1 Introduction

1.1 Overview

Arithmetic is one of the old topics in computing. It dates back to the many early civilizations that used the abacus to perform arithmetic operations. The seventeenth and eighteenth centuries brought many advances with the invention of mechanical counting machines like the slide rule, Schickard's Calculating Clock, Leibniz's Stepped Reckoner, the Pascaline, and Babbage's Difference and Analytical Engines. The vacuum tube computers of the early twentieth century were the first programmable, digital, electronic, computing devices. The introduction of the integrated circuit in the 1950s heralded the present era where the complexity of computing resources is growing exponentially. Today's computers perform extremely advanced operations such as wireless communication and audio, image, and video processing, and are capable of performing over 10^{15} operations per second.

Owing to the fact that computer arithmetic is a well-studied field, it should come as no surprise that there are many books on the various subtopics of computer arithmetic. This book provides a focused view on the optimization of polynomial functions and linear systems. The book discusses optimizations that are applicable to both software and hardware design flows; e.g., it describes the best way to implement arithmetic operations when your target computational device is a digital signal processor (DSP), a field programmable gate array (FPGA) or an application specific integrated circuit (ASIC).

Polynomials are among the most important functions in mathematics and are used in algebraic number theory, geometry, and applied analysis. Polynomial functions appear in applications ranging from basic chemistry and physics to economics, and are used in calculus and numerical analysis to approximate other functions. Furthermore, they are used to construct polynomial rings, a powerful concept in algebra and algebraic geometry.

One of the most important computational uses of polynomials is function evaluation, which lies at the core of many computationally intensive applications. Elementary functions such as sin, cos, tan, $\sin^{-1}$, $\cos^{-1}$, sinh, cosh, tanh, exponentiation and logarithm are often approximated using a polynomial function. Producing an approximation of a function with the required accuracy in a rather large interval may require a polynomial of a large degree. For instance, approximating the function $\ln(1 + x)$ in the range $[-1/2, 1/2]$ with an error less than 10^{-8}

requires a polynomial of degree 12. This requires a significant amount of computation, which without careful optimization results in unacceptable runtime.

Linear systems also play an important role in mathematics and are prevalent in a wide range of applications. A linear system is a mathematical model based on linear operators. Linear systems typically exhibit features and properties that are much simpler and easier to understand and manipulate than the more general, nonlinear case. They are used for mathematical modeling or abstraction in automatic control theory, signal processing, and telecommunications.

Perhaps the foremost computational use of linear systems is in signal processing. A typical signal processing algorithm takes as input a signal or a set of signals and outputs a transformation of them that highlights specific aspects of the data set. For example, the Fourier transform takes as the input the value of a signal over time and returns the corresponding signal transformed into the frequency domain. Such linear transforms are prevalent in almost any form of DSP and include the aforementioned discrete Fourier transform (DFT), as well as the discrete cosine transform (DCT), finite impulse response (FIR) filters, and discrete wavelet transform (DWT).

Polynomials and linear systems lie at the heart of many of the computer intensive tasks in real-time systems. For example, radio frequency communication transceivers, image and video compression, and speech recognition engines all have tight constraints on the time period within which they must compute a function; the processing of each input, whether it be an electromagnetic sample from the antenna, a pixel from a camera or an acoustic sample from a microphone, must be performed within a fixed amount of time in order to keep up with the application's demand. Therefore, the processing time directly limits the real-time behavior of the system.

The bulk of the computation in these applications is performed by mathematical functions. These functions include many of the aforementioned elementary functions (sin, cos, tan, exponentiation, and logarithm) as well as linear transforms (DFT, DCT, DWT, and FIR filters). Application developers often rely on hand-tuned hardware and software libraries to implement these functions. As these are typically a bottleneck in the overall execution of the application, the sooner they finish, the faster the applications run. However, small changes in the parameters of the function (e.g., moving from 16-bit to 32-bit data, changing the coefficients of a filter, adding more precision to the linear transform) require significant redesign of the library elements, perhaps even starting from scratch if the library does not support the exact specification that is required. Further, as the underlying computing platform changes, the libraries should ideally be ported to the new platform with minimal cost and turnaround time. Finally, designers require the ability to tradeoff between different performance metrics including speed, accuracy, and resource usage (i.e., silicon area for hardware implementation and the number of functional units and the amount of memory for software implementations). Therefore, methods to ease the design space exploration over these points are invaluable.

Many of the applications that we consider lie in the realm of embedded computing. These are nontraditional computing systems where the processor is a component of a larger system. Unlike desktops, laptops, and servers, embedded systems are not thought of as primarily computing devices. Example applications include anti-lock braking systems and navigation controls in automobiles, the Mars Rover and robotic surgical systems (along with many other robotics applications), smart phones, MP3 players, televisions, digital video recorders and cameras; these are just some of the devices that can be classified as embedded systems.

There has been an explosive growth in the market for embedded systems primarily in the consumer electronics segment. The increasing trend towards high performance and portable systems has forced researchers to come up with innovative techniques and tools that can achieve these objectives and meet the strict time to market requirements. Most of these consumer applications, such as smart phones, digital cameras, portable music and video players, perform some kind of continuous numerical processing; they constantly process input data and perform extensive calculations. In many cases, these calculations determine the performance and size of the system implemented. Furthermore, since these calculations are energy intensive, they are the major factors determining the battery life of the portable applications.

Embedded system designers face a plethora of decisions. They must attempt to delicately balance a number of often conflicting variables, which include cost, performance, energy consumption, size, and time to market. They are faced with many questions; one of the most important is the choice of the computational device. Microprocessors, microcontrollers, DSPs, FPGAs and ASICs are all appropriate choices depending on the situation, and each has its benefits and drawbacks. Some are relatively easy to program (microprocessors, microcontrollers, DSPs), while others (ASICs, FPGAs) provide better performance and energy consumption. The first three choices require a software design flow, while the last two (ASICs and FPGAs) require hardware design tools. Increasingly, computing devices are "system-on-chip" and consist of several of the aforementioned computational devices. For example, cell phones contain a mix of DSPs, microcontrollers, and ASICs – all on the same physical silicon die. This necessitates a mixed hardware/software design flow, which we discuss in more detail in the following.

Figure 1.1 illustrates a typical design flow for computationally intensive embedded system applications. The application is described using a specification language that expresses the functional requirements in a systematic manner. Additionally, the designer provides constraints, which include the available resources, timing requirements, error tolerance, maximum area, power consumption. The application specification is then analyzed and an appropriate algorithm is selected to implement the desired functionality. For example, signal processing applications must choose the appropriate transforms. Computer graphics applications must select the polynomial models for the surfaces, curves, and textures. An important step is the conversion of floating point representation

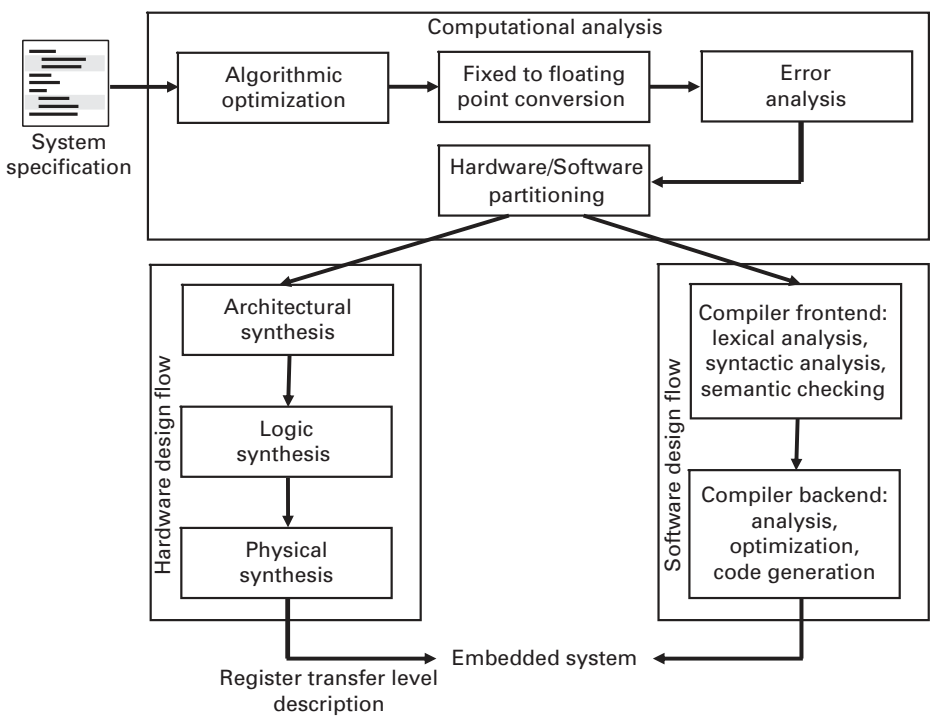

Figure 1.1 Embedded system design flow.

to fixed point representation. Though floating point representation provides greater dynamic range and precision than fixed point, it is far more expensive to compute. Most embedded system applications tolerate a certain degree of inaccuracy and use the much simpler fixed point notation to increase throughput and decrease area, delay, and energy. The conversion of floating point to fixed point produces some errors [1]. These errors should be carefully analyzed to see if they reside within tolerable limits [2, 3].

At this point, the application is roughly divided between hardware and software. The designer, perhaps with the help of automated tools, determines the parts of the system specification that should be mapped onto hardware components and the parts that should be mapped to software. For real-time applications with tight timing constraints, the computation intensive kernels are often implemented in hardware, while the parts of the specification with looser timing constraints are implemented in software. After this decision, the architecture of the system and the memory hierarchy are decided. The custom hardware portions of the system are then designed by means of a behavioral description of the algorithm using a hardware description language (HDL). Hardware synthesis tools transform these hardware descriptions into a register transfer level (RTL) language description by means of powerful hardware synthesis tools. These synthesis tools mainly perform scheduling, resource allocation, and binding of the various operations obtained from an intermediate representation of the

behavior represented in the HDL [4]. In addition the tools perform optimizations such as redundancy elimination (common subexpression elimination (CSE) and value numbering) and critical path minimization. The constant multiplications in the linear systems and polynomials can be decomposed into shifts and additions and the resulting complexity can be further reduced by eliminating common subexpressions [5–8]. Furthermore, there are some numeric transformations of the constant coefficients that can be applied to linear transforms to reduce the strength of the operations [9, 10]. This book provides an in-depth discussion of such transforms. The order and priorities of the various optimizations and transformations are largely application dependent and are the subject of current research. In most cases, this is done by evaluating a number of transformations and selecting the one that best meets the constraints [11]. The RTL description is then synthesized into a gate level netlist, which is subsequently placed and routed using standard physical design tools.

For the software portion of the design, custom instructions tuned to the particular application may be added [12–14]. Certain computation intensive kernels of the application may require platform dependent software in order to achieve the best performance on the available architecture. This is often done manually by selecting the relevant functions from optimized software libraries. For some domains, including signal processing applications, automatic library generators are available [11]. The software is then compiled using various transformations and optimization techniques [15]. Unfortunately, these compiler optimizations perform limited transformations for reducing the complexity of polynomial expressions and linear systems. For some applications, the generated assembly code is optimized (mostly manually) to improve performance, though it is not practical for large and complex programs. An assembler and a linker are then used to generate the executable code.

Opportunities for optimizing polynomial expressions and linear systems exist for both the hardware and the software implementations. These optimizations have the potential for huge impact on the performance and power consumption of the embedded systems. This book presents techniques and algorithms for performing such optimizations during both the hardware design flow and the software compilation.

1.2 Salient features of this book

The unique feature of this book is its treatment of the hardware synthesis and software compilation of arithmetic expressions. It is the first book to discuss automated optimization techniques for arithmetic expressions. The previous literature on this topic, e.g., [16] and [17], deals only with the details of implementing arithmetic intensive functions, but stops short of discussing techniques to optimize them for different target architectures. The book gives a detailed introduction to the kind of arithmetic expressions that occur in real-life applications,

such as signal processing and computer graphics. It shows the reader the importance of optimizing arithmetic expressions to meet performance and resource constraints and improve the quality of silicon. The book describes in detail the different techniques for performing hardware and software optimizations. It also describes how these techniques can be tuned to improve different parameters such as the performance, power consumption, and area of the synthesized hardware. Though most of the algorithms described in it are heuristics, the book also shows how optimal solutions to these problems can be modeled using integer linear programming (ILP). The usefulness of these techniques is then verified by applying them on real benchmarks.

In short, this book gives a comprehensive overview of an important problem in the design and optimization of arithmetic intensive embedded systems. It describes in detail the state of the art techniques that have been developed to solve this problem. This book does not go into detail about the mathematics behind the arithmetic expressions. It assumes that system designers have performed an analysis of the system and have come up with a set of polynomial equations that describe the functionality of the system, within an acceptable error. Furthermore, it assumes that the system designer has decided what is the best architecture (software, ASIC or FPGA or a combination of them) to implement the arithmetic function. The book does not talk about techniques to verify the precision of the optimized arithmetic expressions. Techniques such as those discussed in [2] and [18] can be used to verify if the expressions produce errors within acceptable limits.

1.3 Organization

- Chapter 2 illustrates the different applications that require arithmetic computation. It shows how polynomial expressions and linear computations reside in a number of applications that drive embedded systems and high-performance computing markets. The chapter discusses how polynomials are employed in computer graphics applications and describes the use of linear systems in DSP, cryptography, and address calculation.
- Chapter 3 presents an overview of the software compilation process and shows opportunities to optimize linear systems and polynomial expressions.
- Chapter 4 provides a high-level description of the hardware synthesis design flow. It explains the major steps in this design flow including input specification, algorithm optimization, scheduling, binding, and resource allocation. The chapter illustrates these concepts with a case study of an FIR filter.
- Chapter 5 gives a brief introduction to the concepts in digital arithmetic. It explains number representations including fixed and floating point representations. Also, it presents different architectures to perform two-operand and multiple-operand addition. These concepts are important in order to gain an understanding of the optimizations described in Chapters 6 and 7.

- Chapter 6 presents algebraic optimization techniques for polynomial expressions. It describes representations of polynomial expressions as well as various algorithms to optimize polynomials for both hardware and software implementation. The chapter concludes with experimental results showing the relative benefits of the various optimization techniques.
- Chapter 7 describes algebraic techniques for the optimization of linear arithmetic computations such as FIR filters and other linear transforms. Algorithms to optimize multiple-operand addition are also presented. Finally, the chapter presents experimental results where the usefulness of these techniques is demonstrated using real-life examples.

1.4 Target audience

When writing this book we had several audiences in mind. Much of the material is targeted towards specialists, whether they be researchers in academia or industry, who are designing both software and hardware for polynomial expressions and/or linear systems. The book also provides substantial background of the state of the art algorithms for the implementation of these systems, and serves as a reference for researchers in these areas. This book is designed to accommodate readers with different backgrounds, and the book includes some basic introductory material on several topics including computer arithmetic, software compilation, and hardware synthesis. These introductory chapters give just enough background to demonstrate basic ideas and provide references to gain more in-depth information. Most of the book can be understood by anyone with a basic grounding in computer engineering. The book is suitable for graduate students, either as a reference or as textbook for a specialized class on the topics of hardware synthesis and software compilation for linear systems and polynomial expressions. It is also suitable for an advanced topics class for undergraduate students.

References

[1] C. Shi and R.W. Brodersen, An automated floating-point to fixed-point conversion methodology, *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003*. Washington, DC: IEEE Computer Society, 2003.
[2] C.F. Fang, R.A. Rutenbar, and T. Chen, Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs, *International Conference on Computer Aided Design (ICCAD), San Jose, 2003*. Washington, DC: IEEE Computer Society, 2003.
[3] D. Menard and O. Sentieys, Automatic evaluation of the accuracy of fixed-point algorithms, *Design, Automation and Test in Europe Conference and Exhibition, 2002*. Washington, DC: IEEE Computer Society, 2002.

[4] G.D. Micheli, *Synthesis and optimization of digital circuits*, New York, NY: McGraw-Hill, 1994.
[5] M. Potkonjak, M.B. Srivastava, and A.P. Chandrakasan, Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, **15**(2), 151–65, 1996.
[6] R. Pasko, P. Schaumont, V. Derudder, V. Vernalde, and D. Durackova, A new algorithm for elimination of common subexpressions, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, **18**(1), 58–68, 1999.
[7] R. Pasko, P. Schaumont, V. Derudder, and D. Durackova, Optimization method for broadband modem FIR filter design using common subexpression elimination, *International Symposium on System Synthesis, 1997*. Washington, DC: IEEE Computer Society, 1997.
[8] A. Hosangadi, F. Fallah, and R. Kastner, Common subexpression elimination involving multiple variables for linear DSP synthesis, *IEEE International Conference on Application-Specific Architectures and Processors, 2004*. Washington, DC: IEEE Computer Society, 2004.
[9] A. Chatterjee, R.K. Roy, and M.A. D'Abreu, Greedy hardware optimization for linear digital circuits using number splitting and refactorization, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **1**(4), 423–31, 1993.
[10] H.T. Nguyen and A. Chatterjee, Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **8**, 419–24, 2000.
[11] M. Puschel, B. Singer, J. Xiong, *et al*., SPIRAL: a generator for platform-adapted libraries of signal processing algorithms, *Journal of High Performance Computing and Applications*, **18**, 21–45, 2004.
[12] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sarrafzadeh, Instruction generation for hybrid reconfigurable systems, *International Conference on Computer Aided Design*. New York, NY: ACM, 2001.
[13] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, Automatic instruction set extension and utilization for embedded processors, *IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2003*. Washington, DC: IEEE Computer Society, 2003.
[14] Tensilica Inc., http://www.tensilica.com.
[15] S.S. Muchnick, *Advanced Compiler Design and Implementation*, San Francisco, CA: Morgan Kaufmann Publishers, 1997.
[16] J.P. Deschamps, G.J.A. Bioul, and G.D. Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*, New York, NY: Wiley-Interscience (2006).
[17] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, third edition. Springer, 2007.
[18] C. Fang Fang, R.A. Rutenbar, M. Puschel, and T. Chen, Toward efficient static analysis of Finite-Precision effects in DSP applications via affine arithmetic modeling, *Design Automation Conference*. New York, NY: ACM, 2003.

2 Use of polynomial expressions and linear systems

2.1 Chapter overview

Polynomial expressions and linear systems are found in a wide range of applications: perhaps most fundamentally, Taylor's theorem states that any differentiable function can be approximated by a polynomial. Polynomial approximations are used extensively in computer graphics to model geometric objects. Many of the fundamental digital signal processing transformations are modeled as linear systems, including FIR filters, DCT and H.264 video compression. Cryptographic systems, in particular, those that perform exponentiation during public key encryption, are amenable to modeling using polynomial expressions. Finally, address calculation during data intensive applications requires a number of add and multiply operations that grows larger as the size and dimension of the array increases. This chapter describes these and other applications that require arithmetic computation. We show that polynomial expressions and linear systems are found in a variety of applications that are driving the embedded systems and high-performance computing markets.

2.2 Approximation algorithms

Polynomial functions can be used to approximate any differentiable function. Given a set of points, the unisolvence theorem states that there always exists a unique polynomial, which precisely models these points. This is extremely useful for computing complex functions such as logarithm and trigonometric functions and forms the basis for algorithms in numerical quadrature and numerical ordinary differential equations. More precisely, the unisolvence theorem states that, given a set of $n+1$ unique data points, a unique polynomial with degree n or less exists.

As an example, consider the Taylor expansion of $\sin(x)$ approximated to four terms:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}. \tag{2.1}$$

This is a polynomial of degree 7 that approximates the sine function. Assuming that the terms 1/3!, 1/5!, and 1/7! are precomputed (these will be denoted as S_3, S_5,

and S_7, respectively), the naïve evaluation of this polynomial representation requires 3 additions/subtractions, 12 variable multiplications, and 3 constant multiplications. However, it is possible to optimize this polynomial to reduce the number of operations needed for its computation. For example, the techniques described in this book produce the following set of equations, which are equivalent to the four-term Taylor expansion of sin (x):

$$\begin{aligned} d_1 &= x \cdot x, \\ d_2 &= S_5 - S_7 \cdot d_1, \\ d_3 &= d_2 \cdot d_1 - S_3, \\ d_4 &= d_3 \cdot d_1 + 1, \\ \sin(x) &= x \cdot d_4. \end{aligned}$$

Here, only three additions/subtractions, four variable multiplications, and one constant multiplication are needed.

It is noteworthy that computing these expressions, even in their optimized form, is expensive in terms of hardware, cycle time, and power consumption. If the arguments to these functions are known beforehand, the functions can be precomputed and stored in lookup tables in memory. However, in cases where these arguments are not known or the memory size is limited, these expressions must be computed during the execution of the application that uses them.

2.3 Computer graphics

Computer graphics is a prime example of an application domain that uses polynomials to approximate complex functions. The use of computer graphics is widespread and includes applications such as video games, animated movies, and scientific modeling. In general, these applications are computationally expensive. Advanced graphics is increasingly being integrated into embedded devices due to the consumer demand and improvements in technology. Therefore, techniques that optimize computation time, power, energy, and throughput for graphics applications are of utmost importance.

Polynomials are the fundamental model for approximating arcs, surfaces, curves, and textures. In fact, most geometric objects are formulated in terms of polynomial equations, thereby reducing many graphic problems to the manipulation of polynomial systems [1]. Therefore, solving polynomial systems is an elementary problem in many geometric computations. As an example, consider the process of spline interpolation, which is used to model textures and surfaces. A spline is a method of approximation, in which a function is divided piecewise into a set of polynomials, i.e., each piece of the function is approximated using a polynomial. More formally, given a set of $n+1$ distinct points, a k-spline function is a set of n polynomial functions with degree less than or equal to k. This interpolation allows each polynomial to have a low degree, as opposed to