Journal of Systems Architecture 000 (2016) 1–16

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

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

Matthieu Martel[a], Amine Najahi[b,c,d], Guillaume Revy[b,c,d,*]

[a] University Perpignan Via Domitia, Laboratoire LAMPS, F-66860, Perpignan, France
[b] University Perpignan Via Domitia, DALI, F-66860, Perpignan, France
[c] University Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
[d] CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France

ARTICLE INFO

Article history:
Received 25 January 2016
Revised 21 November 2016
Accepted 23 November 2016
Available online xxx

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

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 developers 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.

1. Introduction

Fixed-point arithmetic is a lightweight alternative to floating-point arithmetic. It does not require dedicated hardware, namely a floating-point unit (FPU), and executes more efficiently. However, developing fixed-point implementations requires numerical expertise from the developer, 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 developer.

In a typical design, DSP developers 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 transferring this software design to architectures lacking a FPU, or when mapping it to hardware, constraints like register size, speed, area, power consumption, or throughput frequently force the developer 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 fixed wordlength environment, such as in software implementations, 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, like 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.

[*] Corresponding author.
E-mail addresses: matthieu.martel@univ-perp.fr (M. Martel), amine.najahi@univ-perp.fr (A. Najahi), guillaume.revy@univ-perp.fr (G. Revy).

http://dx.doi.org/10.1016/j.sysarc.2016.11.010
1383-7621/© 2016 Elsevier B.V. All rights reserved.

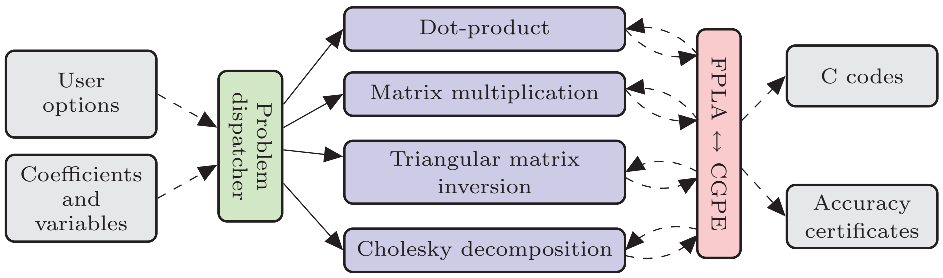

Fig. 1. The current flow of FPLA.

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, which describe software as well as hardware implementations, 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). FPLA handles the aspects peculiar to each class of input problems and relies on the CGPE library for the low-level code synthesis details, as shown in Fig. 1.

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 developer. 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 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 Eq. (1). For instance, if $V \in \mathbb{Z} \cap [-2^{k-1} + 2^{k-2}, 2^{k-1} - 2^{k-2}]$ in Eq. (1), then v is still in the $\mathbf{Q}_{i,f}$ format but with additional constraints on the run-time values it can take. For this reason, we shall denote by $\mathbb{F}\mathrm{ix}$ the *set of fixed-point variables*, where each element has a fixed-point format and an interval that narrows its run-time values.

[1] See http://cgpe.gforge.inria.fr/ and [14] for details.
[2] See http://www.manajahi.com/fpla/ and [15, § 6] for details.