment, we study the influence of the matrix sizes on the methodology presented above. To do so, we consider square matrices of the *Center* benchmark with sizes 8, 16, 32, and 64, where each element has been scaled so as these matrices have the same dynamic range. We run Algorithm 3 using the *average width* criterion as a metric with different *average error* bounds from 2^{-21} to 2^{-14}. Here the bound $\mathcal{C}_2$ has also been ignored. For each of these benchmarks, we determine the number of DPCodes used for each average error, as shown in Table 4 (where "–" means "no result has been found").

	Maximum error							
Matrix size	2^{-21}	2^{-20}	2^{-19}	2^{-18}	2^{-17}	2^{-16}	2^{-15}	2^{-14}
8	24	6	1	1	1	1	1	1
16	–	117	40	16	3	1	1	1
32	–	–	552	147	14	2	1	1
64	–	–	–	2303	931	225	48	1

Table 4: Number of DPCodes for various matrix sizes and error bounds.

This shows clearly that our method is extensible to large matrices, since it allows to reduce the size of the problem to be implemented, while maintaining a good numerical quality. For example, the 64×64 accurate matrix multiplication would require 4096 DPCodes. Using our heuristic, we produce a code with 2303 DPCodes having an average error bounded by 2^{-18}, that is, a reduction of about 45%. Remark that no code with average error bound of 2^{-19} is found, which means that even the accurate algorithm (Algorithm 1) has an error no tighter than 2^{-19}: we can conclude that our heuristic converges towards code having an error close to the best case, but with half less DPCodes. Finally, if the user's accuracy expectations are low, i.e., if an error bound of 2^{-14} is acceptable, then only one DPCode is enough to implement matrix multiplication for all the sizes.

4. Trade-offs between sharp error bounds and run-time overflows: matrix inversion

For a large set of applications, numerical analysts advise against computing matrix inversion since it is known to be numerically unstable. Yet, as stated by Higham [22, § 14], cases exist where the inverse conveys useful information. For instance, in wireless communications, matrix inversion is used in equalization algorithms [31] as well as detection estimation algorithms in space-time

coding [32]. In radar applications, Space Time Adaptive Processing algorithms (STAP) require the inversion a positive-definite covariance matrix [33].

Mainly motivated by the latter application, we present our approach to code synthesis for matrix inversion which is based on Cholesky decomposition. Sections 4.1 and 4.2 detail our method and explain how to implement it respectively. Finally, Section 4.3 exhibits our experimental results which mainly illustrate the sharp error bounds versus risk of overflow trade-offs.

4.1. A methodology for matrix inversion

A survey of floating-point matrix inversion and linear systems solving methods shows that there are many algorithms in use: Cramer's rule, LU decomposition, QR decomposition, ... [21]. A common pattern to the efficient algorithms is the decomposition of the input matrix into a product of easy to invert matrices (triangular or orthogonal matrices). LU decomposition, for instance, proceeds by Gaussian elimination to decompose an input matrix A into two triangular matrices L and U such that $A = LU$. Inverting triangular matrices being straightforward, solving the associated linear system is equally simple and so is obtaining the inverse A^{-1} by the formula $A^{-1} = U^{-1}L^{-1}$. Almost the same chain of reasoning is applicable to QR decomposition. In our tool-chain, we focus on a Cholesky decomposition based approach.

4.1.1. Matrix inversion using Cholesky decomposition

Symmetric positive-definite matrices are ubiquitous in signal processing and can be inverted through Cholesky decomposition which is known to be numerically stable [22]. Given a symmetric positive-definite matrix A, the method follows the three following steps:

1. matrix decomposition: computing a lower triangular matrix L such as $A = LL^T$,
2. triangular matrix inversion: computing L^{-1}, and
3. inverse re-composition through multiplication: $A^{-1} = L^{-T}L^{-1}$.

In floating-point arithmetic, the computationally intensive step is the decomposition part [21]. The decomposition step is also the missing link in fixed-point arithmetic. Indeed, we presented in Section 3 a methodology for fixed-point code synthesis for matrix multiplication that we first described in [25].

Remark that restraining to symmetric positive-definite matrices is not an overkill. Indeed, Cholesky decomposition can be used to invert any non-symmetric positive-definite matrix A by decomposing the following matrix: $M = AA^T$ which is guaranteed to be symmetric to obtain $M = LL^T$. From this decomposition, A^{-1} can be recovered using the formula:

$$A^{-1} = A^T L^{-T} L^{-1}. \tag{10}$$

4.1.2. The Cholesky decomposition step

The Cholesky decomposition of a matrix A is defined if A is a symmetric positive-definite matrix. This method exploits the structure of the matrix, is more efficient than Gaussian elimination, and is numerically stable [22, § 10]. It works by finding a lower triangular matrix L such that:

$$A = L \cdot L^T. \tag{11}$$

And by equating the coefficients in (11) and using the symmetry of A, the following formula for the general term of L is deduced:

$$\ell_{i,j} = \begin{cases} 0 & \text{if } i < j \\ \sqrt{c_{i,i}} & \text{if } i = j \\ \dfrac{c_{i,j}}{\ell_{j,j}} & \text{if } i \neq j \end{cases} \qquad \text{where} \quad c_{i,j} = a_{i,j} - \sum_{k=0}^{j-1} \ell_{i,k} \cdot \ell_{j,k} \tag{12}$$

Again, before generating code for $\ell_{i,j}$, one must generate code for its dependencies. These include all the $\ell_{i,k}$ and $\ell_{j,k}$ with $k \in \{0, \ldots, j-1\}$ as well as $\ell_{j,j}$ if the coefficient is not on the diagonal. Also, synthesizing code that computes $\ell_{i,j}$ from $c_{i,j}$ involves the square root and division operators. Therefore, as explained in Section 2.2.5, the intervention of the user is needed to provide the fixed-point format of the output of division. By doing so, the user sets the appropriate trade-off between the sharpness of the accuracy bounds and the risk of run-time overflows.

4.1.3. The triangular matrix inversion step

Using the so called backward and forward substitution techniques, inverting a triangular matrix is a straightforward process in floating-point arithmetic. Indeed, for a lower triangular matrix M, its inverse N is given by the following equation:

$$n_{i,j} = \begin{cases} 0 & \text{if } i < j \\ \dfrac{1}{m_{i,i}} & \text{if } i = j \\ \dfrac{-c_{i,j}}{m_{i,i}} & \text{if } i > j \end{cases} \qquad \text{where } \; c_{i,j} = \sum_{k=j}^{i-1} m_{i,k} \cdot n_{k,j}. \tag{13}$$

While in floating-point arithmetic, implementing these equations requires only three nested loops, it is more challenging in fixed-point arithmetic. Indeed, the coefficient $n_{i,j}$ depends on other coefficients of the inverse N, namely all the $n_{k,j}$ with $k \in \{j, \ldots, i-1\}$. This implies that the synthesis tool, when generating code that computes $n_{i,j}$ must know the ranges and formats of all the $n_{k,j}$ with $k \in \{j, \ldots, i-1\}$. It is clear that such a tool must follow a determined order in synthesizing code and that it must keep track of the formats and ranges of the computed coefficients so as to reuse them.

4.2. Code synthesis for triangular matrix inversion and Cholesky decomposition

The basic blocks introduced in the previous section were implemented in the FPLA tool. This tool was developed with the aim of generating fixed-point code for the most frequently used linear algebra routines. It handles the aspects peculiar to each class of input problems and relies on the CGPE library for the low-level code synthesis details. For triangular matrix inversion and Cholesky decomposition, FPLA internally keeps track of two matrices of fixed-point variables:

1. the input matrix, and
2. the resulting matrix.

The input matrix is not modified throughout the run. However, the resulting matrix is updated with the range, format, and error bound of each code returned by the CGPE. Therefore, FPLA must correctly handle the ordering of calls to CGPE in such a way that each coefficient's code is generated only after all the information on which it depends has been collected.

In triangular matrix inversion and according to Equation (13), the diagonal elements do not depend on any generated code. Therefore they may be computed in any order. The non-diagonal coefficients depend only on the coefficients that precede them on the same column. Therefore, FPLA can follow a row major, column major, or even a diagonal major approach. The latter consists in generating the elements on the diagonal, followed by those on the first sub-diagonal, and so on. The last code generated in this fashion would be that of the bottom left coefficient $\ell_{n-1,0}$. This is illustrated by Figure 6(a).

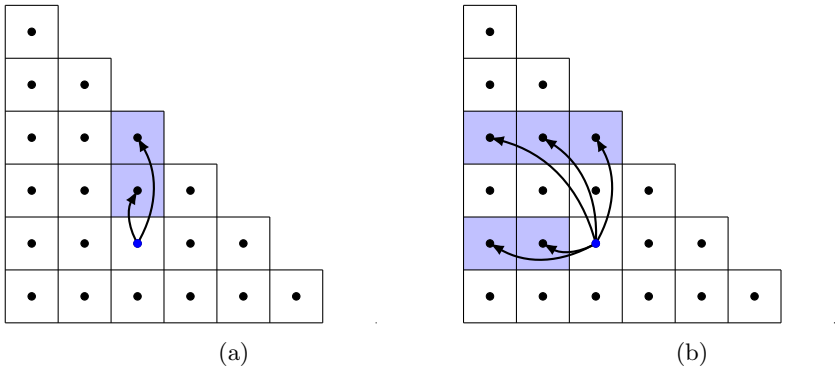


Figure 6: Dependencies of the coefficient $\ell_{5,3}$ (in blue) in the triangular matrix inversion (left) and in the Cholesky's decomposition (right) of a 6×6 matrix.

For Cholesky decomposition, a diagonal element $\ell_{i,i}$ depends on the generated coefficients that precede it on row i. A non-diagonal element $\ell_{i,j}$ depends on the first j elements of row i as well as the first $j+1$ elements of row j. FPLA may satisfy these dependencies by following either a row major or column major

synthesis strategy but not a diagonal major strategy. These dependencies are illustrated by Figure 6(b).

Once all the coefficient codes of the resulting matrix are generated, FPLA generates the global `C` file where each code assigns its result to the correct matrix index. For the sake of space, the synthesized C codes are not presented here, but are available in [15, § 5.3.1].

4.3. Experimental results

In this section, we first explain how we generate our benchmarks. Then we investigate the impact of the output format of division and study the speed of the generation and the sharpness of the error bounds of our generated codes. Finally we show the impact of the matrix condition number on the accuracy of the generated code.

4.3.1. Generation of fixed-point test matrices

The input matrices for which FPLA generates code are made of fixed-point variables. After the synthesis process, in order to test the resulting code on a set of benchmarks, we need to generate matrices of fixed-point numbers that belong to these fixed-point variables. For Cholesky decomposition, the matrices of fixed-point numbers should be symmetric positive-definite. To obtain such matrices, we followed the 5-stage process below:

1. Generate a lower triangular random matrix M whose fixed-point coefficients belong to the input fixed-point variables. Determine A by filling the upper side of the matrix with M^T. Formally, this is equivalent to computing $A = M + (M^T - \mathrm{diag}(M))$. At this point, A is a symmetric matrix that belongs to the input variable matrix.
2. Compute the smallest eigenvalue of A, denoted by $\lambda_{\min}$.
3. If $\lambda_{\min} > 0$, then the matrix A is positive-definite, and we are done.
4. Otherwise, compute $A' = A - (\lambda_{\min} + \delta)I$, for a small δ.
5. A' is guaranteed to be symmetric and positive-definite. This step, checks if all the coefficients of A' belong to their respective fixed-point variables. If it is the case, we are done, otherwise, either try to divide A' by a factor and retest Step 5, or restart from Step 1.

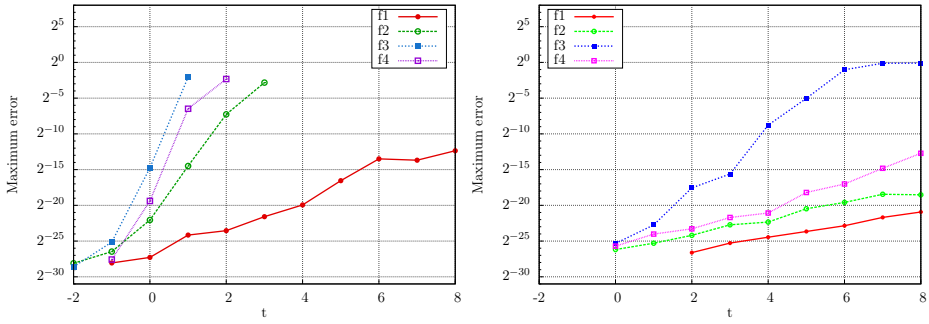
Generating triangular matrices is straightforward. For each coefficient, one must generate randomly a fixed-point number that belongs to the input fixed-point variable.

4.3.2. Impact of the output format of division on accuracy

As mentioned in Section 2.2.5, the output format of division must be explicitly set and has a great impact on the properties of the generated code. In this experiment, we use 4 different functions to set the integer part size of the result of a division. In each case, the output fraction part is determined so as each result fits on 32 bits. The functions used are the following:

1. $f_1(i_1, i_2) = t$,
2. $f_2(i_1, i_2) = \min(i_1, i_2) + t$,
3. $f_3(i_1, i_2) = \max(i_1, i_2) + t$,
4. $f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + t$,

where $t \in \mathbb{Z}$ is a user defined parameter, and i_1 and i_2 are the integer parts of the dividend and divisor, respectively. The function f_1 consists in fixing all the division results to the same fixed-point format. The experiment consists in computing the Cholesky decomposition and the triangular inversion of matrices of size 5 and 10, respectively. Using FPLA, we synthesize codes for each problem and each function in $\{f_1, f_2, f_3, f_4\}$, for t ranging from -2 to 8. Then for each synthesized code, 10 000 example instances are generated and solved both in fixed and floating-point arithmetics. Each example input is a matrix having 32-bits coefficients in the range between -1 and 1. Then the error considered is obtained by comparing the results to floating-point computations and by considering the maximum errors among the 10 000 samples. The results, for both basic blocks, are shown in Figures 7 and 8 for sizes 5 and 10, respectively, where the absence of the curve in some figures means that all of the examples overflew.

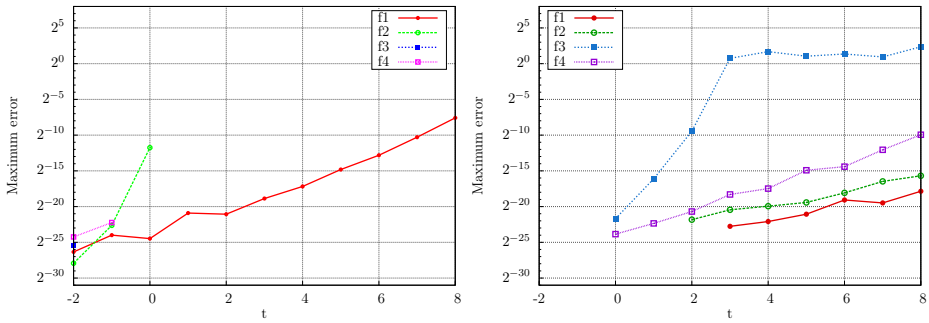


(a) Cholesky decomposition. (b) Triangular matrix inversion.

Figure 7: Results obtained on the Cholesky decomposition and triangular matrix inversion of matrices of size 5×5 with different functions to set the format of division.

Obviously, one can observe that the function used to determine the output format of division has a great impact on the accuracy of the generated code. For example, if we consider the case $t = 0$ on 5×5 Cholesky decomposition on Figure 7(a), using f_1 leads to an error of $\approx 2^{-28}$, while using f_3 gives an error $\approx 2^{-15}$, that is, twice larger than f_1. More particularly, we can observe that a good function choice is one that minimizes the output integer part but not too much. Indeed, as long as $t \geq -1$, using the function f_1 always leads to better maximum error than using the function f_3. In addition, surprisingly, as long as $t \geq -1$, the function that gives the best results is $f_1(i_1, i_2) = t$, namely the function that fixes explicitly all the division results of a resulting code to the same fixed-point format independently of the input formats.

Indeed the problem of using a function that depends on the input formats comes from the fact that it quickly leads to a growth of the integer part of



(a) Cholesky decomposition.

(b) Triangular matrix inversion.

Figure 8: Results obtained on the Cholesky decomposition and triangular matrix inversion of matrices of size 10×10 with different functions to set the format of division.

each computed coefficient, since it relies on the previously computed coefficient themselves. Hence the interest of f_1 is that it avoids this fast growth, and leads to result coefficients having a fixed and relatively small integer part, thus to tighter errors than the other functions. This remark is true for the four experiments of Figures 7 and 8 when $t \geq 0$ where f_1 is the only function that leads to successful results. This phenomenon becomes obvious as the matrix size increases.

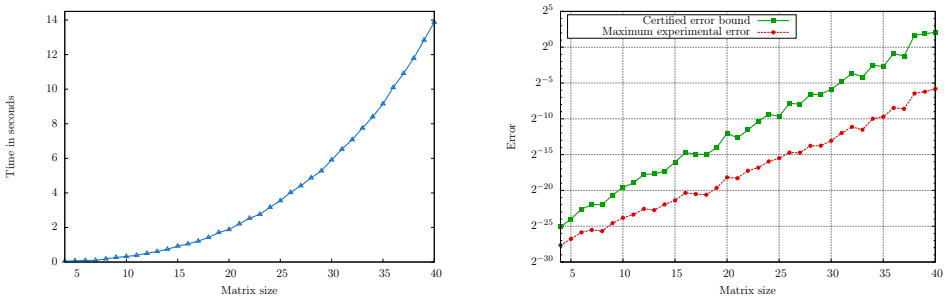
However, one should not be too optimistic and set the value t of f_1 to a very low value when implementing triangular matrix inversion, and cases occur where f_1 leads to unsuccessful results. Indeed, the value of the diagonal coefficient of the inverse matrix is $1/a_{i,i}$ and since $a_{i,i}$ may be arbitrarily small, one way to fix the right t is to choose it such that no division overflows occur when computing the division of the diagonal elements.

These experiments may also be seen as simulations to find the right t. Indeed, suppose we need to generate fixed-point code for the inversion of size-10 triangular matrices. FPLA comes with helper scripts that generate tests matrices and produce the figures similar to Figures 7 and 8. A strategy then consists in using these figures to restrain the search for the adequate output format. In the cases of interest, $g(i_1, i_2) = 3$ and $h(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor$, except for size-10 Cholesky decomposition, seem to be the most promising functions to set the output of division, in terms of accuracy.

4.3.3. Sharpness of the error bounds and generation time

The originality of our approach is the automatic generation of certified error bounds along with the synthesized code. This enables the generated code to be used in critical and precision sensitive applications. However, it is equally important that these bounds be sharp, at least for a large class of matrices. To investigate their sharpness, we compare in this second experiment the error

bounds for the case of triangular matrix inversion with the experimental errors obtained from inverting 10 000 sample matrices. This experiment is carried out using the function f_4 introduced in the previous experiment with $t = 1$. For each matrix size from 4 to 40, `C` code and error bounds are obtained by running FPLA. Figure 9(a) shows the evolution of the generation time when the size of the matrices grows. On an Intel Core i7-870 desktop machine running at 2.93 GHz, it does not exceed 14 seconds for 40×40 matrices. This is clearly an improvement of several orders of magnitude over a hand written fixed-point code.



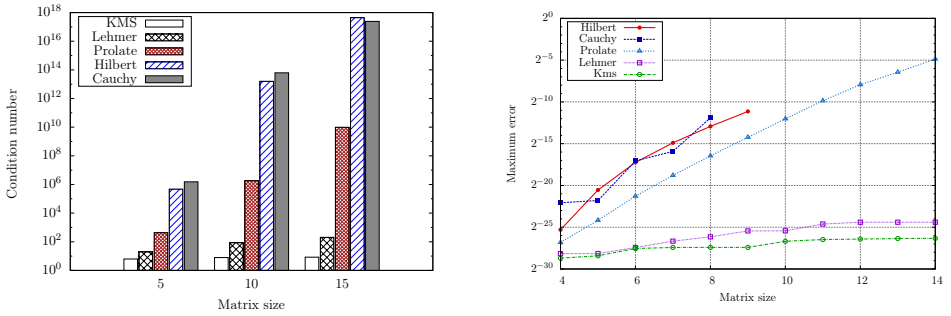
(a) (b)

Figure 9: Generation time (left) and comparison of the error bounds and experimental errors (right) for the inversion of triangular matrices of size 4 to 40.

Besides being quickly generated, Figure 9(b) shows that these codes have low accuracy bounds, at least when the matrix size is less than 30. The bounds vary from 2^{-26} to 2^2 while the experimental errors vary from 2^{-28} to 2^{-6}. The difference between the error bounds and experimental errors is less than 2 bits for size-4 matrices and is inferior to 5 bits for size-15 matrices, and it grows as the size of the input matrices grows. Nevertheless, the two curves have the same overall shape, and the gap between them grows smoothly. And, although the bounds obtained for matrices of size larger than 35 are too large to be useful in practice, the experimental errors are still tight enough and do not exceed 2^{-6}. These issues may be tackled by considering other means to handle division that are more suited to large matrices. Indeed, our experiments tend to show that the output format of division impacts heavily the accuracy of the result and that there is no way to determine a format that is adapted to all matrix sizes. We also argue that a bound of 2^{-12} on the inversion of size-20 matrices is satisfying for a broad range of applications, and this is a large improvement over hand-written fixed-point codes or codes whose numerical quality is asserted uniquely by simulations and *a posteriori* processes.

4.3.4. Impact of the matrix condition number on accuracy

The sample matrices considered in the previous experiments were randomly drawn in the input intervals. In this third experiment, we consider the Cholesky



(a) . (b)

Figure 10: Evolution of the conditioning of standard matrices of size 5, 10, and 15 (right) and maximum errors measured when computing the Cholesky decomposition of various kinds of standard matrices for sizes varying from 4 to 14 (left).

decomposition of some standard matrices namely, KMS, Lehmer, Prolate, Hilbert, and Cauchy matrices. These symmetric positive-definite matrices have multiple properties and are often provided by numerical computing environments. Indeed, we generated them using MATLAB's `gallery('name', size)` command. Among these, Hilbert and Cauchy matrices and to a lower extent Prolate are ill-conditioned as shown in Figure 10(a).

Nonetheless, with a fixed-point code generated for matrices in the input format $\mathbf{Q}_{1.31}$, we were able to check that the fixed-point results, whenever computable, are accurate as shown in Figure 10(b). For sizes larger than 8 and 9, respectively, overflows occur when computing the decompositions of Cauchy and Hilbert matrices. But this fact does not invalidate our approach. Indeed, these matrices are very ill-conditioned and are difficult to decompose accurately even in floating-point arithmetic. On the other hand, KMS and Lehmer matrices have a linearly growing condition number and are therefore very well suited to our approach. As shown by the two bottom curves of Figure 10(b), the code generated by FPLA decomposes these matrices with a precision of up to 24 bits.

In practice, nothing is noticeable on synthesis time, since the input matrices are made of fixed-point variables and that the same fixed-point code is generated for the 5 different classes of matrices. However, at run-time, ill-conditioned matrices tend to overflow more often and for smaller matrix sizes than well-conditioned matrices.

The above results show that accurate fixed-point codes accompanied by bounds on the rounding errors can be automatically generated in a few seconds to invert and to decompose matrices of sizes up to 40. The greatest difficulty of this process is related to fixing the output format of divisions.

Finally, a further research direction on matrix inversion consists in investigating other trade-offs involved in the code synthesis process. This includes, similarly to the work presented in Section 3, a study of the trade-off between code size and accuracy.

5. Conclusion

In this article, we addressed the automated synthesis of certified fixed-point programs and treated the particular cases of code generation for linear algebra basic blocks. The article tackles two recurrent issues encountered by embedded systems developers: the difficulty of fixed-point programming and the perceived low numerical quality of fixed-point computations.

We presented an example of an arithmetic model in Section 2 and implemented it in the CGPE library. Our arithmetic model gives bounds on the rounding errors of fixed-point square root and division as well as the means to implement them. These operators are useful for linear algebra basic blocks and are often overlooked in research publications.

Next, we used the arithmetic model and the CGPE tool to investigate code generation for higher level problems. Such problems include linear algebra basic blocks such as matrix multiplication and inversion. We showed that code synthesis for matrix multiplication can rely on straightforward approaches. But these approaches are on the edges of the accuracy versus code size spectrum. Therefore, we suggested, implemented, and provided experimental data for a novel approach that finds trade-offs between these algorithms.

Finally, we tackled matrix inversion. We showed that the order of synthesis must be arranged to respect the dependencies between the coefficients. Matrix inversion also provided a test case for our square root and division operators. We showed through our experiments that the user must be careful in setting the output format of division. Indeed, small output formats lead to tight accuracy bounds but involve a high risk of run-time overflows.

Finally, we consider that this work is extensible in the following directions:

Towards code synthesis for higher level problems. As of today, fixed-point programmers working on large applications have no choice but to implement part of them in floating-point arithmetic. For instance, an application like Space Time Adaptive Processing (STAP) for radars involves computing matrix inversion. Due to the large number of steps of matrix inversion, it is tedious to implement it without appropriate tools. The same applies to other frequently used basic blocks such as the Fast Fourier Transform, two dimensional convolutions, and advanced filters like Kalman's. Future work should extend the trade-off strategies studied in this article to these basic blocks.

Towards high level synthesis. High level synthesis consists in generating hardware architectures instead of software implementations. Targeting FPGAs has two justifications: this type of hardware is becoming more and more popular today and presents the advantage of allowing fully custom designs. For instance, a fixed-point FPGA implementation can use custom word-length numbers and therefore beat floating-point arithmetic in terms of chip area, energy consumption, memory bandwidth consumption, and latency.

References

[1] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, 2008.

[2] R. Yates, Fixed-Point Arithmetic: An Introduction, Digital Signal Labs (2013).

[3] D. Menard, D. Chillet, O. Sentieys, Floating-to-fixed-point conversion for digital signal processors, in: EURASIP Journal on Applied Signal Processing, 2006, pp. 1–15.

[4] Z. Nikolic, H. T. Nguyen, G. Frantz, Design and Implementation of Numerical Linear Algebra Algorithms on Fixed-Point DSPs, EURASIP J. Adv. Sig. Proceedings 2007.

[5] W. Sung, K.-I. Kum, Simulation-based word-length optimization method for fixed-point digital signal processing systems, IEEE Trans. Signal Processing 43 (12) (1995) 3087–3090.

[6] D.-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, G. A. Constantinides, Accuracy-guaranteed bit-width optimization, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 25 (10) (2006) 1990–2000.

[7] C. F. Fang, R. A. Rutenbar, T. Chen, Fast, Accurate Static Analysis for Fixed-Point Finite-Precision Effects in DSP Designs, in: Proc. of the 2003 IEEE/ACM International Conf. on Computer-aided Design (ICCAD'03), IEEE Computer Society, 2003, pp. 275–282.

[8] G. Golub, I. Mitchell, Matrix factorizations in Fixed Point on the C6x VLIW architecture, Tech. rep., Stanford University, Standford, California, USA (1998).

[9] M. Mehlhose, S. Schiffermüller, Efficient Fixed-Point Implementation of Linear Equalization for Cooperative MIMO Systems, 17th European Signal Processing Conference (EUSIPCO 2009).

[10] A. Irturk, B. Benson, S. Mirzaei, R. Kastner, GUSTO: An automatic generation and optimization tool for matrix inversion architectures, ACM Trans. Embed. Comput. Syst. 9 (4) (2010) 32:1–32:21.

[11] A. Y. A. Syed M. Qasim, Ahmed A. Telba, FPGA Design and Implementation of Matrix Multiplier Architectures for Image and Signal Processing Applications, International Journal of Computer Science and Network Security 10 (2) (2010) 168–176.

[12] I. Sotiropoulos, I. Papaefstathiou, A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions systems, in: Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, 2009, pp. 276–281.

[13] S. J. Campbell, S. P. Khatri, Resource and delay efficient matrix multiplication using newer FPGA devices, in: Proceedings of the 16th ACM Great Lakes Symposium on VLSI, GLSVLSI '06, ACM, New York, NY, USA, 2006, pp. 308–311.

[14] C. Mouilleron, G. Revy, Automatic Generation of Fast and Certified Code for Polynomial Evaluation, in: Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20), 2011, pp. 95–103.

[15] M. A. Najahi, Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks, Ph.D. thesis, Univ. de Perpignan Via Domitia (2014).

[16] B. Lopez, T. Hilaire, L.-S. Didier, Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation, in: Proc. of the Conference on Design and Architectures for Signal and Image Processing (DASIP), 2012, pp. 160–167.

[17] R. E. Moore, Interval Analysis, Prentice-Hall, 1966.

[18] E. Hansen, A generalized interval arithmetic, in: K. Nickel (Ed.), Interval Mathematics, Vol. 29 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1975, pp. 7–18. doi:10.1007/3-540-07170-9_2.
URL http://dx.doi.org/10.1007/3-540-07170-9_2

[19] M. D. Ercegovac, L. Imbert, D. W. Matula, J.-M. Muller, G. Wei, Improving Goldschmidt division, square root, and square root reciprocal, IEEE Trans. Computers 49 (7) (2000) 759–763.

[20] M. D. Ercegovac, T. Lang, Digital Arithmetic, Morgan Kaufmann Publishers, 2004.

[21] G. H. Golub, C. F. Van Loan, Matrix Computations (3rd Ed.), Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[22] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd Edition, Society for Industrial and Applied Mathematics, 2002.

[23] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, 3rd Edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[24] S. Kim, K. il Kum, W. Sung, Fixed-point optimization utility for c and c++ based digital signal processing programs, in: IEEE Trans. Circuits and Systems II, 1996, pp. 1455–1464.

[25] M. Martel, A. Najahi, G. Revy, Code Size and Accuracy-Aware Synthesis of Fixed-Point Programs for Matrix Multiplication, in: Proc. of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS'14), 2014, pp. 204–214.

[26] D.-U. Lee, J. D. Villasenor, Optimized custom precision function evaluation for embedded processors, IEEE Trans. Comput. 58 (1) (2009) 46–59.

[27] R. E. Moore, R. B. Kearfott, M. J. Cloud, Introduction to Interval Analysis, SIAM, 2009.

[28] M. I. Shamos, D. Hoey, Closest-point problems, in: FOCS, 1975, pp. 151–162.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms (3. ed.), MIT Press, 2009.

[30] C. Mouilleron, Efficient computation with structured matrices and arithmetic expressions, Ph.D. thesis, Univ. de Lyon - ENS de Lyon (2011).

[31] L. Zhou, L. Qiu, J. Zhu, A novel adaptive equalization algorithm for MIMO communication system, in: Proc. of the IEEE 62nd Vehicular Technology Conference (VTC-2005-Fall), Vol. 4, 2005, pp. 2408–2412. doi:10.1109/VETECF.2005.1558980.

[32] H. Chen, X. Deng, A. Haimovich, Layered turbo space-time coded mimo-ofdm systems for time varying channels, in: Proc. of the 2003 IEEE Global Telecommunications Conference (GLOBECOM'03), Vol. 4, 2003, pp. 1831–1836 vol.4. doi:10.1109/GLOCOM.2003.1258555.

[33] J. R. Guerci, Space-time adaptive processing for radar, Artech House, 2003.