Code Size and Accuracy-aware Synthesis of Fixed-point Programs for Matrix Multiplication

Matthieu Martel[1,2,3], Amine Najahi[1,2,3] and Guillaume Revy[1,2,3]
[1] *Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France*
[2] *Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France*
[3] *CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France*

Keywords: Automated Code Synthesis, Matrix Multiplication, Fixed-point Arithmetic, Certified Numerical Accuracy.

Abstract: In digital signal processing, many primitives boil down to a matrix multiplication. In order to enable savings in time, energy consumption, and on-chip area, these primitives are often implemented in fixed-point arithmetic. Various conflicting goals undermine the process of writing fixed-point codes, such as numerical accuracy, run-time latency, and size of the codes. In this article, we introduce a new methodology to automate the synthesis of small and accurate codes for matrix multiplication in fixed-point arithmetic. Our approach relies on a heuristic to merge matrix rows or columns in order to reduce the synthesized code size, while guaranteeing a target accuracy. We suggest a merging strategy based on finding closest pairs of vectors, which makes it possible to address in a few seconds problems such as the synthesis of small and accurate codes for size-64 and more matrix multiplication. Finally, we illustrate its efficiency on a set of benchmarks, and we show that it allows to reduce the synthesized code size by more than 50% while maintaining good numerical properties.

1 INTRODUCTION

Embedded systems are usually dedicated to one or few tasks that are often highly demanding on computational resources. Examples of computational applications widely deployed on these targets include discrete transforms (DCT, FFT, and digital filters) as well as image processing. Since floating-point implementations are costly in hardware resources, embedded system programmers prefer the fixed-point arithmetic (Yates, 2009). Indeed, the latter requires no specific hardware, relies on integer arithmetic only, and is highly efficient in terms of execution speed and energy consumption. Besides, in such systems, the accuracy is often critical, and there is a real need for the design of numerically certified basic blocks.

However, there are currently two hurdles to the widespread use of fixed-point arithmetic. First, fixed-point programming is a tedious process that requires a high degree of expertise since the programmer is in charge of such arithmetical details as alignments and overflow prevention. Second, the low dynamic range of fixed-point numbers compared to floating-point numbers led to a persistent belief that fixed-point computations are inherently unsafe and should be confined to uncritical applications. For all of these reasons, the fixed-point arithmetic has been for long limited to small and simple enough problems. In this article, our goal is to overcome these limitations for the case of matrix multiplication which is a widely deployed basic block in embedded applications.

For this purpose, we suggest and implement a novel methodology to automate the synthesis of tight and accurate codes for matrix multiplication that lends itself naturally to matrix-vector multiplication. In fact, a classical $n \times n$ matrix multiplication requires n^2 dot-products. For the sake of accuracy, each dot-product may rely on a particular optimized fixed-point code, leading to large code sizes. The challenge is therefore to reduce the number of synthesized dot-products without harming the overall accuracy of matrix multiplication. Our contribution is a novel strategy to carefully select and merge close enough rows and columns of the input matrices in order to reduce the number of synthesized dot-products, while guaranteeing a certain accuracy bound. This methodology is implemented in an automated tool and allows to quickly and easily treat problems previously considered intractable or unsuitable for this arithmetic. For instance, writing a code for the accurate multiplication of size-64 matrices is almost impossible to achieve by hand since it involves various trade-

Martel M., Najahi A. and Revy G..
Code Size and Accuracy-aware Synthesis of Fixed-point Programs for Matrix Multiplication.
DOI: 10.5220/0004884802040214
In *Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems* (PECCS-2014), pages 204-214
ISBN: 978-989-758-000-0

offs between code size, runtime latency, and accuracy. Moreover, to increase the level of confidence in the synthesized codes, our tool uses an analytic methodology based on interval arithmetic (Moore et al., 2009) to compute strict bounds on the roundoff errors and to guarantee the absence of overflow. With each synthesized code, it provides an accuracy certificate that bounds the accuracy errors due to finite wordlength effects, and that can be checked using the formal verification tool Gappa[1] (Melquiond, 2006).

Although many work on linear algebra routines in fixed-point arithmetic exists, to our knowledge, this article is the first one where certified fixed-point techniques are applied to such large problems, as size-64 matrix multiplication. Indeed (Nikolic et al., 2007) deals with the transformation from floating-point to fixed-point of matrix decomposition algorithms for DSPs and (Golub and Mitchell, 1998) with the implementation of matrix factorization algorithms for the particular C6x VLIW processor, while (Mehlhose and Schiffermüller, 2009) and (Irturk et al., 2010) discuss matrix inversion for the C64x+ DSP core and FPGAs, respectively. For the matrix multiplication, (Syed M. Qasim, 2010) present 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 (Sotiropoulos and Papaefstathiou, 2009), 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 (Campbell and Khatri, 2006) a delay and resource efficient methodology is introduced to implement a FPGA architecture for matrix multiplication in integer/fixed-point arithmetic. These works suggest a variety of techniques to determine and optimize fixed-point wordlengths. However, the roundoff error bounds in their methodologies are computed *a posteriori* by simulating the fixed-point design and comparing its output to that of a floating-point design. Also when determining fixed-point formats of intermediate variables, most of these works use a technique introduced by Sung *et al.* (Kim et al., 1998). This technique consists in using floating-point simulation to estimate the range of intermediate computations and convert the program to fixed-point arithmetic. In (Nikolic et al., 2007), the Sung technique is used to suggest a number of linear algebra routines. This simulation based technique has two drawbacks: 1. Its duration is exponentially proportional to the number of input variables. It is therefore impractical for large problems. 2. It provides no strict guaranties that intermediate computations will not overflow. On the other hand, the authors of (Lee et al., 2006) use a more rigorous approach to generate fixed-point codes for various problems. But we are not aware of any automated implementation and their examples do not supersede the size-8 DCT.

[1]See http://gappa.gforge.inria.fr.

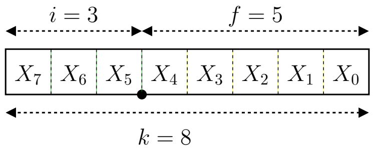


Figure 1: Fixed-point number in $Q_{3,5}$ on $k = 8$ bits.

This article is organized as follows: After some background on fixed-point arithmetic in Section 2, Section 3 discusses two straightforward approaches for synthesizing matrix multiplication programs. Then our new strategy to synthesize codes that satisfy certain tradeoff goals is presented in Section 4. Finally Section 5 is dedicated to some experimental benchmarks, before a conclusion in Section 6.

2 BACKGROUND ON FIXED-POINT ARITHMETIC

This section presents our fixed-point arithmetic model, including an error model, and it discusses the numerical issues arising from the computation of dot-products in this arithmetic.

2.1 Fixed-point Arithmetic Model

Fixed-point Number. Fixed-point arithmetic allows one to represent a real value by means of an integer associated to an *implicit* scaling factor. Let X be a k-bit signed integer in radix 2, encoded using two's complement notation. Combined with a factor $f \in \mathbb{Z}$, it represents the real value x defined as follows:

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

In the sequel of this article, $Q_{i,f}$ denotes the format of a given fixed-point value represented using a k-bit integer associated with a scaling factor f, with $k = i + f$, as shown in Figure 1.

Hence a fixed-point variable v in $Q_{i,f}$ is such that:

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

by step of 2^{-f}.

Set of Fixed-point Intervals. In practice, a fixed-point variable v may lie in a smaller range than the one defined 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 $Q_{i,f}$ format but with additional constraints on the runtime values it can take. For this reason, in this article, we denote by $\mathbb{F}\text{ix}$ the *set of fixed-point intervals*, where each element has a fixed format and an interval that narrows its runtime values.

Notice that unlike the exponent of floating-point variables, the scaling factor of fixed-point variables is fixed and is not encoded into the program. It is known only by the programmer, who is in charge of all the arithmetical details. For example, when adding two fixed-point values, both operand points have first to be aligned, that is, operands have to be set in the same fixed-point format. This alignment may lead to a potential roundoff error in finite precision.

2.2 Error Model in Fixed-point Arithmetic

Let v, v_ℓ, and v_r be three fixed-point variables in the formats $Q_{i,f}$, Q_{i_ℓ,f_ℓ}, and Q_{i_r,f_r}, respectively, and $\diamond$ be an operation in $\{+,-,\times\}$:

$$v = v_\ell \diamond v_r.$$

For the sake of conciseness, we do not deal here with the determination of fixed-point formats, but it can be found in (Yates, 2009) or (Mouilleron et al., 2013). Let us rather detail how we compute an interval $\mathbf{error}(v)$ enclosing the error entailed by the evaluation of v, where $\mathbf{value}(v)$ is an interval enclosing the value of v.

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

$$\mathbf{error}(v) = \mathbf{error}(v_\ell) \diamond \mathbf{error}(v_r).$$

- If the operation is a multiplication, we have:

$$\begin{aligned} \mathbf{error}(v) &= \mathbf{error}_\times + \mathbf{error}(v_\ell)\cdot\mathbf{error}(v_r) \\ &+ \mathbf{error}(v_\ell)\cdot\mathbf{value}(v_r) \\ &+ \mathbf{value}(v_\ell)\cdot\mathbf{error}(v_r), \end{aligned}$$

  where $\mathbf{error}_\times$ is the error entailed by the multiplication itself. Usually in fixed-point arithmetic this error is due to the truncation of the exact result of the multiplication to fit in a smaller format with f fraction bits. Hence we have:

$$\mathbf{error}_\times = [2^{-(f_\ell+f_r)} - 2^{-f}, 0].$$

  Most 32-bits DSP processors provide a 32×32 multiplier that returns the 32 most significant bits of the exact result, which is the multiplier considered in this work.

Left shift entails no error but only a possible overflow. However right shift may also be followed by the truncation of the exact result to fit in a smaller format with f fraction bits. Hence the evaluation of $v = v_\ell >> r$ entails an error defined as follows:

$$\mathbf{error}(v) = \mathbf{error}(v_\ell) + \mathbf{error}_{>>}$$

with in practice:

$$\mathbf{error}_{>>} = [2^{-f_\ell} - 2^{-f}, 0] \quad \text{and} \quad f = f_\ell - r.$$

2.3 Synthesis of Dot-product Code in Fixed-point Arithmetic

The code synthesized by our tool for matrix multiplication relies on dot-product evaluations as basic blocks. Besides the arithmetic model, the computation order of summations when evaluating dot-products has a great impact on the accuracy of the resulting codes. This is a well-known issue in floating-point arithmetic (Ogita et al., 2005),(Rump, 2009) but the same holds for fixed-point arithmetic. Precisely, summing n variables can be done using $\prod_{i=1}^{n-2}(2i+1)$ different ways.[2] For example, there exists 945 different schemes to implement a size-6 dot-product. Due to this huge combinatorics, there is a need for heuristics to address the issue of synthesizing accurate dot-product codes.

Our approach relies on the use of the CGPE[3] library (Mouilleron and Revy, 2011). Introduced by Revy *et al.*, it was initially designed to synthesize codes for polynomial evaluation, and it has been so far extended to summation and dot-product. CGPE implements the previous arithmetic model and embraces some heuristics to produce fast and numerically certified codes, relying on k-bit integer arithmetic only. Typically $k \in \{16, 32, 64\}$ and, for the rest of the article, we consider $k = 32$. CGPE takes as input an interval of fixed-point values for each coefficient and variable, a maximum error bound allowed for the code evaluation, and some architectural constraints. Then it outputs codes exposing a high degree of instruction-level parallelism and for which we are able to bound the evaluation error.

For dot-products, CGPE takes two size-n vectors $\mathcal{V}_1$ and $\mathcal{V}_2$ whose elements belong to the vectors of intervals

$$V_1, V_2 \in \mathbb{F}\text{ix}^n,$$

with $\mathbb{F}\text{ix}$ the set of fixed-point intervals defined in Section 2.1. It computes fast and numerically certified codes to implement $\mathcal{V}_1 \cdot \mathcal{V}_2$. In the sequel of the article, we will refer to the routine `DPSynthesis`(V_1, V_2) to compute automatically these codes.

[2] See http://oeis.org/A001147.

[3] See http://cgpe.gforge.inria.fr.

3 STRAIGHTFORWARD APPROACHES FOR THE SYNTHESIS OF MATRIX MULTIPLICATION PROGRAMS

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

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

In the article, 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.

We discuss now two straightforward approaches for the synthesis of a fixed-point program to multiply A' and B', where A' and B' are two matrices that belong to A and B, that is, where each element $A'_{i,k}$ and $B'_{k,j}$ belongs to the intervals $A_{i,k}$ 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:

$$C_{i,j} = A_{i,:} \cdot B_{:,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}, \qquad (2)$$

At the end of the section, we give some code size and accuracy estimates for each approach.

3.1 Accurate and Compact Approaches

Following Equation (2), a first straightforward approach to write code for matrix multiplication consists in synthesizing a program for each dot-product $A_{i,:} \cdot B_{:,j}$. Algorithm 1 below implements this approach, where `DPSynthesis`$(A_{i,:}, B_{:,j})$ produces a fast and numerically certified code for the computation of $A_{i,:} \cdot B_{:,j}$.

Remark that Algorithm 1 issues $m \times p$ requests to the `DPSynthesis` routine. At runtime, only one call to each generated code will be issued, for a total of $m \times p$ calls.

Algorithm 1: Accurate algorithm.

```
Input:
  Two matrices $A \in \mathbb{F}\mathrm{ix}^{m \times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n \times p}$
Output:
  Code to compute the product $A \cdot B$
Algorithm:
 1: for $1 \le i \le m$ do
 2:     for $1 \le j \le p$ do
 3:         DPSynthesis($A_{i,:}, B_{:,j}$)
 4:     end for
 5: end for
```

Algorithm 2: Compact algorithm.

```
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($\mathcal{U}, \mathcal{V}$)
```

To significantly reduce the size of the whole program, the computed codes could be refactored to evaluate more than one dot-product. Algorithm 2 below, whose input and output are the same as Algorithm 1, 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. Merging two fixed-point intervals means computing their union. Using this approach, it issues a unique call to `DPSynthesis` at synthesis. But, at runtime, $m \times p$ calls to the synthesized code are still needed to evaluate the matrix product.

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

$$A = \begin{pmatrix} [-1000, 1000] & [-3000, 3000] \\ [-1, 1] & [-1, 1] \end{pmatrix}$$

and

$$B = \begin{pmatrix} [-2000, 2000] & [-2, 2] \\ [-4000, 4000] & [-10, 10] \end{pmatrix},$$

where $A_{1,1}$ and $B_{1,1}$ are in the format $Q_{11,21}$, $A_{1,2}$ in $Q_{12,20}$, $A_{2,1}$, $A_{2,2}$, $B_{2,1}$ in $Q_{2,30}$, $B_{1,2}$ in $Q_{3,29}$, and $B_{2,2}$ in $Q_{5,27}$. Algorithm 1 produces 4 distinct codes, denoted by $\mathrm{DPCode}_{1,1}$, $\mathrm{DPCode}_{1,2}$, $\mathrm{DPCode}_{2,1}$, and $\mathrm{DPCode}_{2,2}$. On the other hand, Algorithm 2 first computes $\mathcal{U}$ and $\mathcal{V}$ as follows:

$$\mathcal{U} = A_{1,:} \cup A_{2,:} = ([-1000, 1000][-3000, 3000])$$

and

$$\mathcal{V} = B_{:,1} \cup B_{:,2} = \begin{pmatrix} [-2000, 2000] \\ [-4000, 4000] \end{pmatrix}.$$

Then, $\mathrm{DPCode}_{\mathcal{U},\mathcal{V}}$ is generated that evaluates the dot-product of $\mathcal{U}$ and $\mathcal{V}$. Table 1 summarizes the properties of the codes produced by both algorithms on the example.

On the first hand, Table 1(a) shows that Algorithm 1 produces codes optimized for the range of its entries: it is clearly superior in terms of accuracy since a dedicated code evaluates each runtime dot-product. On the other hand, Table 1(b) shows that, as expected, Algorithm 2 produces far less code: it is is optimal in terms of code size since a unique dot-product code is generated, but remains a worst-case in terms of accuracy. Fixed-point arithmetic is primarily

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

(a) Results for the 4 codes generated by Algorithm 1.

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

(b) Results for the code generated by Algorithm 2.

Dot-product	$A_{1,:} \cdot B_{:,1}$	$A_{1,:} \cdot B_{:,2}$	$A_{2,:} \cdot B_{:,1}$	$A_{2,:} \cdot B_{:,2}$
Evaluated using	DPCode$_{\mathcal{U},\mathcal{V}}$			
Output format	$Q_{26,6}$			
Certified error	$\approx 2^{-5}$			
Maximum error	$\approx 2^{-5}$			
Average error	$\approx 2^{-5}$			

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 article, we go further than Algorithms 1 and 2, and explore the possible means to achieve tradeoffs between the two conflicting goals.

3.2 Code Size and Accuracy Estimates

The dot-product is the basic block of classical matrix multiplication. In a size-n dot-product, regardless of the evaluation scheme used, n multiplications and $n-1$ additions are performed, since it can be reduced to a size-n summation. Also, depending on the formats of their operands, additions frequently require alignment shifts. Thus the number of shifts is bounded by $2n$ and, in absence of overflow, it falls to n. Hence $4n-1$ is a worst case bound on the number of elementary operations (additions, multiplications, and shifts) needed to evaluate a size-n dot-product. Globally, $(4n-1) \cdot t$ is a bound on the total size of a matrix multiplication code, where $t \in \{1, \cdots, m \times p\}$ is the number of generated dot-product codes. On the previous example, this bound evaluates to 28 for Algorithm 1 vs. 7 for Algorithm 2, since $n = 2$, and $t = 4$ and 1, respectively.

As for accuracy estimates, given a matrix multiplication program composed of several DPCodes, the maximum of all error bounds can be considered as a measure of the numerical quality. However, examples can be found where this measure does not reflect the numerical accuracy of all the codes. Consider for instance the example of Section 3.1, where both algorithms generate codes that have the same maximum error bound ($\approx 2^{-5}$), yet 3 of the 4 DPCodes generated by Algorithm 1 are by far more accurate than this bound. For this reason, we may also rely on the average error since we consider it as a more faithful criterion to estimate the accuracy.

4 DYNAMIC CLOSEST PAIR ALGORITHM FOR CODE SIZE VS. ACCURACY TRADEOFFS

In this section, we discuss how to achieve code size vs. accuracy tradeoffs, and the related combinatorics. We finally detail our new approach based on rows and columns merging and implemented in the *Dynamic Closest Pair algorithm*.

4.1 How to Achieve Tradeoffs

On the first hand, when developing a numerical application for embedded systems, the amount of program memory available imposes an upper bound on the code size. On the other hand, the nature of the application and its environment help in deciding on the required degree of accuracy.

Once these parameters set, the programmer tries Algorithms 1 and 2. Either the accurate algorithm (Algorithm 1) is not accurate enough: a solution we do not discuss here consists in adapting the fixed-point computation wordlengths to reach the required accuracy, as in (Lee and Villasenor, 2009). Or the compact algorithm (Algorithm 2) does not satisfy the code size constraint: other solutions must be considered such as adding more hardware resources. 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 vs. accuracy tradeoffs.

Recall that $m \times p$ dot-product calls are required at runtime. To evaluate them using less than $m \times p$ DPCodes, it is necessary to refactor some DPCodes so that they would evaluate more than one runtime 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 2 on a 4×4 matrix multipli-

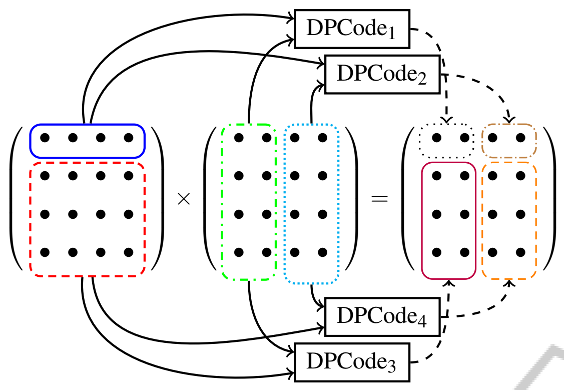


Figure 2: Merging strategy on a 4×4 matrix multiplication.

cation. In this example, each matrix is compressed into a 2×2 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 runtime.

4.2 Combinatorial Aspect of the Merging Strategy

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

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

and their two associated sets of vectors

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

In our case, the problem of finding an interesting code size vs. accuracy tradeoff reduces to finding partitions of the sets S_A and 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:
$$(4n-1) \cdot k_A \cdot k_B < \sigma,$$
2. and the error bound ε is guaranteed, that is:
$$\varepsilon_{\text{matrix}} < \varepsilon,$$
where $\varepsilon_{\text{matrix}}$ is either the minimal, the maximal, or the average computation error depending on the certification level required by the user.

Remark 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 compute the error bound 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[4] 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:[5]

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

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

$$\mathcal{P}(m,p) = B(m) \cdot B(p) - 2, \tag{3}$$

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

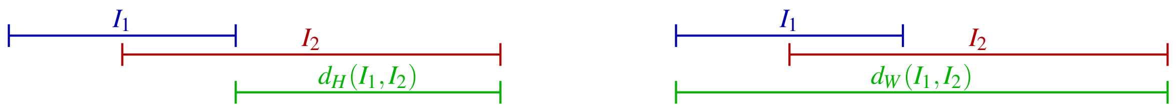
1. The partition of $\mathcal{S}_A$ and $\mathcal{S}_B$ into respectively m and p subsets which correspond to putting one and only one vector in each subset. This is the partitioning that leads to Algorithm 1.
2. The partition of $\mathcal{S}_A$ and $\mathcal{S}_B$ into one subset each. This partitioning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

Observe that even for small matrix sizes, the number $\mathcal{P}$ in Equation (3) is huge: $\mathcal{P}(5,5) = 2702$, $\mathcal{P}(10,10) \geq 2^{33}$, and $\mathcal{P}(64,64) \geq 2^{433}$. Therefore, aggressive heuristics will be necessary to tackle this problem. In the following, we introduce a method based on finding closest pairs of vectors according to a certain metric. This allows to find partitions that achieve the required tradeoff.

4.3 Dynamic Closest Pair Algorithm

Merging two fixed-point interval vectors $\mathcal{U}$ and $\mathcal{V}$ component-wise 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 3.1. 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.

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



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

Figure 3: Illustration of distances between two intervals.

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 interval arithmetic: the Hausdorff and the fixed-point distances. However, as our method does not use the mathematical properties of distances, any criterion that may discriminate between pairs of fixed-point interval vectors may be used. For instance, although not a distance, the width criterion introduced below was used in our experiments.

Hausdorff Distance. A fixed-point interval variable corresponds to a rational discrete interval. It follows that the Hausdorff distance (Moore et al., 2009), widely used as a metric in interval arithmetic, can be applied to fixed-point interval variables. Given two fixed-point intervals I_1 and I_2, this distance $d_H(I_1, I_2)$ is defined as follows:

$$d_H : \mathbb{F}\mathrm{ix} \times \mathbb{F}\mathrm{ix} \to \mathbb{R}^+$$
$$d_H(I_1, I_2) = \max\left\{\left|\underline{I_1} - \underline{I_2}\right|, \left|\overline{I_1} - \overline{I_2}\right|\right\},$$

where $\underline{I_1}$ and $\overline{I_1}$ stand for the lower and upper bound of the interval I_1, respectively. Roughly, this distance computes the maximum increase suffered by I_1 and I_2 when computing their union $I_1 \cup I_2$, as illustrated on Figure 3(a).

Fixed-point Distance. Contrarily to Hausdorff's distance which reasons on the intervals 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 intervals I_1 and I_2, this distance $d_F(I_1, I_2)$ is defined as follows:

$$d_F : \mathbb{F}\mathrm{ix} \times \mathbb{F}\mathrm{ix} \to \mathbb{N}$$
$$d_F(I_1, I_2) = |IntegerPart(I_1) - IntegerPart(I_2)|.$$

Analogously to Hausdorff distance, this distance computes the increase in the integer part suffered by I_1 and I_2 when computing their union $I_1 \cup I_2$.

Width Criterion. Given two fixed-point intervals I_1 and I_2, our third metric computes the width of the interval resulting from the union $I_1 \cup I_2$, as illustrated on Figure 3(b). Formally, it is defined as follows:

$$d_W : \mathbb{F}\mathrm{ix} \times \mathbb{F}\mathrm{ix} \to \mathbb{R}^+$$
$$d_W(I_1, I_2) = \left(\overline{I_1 \cup I_2} - \underline{I_1 \cup I_2}\right).$$

Algorithm 3: Dynamic Closest Pair algorithm.

```
Input:
  Two matrices $A \in \mathbb{F}\mathrm{ix}^{m \times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n \times p}$
  An accuracy bound $C_1$ (ex. average error bound is $< \varepsilon$)
  A code size bound $C_2$
  A metric $d$
Output:
  Code to compute $A \cdot B$ s.t. $C_1$ and $C_2$ are satisfied,
  or no code otherwise
Algorithm:
 1: $S_A \leftarrow \{A_{1,:}, \dots, A_{m,:}\}$
 2: $S_B \leftarrow \{B_{:,1}, \dots, B_{:,p}\}$
 3: while $C_1$ is satisfied do
 4:     $(u_A, v_A), d_A \leftarrow findClosestPair(S_A, d)$
 5:     $(u_B, v_B), d_B \leftarrow findClosestPair(S_B, d)$
 6:     if $d_A \leq d_B$ then
 7:         $remove(u_A, v_A, S_A)$
 8:         $insert(u_A \cup v_A, S_A)$
 9:     else
10:         $remove(u_B, v_B, S_B)$
11:         $insert(u_B \cup v_B, S_B)$
12:     end if
13:     for $(A_i, B_j) \in S_A \times S_B$ do
14:         DPSynthesis$(A_i, B_j)$
15:     end for
16: end while
17: /* Revert the last merging step. */
18: /* Check the bound $C_2$. */
```

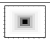
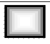
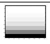
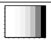
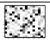
Notice that although the metrics are introduced as functions of two fixed-point intervals, we generalized them to fixed-point interval vectors by considering either the component-wise max or average value.

Given one of the above metrics and a set S of vectors, we are able to implement the $findClosestPair$ routine that returns the closest pair of vectors in S. There are several ways to implement such a routine. A $O(n^2)$ naive approach would 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 (Shamos and Hoey, 1975), (Cormen et al., 2009, §33).

Nevertheless, our contribution lies mainly in the design of Algorithm 3 which is based on a dynamic search of a code that satisfies an accuracy bound C_1 and code size bound C_2.

Here, we assume that Algorithm 1 satisfies the accuracy bound C_1, otherwise, no smaller code satisfying C_1 could be found. Therefore, Algorithm 3

Table 2: Weight matrices considered for the benchmarks.

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	$2^{\mathrm{rand}(0,\lfloor n/2 \rfloor-1)}$	

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 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 C_2, a tradeoff solution has been found. Otherwise, Algorithm 3 failed to find a code that satisfies both bounds C_1 and C_2. This algorithm was implemented in the FPLA tool,[6] that relies on CGPE and work is in progress to enhance it with more linear algebra routines. Section 5 studies the efficiency of this algorithm on a variety of fixed-point benchmarks.

5 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.

5.1 Experimental Environment

Experiments have been carried out on an Intel Xeon Quad Core 3.1 GHz running the GNU/Linux environment. In our experiments, we used 3 structured and 1 unstructured benchmarks. For structured benchmarks, the large coefficient distributions throughout the matrices follow different patterns. This is achieved through weight matrices, as shown in Table 2 where $W_{i,j}$ corresponds to the element of row i and column j of the considered weight matrix.

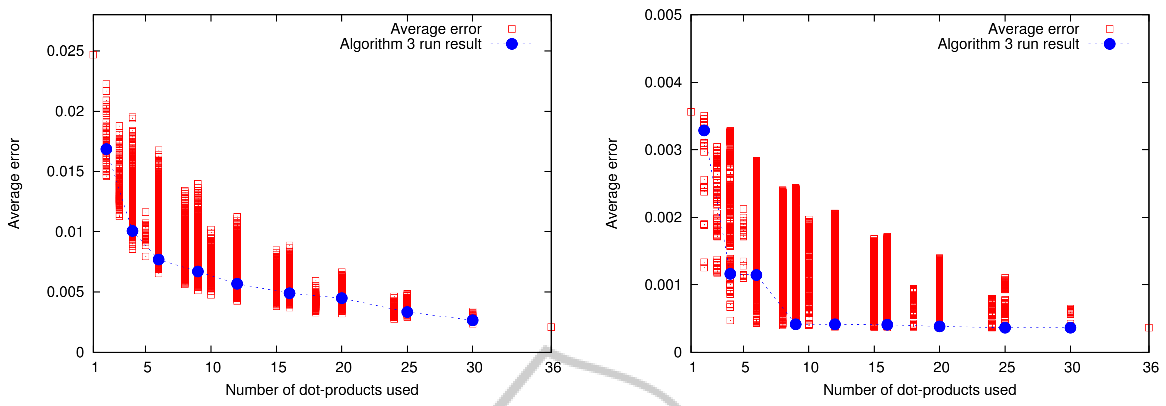
Notice, that the dynamic range defined as $max(W_{i,j})/min(W_{i,j})$ is the same for all benchmarks, and is equal to $2^{\lfloor n/2 \rfloor}$. The reason we did not directly use these matrices in our experiments is that the 3 first patterns correspond to structured matrices in the usual sense and that better algorithms to multiply structured matrices exist (Mouilleron, 2011). 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 2 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.

[6]FPLA: Fixed-Point Linear Algebra.

5.2 Efficiency of the Distance based Heuristic

As a first experiment, let us consider 2 of the previous benchmarks: *centered* 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 or column mergings: from Equation (3) and including the two degenerated cases, for size-6 matrices, there are 41 209 such mergings. For each of these, we synthesize the codes for computing $A \cdot B$, and determine the average error. This exhaustive experiment took approximately 2h15min per benchmark. Figure 4 shows the average error of the produced codes according to the number of DPCodes involved. Next we ran our tool using Hausdorff's distance to observe the behavior of Algorithm 3 and recorded all the intermediate steps. This took less than 10s for each benchmark and corresponds to the dark blue dots in Figure 4. Notice on both sides the accurate algorithm which produces 36 dot-products and the compact algorithm which produces only 1 dot-product. Notice also that Algorithm 3 is an iterative and deterministic algorithm. Once it goes in a wrong branch of the result space, this may lead to a code having an average error slightly larger than the best case. This 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.

Despite this, these experiments show the interest of our heuristic 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.



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

Figure 4: Average error according to the number of dot-product codes.

This is particularly the case on Figure 4(a) for *random* benchmarks. Moreover, Algorithm 3 converges toward code having good numerical quality much faster than the exhaustive approach.

5.3 Impact of the Metric on the Tradeoff 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 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 max based ones.

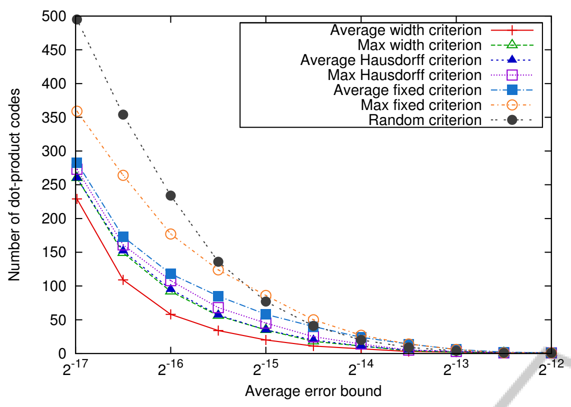
5.4 Impact of the Matrix Size

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 *average width* criterion 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 3 (where "–" means "no result has been found").

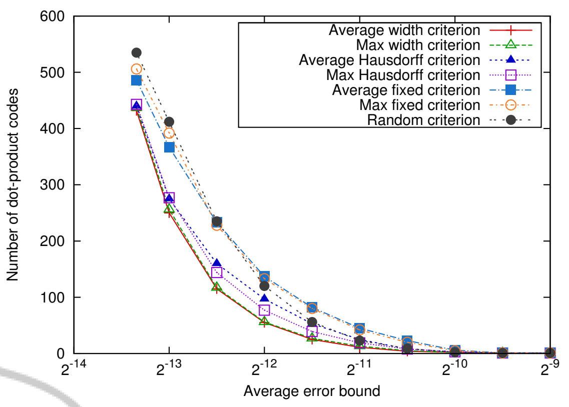
This shows clearly that our method is extensible to large matrices, since it allows to reduce the size of the problem to be implemented, while maintaining a good

Table 3: Number of DPCodes for various matrix sizes.

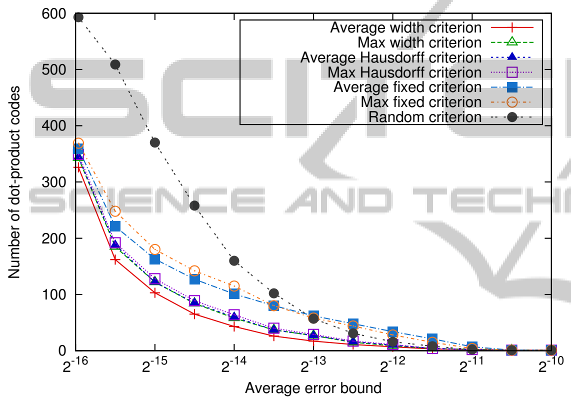
	2^{-21}	2^{-20}	2^{-19}	2^{-18}	2^{-17}	2^{-16}	2^{-15}	2^{-14}
8	24	6	1	1	1	1	1	1
16	–	117	40	16	3	1	1	1
32	–	–	552	147	14	2	1	1
64	–	–	–	2303	931	225	48	1



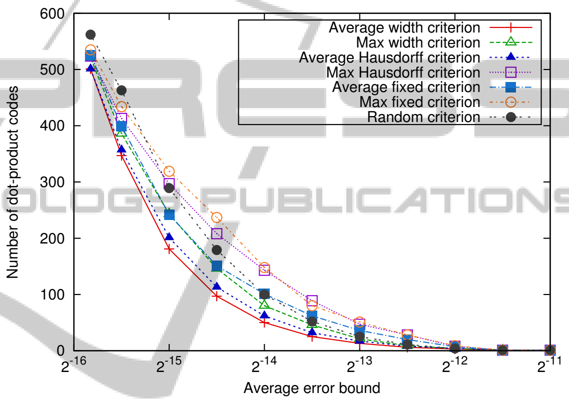
(a) Center matrices.



(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.

numerical quality. For example, the 64×64 accurate matrix multiplication would use 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. Remark finally that it falls to 1 DPCode if we target an error bound of 2^{-14}.

6 CONCLUSIONS

In this article, we discussed the automated synthesis of small and accurate programs for matrix multiplication in fixed-point arithmetic. More particularly, we presented a new strategy based on the merging of row or column vectors of the matrices, so as to reduce the size of the generated code while guaranteeing a certain error bound is satisfied. We also suggested criteria to decide which vectors are to be merged. The efficiency of this approach has been illustrated on a set of benchmarks. It remains now to validate this approach and the synthesized codes on real digital signal processing applications (including State Space filter realizations or MP3 compression).

In addition, the further research direction is threefold. As a first direction, it would be interesting to study the possibility of extending this approach to tri-matrix multiplication, widely used in DSP applications (ALshebeili, 2001),(Qasim et al., 2008), and even to matrix chain multiplications or matrix powers. In this work, we only deal with the basic matrix multiplication algorithm. As a second direction, we could investigate the interest of using other families of matrix multiplication algorithms, such as those based on blocking, and the impact on a fixed-point implementation. Finally, fixed-point arithmetic is widely used for implementing linear algebra algorithms on hardware providing no floating-point unit. In this sense, another research direction would be the automated synthesis of such algorithms in fixed-point arithmetic, like matrix inversion, for these particular hardware and the adaptation of the techniques presented here to such particular problems.

ACKNOWLEDGEMENTS

The work was supported by the ANR project DEFIS (*Ingénierie Numérique et Sécurité 2011*, ANR-11-INSE-0008).

REFERENCES

ALshebeili, S. A. (2001). Computation of higher-order cross moments based on matrix multiplication. *Journal of the Franklin Institute*, 338(7):811–816.

Campbell, S. J. and Khatri, S. P. (2006). Resource and delay efficient matrix multiplication using newer FPGA devices. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, GLSVLSI '06, pages 308–311, New York, NY, USA. ACM.

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

Golub, G. and Mitchell, I. (1998). Matrix factorizations in Fixed Point on the C6x VLIW architecture. Technical report, Stanford University, Standford, California, USA.

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

Kim, S., il Kum, K., and Sung, W. (1998). Fixed-point optimization utility for C and C++ based digital signal processing programs. In *IEEE Trans. Circuits and Systems II*, pages 1455–146.

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

Lee, D.-U. and Villasenor, J. D. (2009). Optimized Custom Precision Function Evaluation for Embedded Processors. *IEEE Transactions on Computers*, 58(1):46–59.

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

Melquiond, G. (2006). *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, ÉNS Lyon.

Moore, R. E., Kearfott, R. B., and Cloud, M. J. (2009). *Introduction to Interval Analysis*. SIAM.

Mouilleron, C. (2011). *Efficient computation with structured matrices and arithmetic expressions*. PhD thesis, Univ. de Lyon - ENS de Lyon.

Mouilleron, C., Najahi, A., and Revy, G. (2013). Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic. Technical Report 13006.

Mouilleron, C. and Revy, G. (2011). Automatic Generation of Fast and Certified Code for Polynomial Evaluation. In *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, Tuebingen, Germany.

Nikolic, Z., Nguyen, H. T., and Frantz, G. (2007). Design and Implementation of Numerical Linear Algebra Algorithms on Fixed-Point DSPs. *EURASIP J. Adv. Sig. Proc.*, 2007.

Ogita, T., Rump, S. M., and Oishi, S. (2005). Accurate Sum and Dot Product. *SIAM J. Sci. Comput.*, 26(6):1955–1988.

Qasim, S. M., Abbasi, S., Alshebeili, S., Almashary, B., and Khan, A. A. (2008). FPGA Based Parallel Architecture for the Computation of Third-Order Cross Moments. *International Journal of Computer, Information and Systems Science, and Engineering*, 2(3):216–220.

Rump, S. M. (2009). Ultimately Fast Accurate Summation. *SIAM J. Sci. Comput.*, 31(5):3466–3502.

Shamos, M. I. and Hoey, D. (1975). Closest-point problems. In *FOCS*, pages 151–162.

Sotiropoulos, I. and Papaefstathiou, I. (2009). A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions systems. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 276–281.

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

Yates, R. (2009). *Fixed-Point Arithmetic: An Introduction*. Digital Signal Labs.