Towards Reversed Approximate Hardware Design

Saman Froehlich Daniel Große Rolf Drechsler
Cyber-Physical Systems, DFKI GmbH and Group of Computer Architecture, University of Bremen, Germany
saman.froehlich@dfki.de {grosse,drechsle}@cs.uni-bremen.de

***Abstract*—Approximate computing is an emerging design paradigm for trading off computational accuracy for computational effort. Due to their inherited error resilience many applications significantly benefit from approximate computing. To realize approximation, dedicated approximate circuits have been developed and provide a solid foundation for energy and time efficient computing. However, when it comes to the design and integration of the approximate HW, complex error analysis is required to determine the effect of the error with respect to application specific error norms. This frequently leads to sub-optimal results.**

In this work, we propose to reverse the typical design flow for approximate HW and demonstrate the new flow for a first application: LU-Factorization, which is one of the most basic and most popular numerical algorithm known. The general idea of the reversed flow for approximate HW design is to start with the application and determine the required computational accuracy such that the computational error of the result is below the application specific error bound. This allows us to push the approximate HW to its limits, while guaranteeing that the result is correct by construction wrt. the requirements. The effectiveness of our approach for LU-Factorization is shown on a well-known and large set of benchmarks.

I. Introduction

Approximate computing is an emerging field of research in computer science. The main purpose of approximate computing is to trade off accuracy for computational speed and HW complexity. A lot of effort has been spent in the development of approximate HW, such as adders [1], [2], [3], multipliers [4], [5] and general Synthesis and HW generation [6], [7], [8], [9].

The approaches for approximation differ, as some tweak the HW parameters, for example by voltage scaling and over clocking [10], [11], while others modify the behavior by changing the implemented function (e.g. [1], [8], [12], [5]).

However, designing and integrating approximate HW for a concrete application is often a difficult task. Approximate HW is evaluated with respect to typical error metrics for approximate computing (e.g. error-rate, worst-case error, bit-flip error) [13], [14]. Linking these error metrics to application specific error norms can be very challenging.

In this paper, we first review the conventional approximate HW design flow. As application domain we have chosen numerical algorithms, since techniques for error analysis are well known in this field . We identify the main shortcoming of the conventional approximate HW design flow, that is, long design loops are necessary. These loops result from numerous iterations of the complete approximate design flow starting from different approximate HW components until a suitable approximate HW component is found which satisfies the application specific error bound. To overcome this deficiency, we *reverse the conventional approximate HW design flow* by starting from the application specific error bound such that we can take full advantage of the approximate HW which is *correct by construction* and thus guarantees that the application specific error bound holds.

To demonstrate our proposed reversed approximate HW design flow, we consider as a first application the problem of solving linear equations. This problem is a challenge faced in many applications: electronic circuits when using Kirchhoffs rules or network analysis when analyzing traffic flows, to mention only a few (a lot of examples can be found in [15]). In this context, the *LU-Factorization* is one of the most common ways to solve a linear system of equations of the form $Ax = b$ directly, i.e. not using an iterative method. Essentially, LU-Factorization is based on decomposing the matrix representing the linear system into a product of two triangular matrices (one *L*ower and one *U*pper triangular matrix). The major advantage of this approach is that solving the same system A for many right-hand sides $b_1, b_2, \dots$ can be done very fast, once the LU-Factorization is computed (cf. [15], p.494).

Following the proposed reversed approximate HW design flow, we demonstrate how to speed up the computations during LU-Factorization using approximate computing. More precisely, for a given LU-Factorization and an application specific upper bound on the loss of accuracy of the result, we derive a bound for the computational accuracy needed which determines the integrated approximation. This bound can be calculated before any right-hand side b_i is known and thus offline and without any negative effect on the runtime of the application. The formula for the bound is not restricted to a specific error-norm, which can differ between applications.

We demonstrate the effectiveness of our approach for several systems of equations of different sizes with different right-hand sides using approximate HW. For this purpose, we use the ideas of [4] of reducing the computational complexity of multiplications by implementing a floating-point multiplier, which ignores the l least significant bits of the mantissa. Our experiments show that significant speed-ups with guaranteed accuracy are possible.

The remainder of this paper is structured as follows: In Section II related word is reviewed. Section III reviews the conventional approximate HW design flow and introduces the proposed reversed design flow. Afterwards, in Section IV the basics for the considered numerical algorithm, i.e. LU-Factorization are reviewed. We derive error bounds

This work was supported in part by the German Research Foundation (DFG) within the project MANIAC (DR 287/29-1) and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

depending on accuracy requirements in Section V and give a method on how to relate these to floating-point arithmetic in Section VI. Section VII presents the experimental evaluation. Finally, Section VIII concludes the paper.

II. Related Work

A lot of work has been done in analyzing the accuracy of Gaussian Elimination and LU-Factorization.

Wilkinson was the first one to analyze rounding errors in the context of numerical algorithms in detail in [16] and [17]. Amongst other things, he introduced the common models for rounding errors induced by the usage of floating-point arithmetic. Others, such as the author of [18], have built upon his work.

The author of [19] gives a summary of the work done on the accuracy of Gaussian Elimination. He gives some information about the accuracy of floating-point numbers in [18], it's application in numerical algorithms and their implementations. Alongside other applications he analyzes the sensitivity of linear systems towards perturbations, after introducing some more basics like vector and matrix norms etc..

In [20], the authors give a detailed analysis of the backward error for the ∞-norm for totally positive matrices. They analyze rounding errors in the context of spline interpolation.

The author of [21] did some general work to analyze algorithms and based on this, derived concrete formulas for linearized errors of the calculation of the LU-Factorization and the solving of linear systems in [22].

The authors of [23] have introduced a way on how to use approximation in iterative algorithms. They give general schemes on how to adopt the level of approximation in a flexible environment based on the results and the change of the results of each iteration. However they do not give any details about error bounds and do not mention direct solvers for linear equations.

III. Approximate HW Design Flow for Numerical Applications

First, in this section we review the conventional approximate HW design flow for numerical applications and describe its shortcomings. Then, we introduce the proposed reversed flow which provides a direct solution to take full advantage of approximate HW for the considered numerical applications.

A. Conventional Approximate HW Design Flow

Given a numerical application the design team is faced with the question of how to select (or built) an approximate HW design such that approximation is fully exploited, while at the same time application specific error bounds are not violated. Here, approximate HW designs are usually evaluated with respect to error metrics specific to approximate computing. A transfer to application specific error norms is generally not straightforward.

Fig. 1(a) illustrates the conventional design flow for designing approximate HW for a numerical application: First, an approximate HW design is chosen. For example, a dedicated approximate adder or an approximate multiplier is selected. This design is evaluated in terms of approximate error metrics, such as average case error, or worst-case error etc. Then, in the third step, the effect of the error induced by the approximate component on the computational accuracy of the system is evaluated. Consider for instance an approximate adder. Knowing the worst-case error of the approximate adder, it is possible to directly conclude a bound for its accuracy. For this purpose, classical numeric error analysis can be used. Once the computational accuracy for the relevant operations is known, its propagation to the application specific error norms can be calculated (6th step). Finally, the designer has to decide whether the result is good enough: Therefore, the error e is evaluated in the application specific error norm and compared to a application specified error bound B. If the result meets the bound B and is close enough to B, then a suitable solution based on the in the first step selected approximated component(s) has been found. However, the circuit may still be sub-optimal, since there might be a better solution. Even worse, the normal case is that either the bound B is violated or the result is too far away from the tolerated deviation; in both cases the complete flow has to be repeated in a long and costly loop. In practice, several iterations appear. To overcome this problem we reverse this flow as presented in the next subsection.

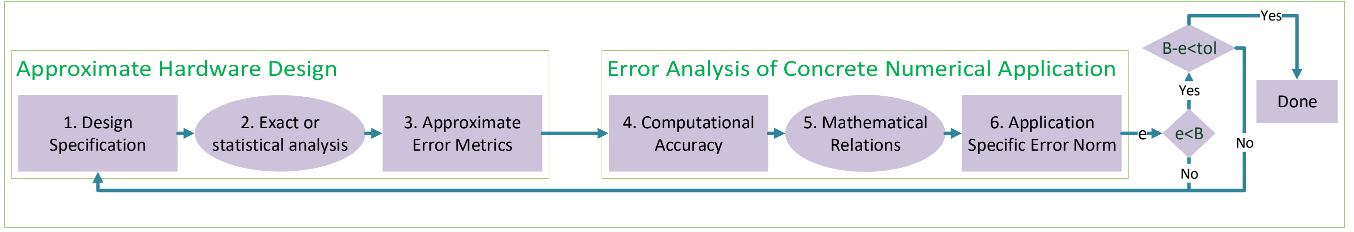
In addition, statistical design flows are being used. Parameters of the approximate hardware are being adjusted by using it on a fixed set of problems and evaluating the resulting error induced by approximation. However such a design flow can not guarantee, that the error bound holds for every possible input combination, if the input space is large.

B. Proposed Reversed Approximate HW Design Flow

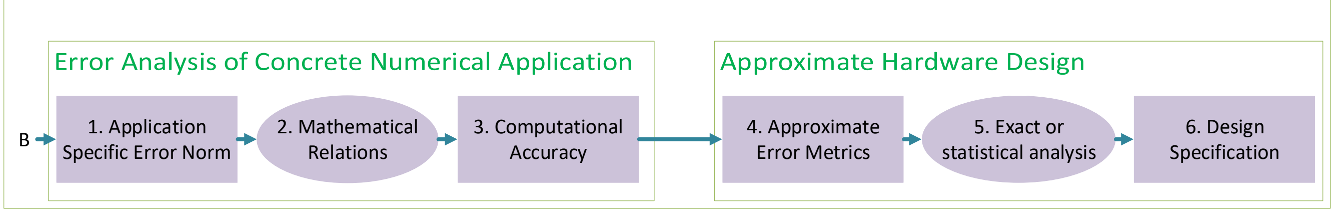
We propose to reverse the conventional approximate HW design flow for numeric applications as shown in Fig. 1(b). Instead of starting with the design of the approximate HW, we start with the application specific error bound. We reverse the mathematical analysis by calculating the required computational accuracy in order to guarantee that the calculated error of the final result is below the given application specific error bound B. We relate the required computational accuracy to a suitable approximate error metric. Since this allows to control the valid maximum deviation from the correct value in terms of the approximate error metric, we are able to directly design the approximate HW which fits the requirements. Hence, no loops are necessary and the resulting approximate HW layout is guaranteed to be correct by construction wrt. the application specific error bound and exploiting the full potential of the approximate HW.

IV. LU-Factorization Basics, Floating-Point Arithmetic and Matrices

This section reviews the basic knowledge of the LU-Factorization, accuracy in the context of floating-point numbers and matrix operations, which is necessary to understand the remainder of this paper. It also introduces notations



(a) Conventional approximate HW design flow for numerical applications



(b) Proposed reversed approximate HW design flow for numerical applications

Fig. 1. Design flows

used throughout the paper. First, we give a summary of the LU-Factorization. Afterwards we introduce brief definitions of matrix operations, which are necessary to know, in order to understand the relations between equations, which are explained in this paper. Finally, we introduce notations used throughout this paper, including the unit round off and a model for the rounding errors induced by the usage of floating-point arithmetic.

A. LU-Factorization

The LU-Factorization of a matrix A is commonly defined as $A = LU$ with L being a lower triangular matrix with unit diagonal elements and U being an upper triangular matrix.

When solving a linear system of equations $Ax = b$ for a non singular matrix A with a given LU-Factorization $A = LU$ and a given right-hand side b, the resulting equation is

$$LUx = b. \tag{1}$$

In the following we give an example on how to solve a linear system of equations using the LU-Factorization.

Matrix A and vector b are defined by Eq. 2. The corresponding LU-Factorization of A is given by Eq. 3. To solve $LUx = b$, we substitute $y = Ux$, such that the remaining equation is $Ly = b$. We formulate this problem in Eq. 4a and solve it in Eq. 4b. The remaining problem is to solve $Ux = y$, which is formulated in Eq. 5a and solved in Eq. 5b. The result is given in Eq. 6.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 8 & 11 \\ 3 & 22 & 35 \end{pmatrix}, \quad b = \begin{pmatrix} 14 \\ 51 \\ 152 \end{pmatrix} \tag{2}$$

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} \tag{3}$$

$$\begin{matrix} L & y & = & b \\ \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{pmatrix} & \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} & = & \begin{pmatrix} 14 \\ 51 \\ 152 \end{pmatrix} \end{matrix} \tag{4a}$$

$$\begin{aligned} y_1 &= 14 \\ y_2 &= 51 - 2y_1 = 23 \\ y_3 &= 152 - 3y_1 - 4y_2 = 18 \end{aligned} \tag{4b}$$

$$\begin{matrix} U & x & = & y \\ \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} & \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} & = & \begin{pmatrix} 14 \\ 23 \\ 18 \end{pmatrix} \end{matrix} \tag{5a}$$

$$\begin{aligned} 6x_3 &= 18 & &\Rightarrow x_3 = 3 \\ 4x_2 &= 23 - 5x_3 = 8 & &\Rightarrow x_2 = 2 \\ x_1 &= 14 - 2y_2 - 3y_3 = 1 & &\Rightarrow x_1 = 1 \end{aligned} \tag{5b}$$

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{6}$$

One can observe that the calculations mainly consist of multiplications and subtractions. While the number of divisions is linear in the dimension of the problem, the number of subtractions and multiplications grows quadratic.

B. Matrix Operations

Before we take a closer look at LU-Factorization specific error norms, we need to introduce some basic matrix operations, which can be found in literature, for example in [24]. We will utilize them to reverse the mathematical error analysis of the LU-Factorization.

An induced matrix norm is defined by:

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}, \quad A \in \mathbb{C}^{m \times n}, x \in \mathbb{C}^n. \tag{7}$$

There are also matrix norms which are not induced (not related to a vector norm), but these are not relevant for this paper. Whenever we use the expression matrix norm, we mean an induced matrix norm.

One can interpret a matrix norm in the following way: Imagine a vector norm denoting the length of a vector. A

matrix norm denotes how much it stretches the vector. The fraction $\frac{\|Ax\|}{\|x\|}$ can be interpreted as the magnification of x that takes place, as x is multiplied with A. Take rotation matrices as an example. These matrices generally have a norm of 1, as they do not stretch the multiplied vectors in anyway, but rotate them around the origin.

Matrix norms have the property of being submultiplicative. Submultiplicativity for matrix norms is defined as:

$$\|AB\| \leq \|A\| \, \|B\| \, , \quad A \in \mathbb{C}^{m \times n}, B \in \mathbb{C}^{n \times p}. \tag{8}$$

C. Notations

In order to distinguish between a computation in exact arithmetic and arithmetic using approximation, we need to introduce additional notations.

If an operator $\odot$ with $\odot \in \{+, -, *, /\}$ is used, we refer to exact arithmetic. When using approximation to calculate the result of an operation, we use the notation $(\odot)_a$. So for example $\left(\frac{x}{y}\right)_a$ would give an approximation for the exact result of $\frac{x}{y}$.

Throughout this paper we use the notation of $|A|$ denoting the matrix of absolute values of A. By matrix of absolute values we mean a matrix consisting of the absolute values of each entry of A. Thus if $A = (a_{i,j}), i = 1 \dots n, j = 1 \dots m$, then $|A| = (|a_{i,j}|), i = 1 \dots n, j = 1 \dots m$. A comparison between matrices is always point-wise.

Since we want to introduce approximation while using floating-point arithmetic (for more information on the definition of floating-point arithmetic see [25]), we need to define an upper bound for the error, which is introduced due to the usage of floating-point arithmetic. A common model for the rounding error of a number $\alpha \in \mathbb{R}$ when it is represented in floating-point arithmetic is $(\alpha)_a = \alpha(1 + \delta_\alpha), \delta_\alpha \in \mathbb{R}$.

The upper bound for all δ and any operation $\odot$ with operands α and β is called the *unit round off* μ. Thus $\forall \alpha, \beta \in \mathbb{R}$ and all $\odot$ holds:

$$(\alpha \odot \beta)_a = (\alpha \odot \beta)(1 + \delta_{\alpha \odot \beta}), \quad |\delta_{\alpha \odot \beta}| \leq \mu, \tag{9}$$

see for example [16], [18]. μ is a measure for the worst-case computational accuracy achieved by executing a single operation in floating-point arithmetic.

V. Error bounds for the LU-Factorization in floating-point arithmetic

This section is structured as follows: Subsection V-A introduces general error norms for the solving of linear systems of equations, which are of interest for different applications. Subsection V-B derives an upper bound for one of them. Finally, Subsection V-C uses this upper bound to derive bounds for the unit round off μ.

A. Error Norms

Let x be the solution to

$$Ax = b \tag{10}$$

and $\hat{x}$ the solution to the disturbed system

$$\underbrace{(A + \Delta A)}_{=:\hat{A}} \hat{x} = b, \tag{11}$$

where $\Delta A \in \mathbb{R}^{m \times m}$ is the disturbance and $A \in \mathbb{R}^{m \times m}$, $\hat{A} \in \mathbb{R}^{m \times m}$, $b \in \mathbb{R}^m$, $x \in \mathbb{R}^m$, $\hat{x} \in \mathbb{R}^m$ and A and $\hat{A}$ are regular.

One is usually interested in the forward or the backward error induced by the disturbance. So in how large ΔA may at most be, such that either

$$\epsilon(\hat{x}) = \frac{\|\Delta x\|}{\|x\|}, \quad \Delta x = \hat{x} - x \quad \text{(forward error)},$$

which we call the forward error, or

$$\rho(\hat{x}) := \frac{\|b - A\hat{x}\|}{\|A\| \, \|\hat{x}\|} \quad \text{(backward error)}, \tag{12}$$

which we call backward error, is smaller than a given bound B for a given norm $\|\cdot\|$ [18], [26].

These error norms can be interpreted as follows: The forward error is a measure for relative size of the error Δx calculated in the result $\hat{x}$. It limits how large the error in the calculated result may become in relation to the size of the real result x of the equation. A forward error of 0.1 would mean that the norm of the error Δx of the calculated result is 10% of the size of the norm of the real result x.

The backward error is not a measure for the error in x, but a measure for how far the problem we actually solved is away from the problem we initially wanted to solve. It quantifies how well the data of the problem (i.e. the entries of A) is measured. From an application point of view, a bound for the backward error means, that the user is satisfied with a solution of a problem for some data that lie within a certain uncertainty range of the measured data [27]. We focus on the backward error in this paper.

ΔA may be induced due to the approximations performed during the calculation of the LU-Factorization of A and/or the limited accuracy when solving Eq. 1. We assume that the original problem (Eq. 10) is well enough defined, such that the result can be calculated with sufficient accuracy, when machine accuracy is used.

The remainder of this section is structured as follows: In Subsection V-B, we derive a general upper bound for the backward error. Subsection V-C uses this bound to determine limits for the unit round off μ as defined by Eq. 9, which can be used in Section VI to determine limits for the needed accuracy of a floating-point multiplier.

B. Influence of Disturbances on the Results

Lemma 1. *Let $A, \Delta A \in \mathbb{C}^{n \times n}$ and A be non singular. Let further $\|\cdot\|$ denote a norm. Let x be the solution to $Ax = b$ and $x + \Delta x$ be the solution to $(A + \Delta A)(x + \Delta x) = b$. Then*

$$\frac{\|\Delta x\|}{\|\hat{x}\|} \leq \kappa(A) \frac{\|\Delta A\|}{\|A\|},$$

with $\kappa(A) = \|A\| \, \|A^{-1}\|$ denoting the condition number of A.

Proof. The equation is well known. See for example [24], Theorem 2.3.3. □

It is important to note, that $\frac{\|\Delta x\|}{\|\hat{x}\|}$ is an upper bound for the backward error as defined by Eq. 12:

$$\rho(\hat{x}) := \frac{\|b - A\hat{x}\|}{\|A\| \, \|\hat{x}\|} = \frac{\|\overbrace{b - A(x}^{=b} + \Delta x)\|}{\|A\| \, \|\hat{x}\|} = \frac{\|A\Delta x\|}{\|A\| \, \|\hat{x}\|} \leq \frac{\|A\| \, \|\Delta x\|}{\|A\| \, \|\hat{x}\|} = \frac{\|\Delta x\|}{\|\hat{x}\|} \tag{13}$$

This enables us to use Lemma 1 for our calculations, since a bound for $\frac{\|\Delta x\|}{\|\hat{x}\|}$ results in a bound for $\rho(\hat{x})$.

C. Equation solving with Limited Accuracy

Solving a system of equations Ax for any right side b will introduce additional errors, if we use approximated operations. In order to solve Eq. 10, we need to first solve

$$Ly = b \tag{14}$$

and subsequently

$$Ux = y. \tag{15}$$

Solving the systems in Eq. 14 and 15 in floating-point arithmetic introduces the following errors, if $A \in \mathbb{R}^{n \times n}$ and $n\mu < 1$ (which is usually the case):

$$(A + \Delta A)\hat{x} = b$$
$$|\Delta A| \leq \gamma_n(1 + \gamma_n)|L||U|, \tag{16}$$

with

$$\gamma_n = \frac{n\mu}{1 - n\mu} \Rightarrow \mu = \frac{\gamma_n}{n + n\gamma_n}, \tag{17}$$

see [20].

The proof of Eq. 16 would require a lot of additional effort in this context, so instead, we try to illustrate it briefly: $\gamma_n |T|$ can be interpreted as a bound for the error of the result achieved, when solving a linear system of equations T of triangular form using substitution (see [18], Theorem 8.5). When solving $Ax = b$ using the LU-Factorization, we solve the two triangular systems of equations L and U. We take the result of the first triangular system as the right-hand side of the second triangular system. This is the quadratic part of the factor of the error bound (γ_n^2). The solving of the second triangular system adds an additional error, which itself is bounded by γ_n. Thus we have $\gamma_n + \gamma_n^2 = \gamma_n(1 + \gamma_n)$.

We can combine Lemma 1 and Eq. 16 to get an upper bound for the backward error as defined in Eq. 12 depending on γ_n. Due to the definition of matrix norms, it is obvious that $\|\Delta A\| \leq \||\Delta A|\|$ and thus we can conclude that

$$\frac{\|\Delta x\|}{\|\hat{x}\|} \leq \kappa(A)\frac{\gamma_n(1 + \gamma_n)\||L||U|\|}{\|A\|}. \tag{18}$$

Eq. 18 can easily be solved for γ_n:

$$\gamma_n \geq -\frac{1}{2} + \sqrt{\frac{1}{4} + \frac{\|\Delta x\|}{\|\hat{x}\|}\frac{\|A\|}{\kappa(A) \, \||L| \, |U|\|}} \tag{19}$$

Since we do not want the resulting error to be larger than allowed, we choose γ_n in such a way that equality is given for Eq. 19.

By combining Eq. 19 and Eq. 17, we can calculate μ depending on the inputs:

$$\boxed{\mu = \frac{-\frac{1}{2} + \sqrt{\frac{1}{4} + \frac{\|\Delta x\|}{\|\hat{x}\|}\frac{\|A\|}{\kappa(A)\||L||U|\|}}}{n + n\left(-\frac{1}{2} + \sqrt{\frac{1}{4} + \frac{\|\Delta x\|}{\|\hat{x}\|}\frac{\|A\|}{\kappa(A)\||L||U|\|}}\right)}} \tag{20}$$

The only unknown on the right-hand side of this equation is the fraction $\frac{\|\Delta x\|}{\|\hat{x}\|}$. Since the value used for it is an upper bound for the resulting backward error (see Eq. 13), we can simply set it to the value given for the maximum tolerable backward error and be sure, that the backward error of the result will not exceed the given bound, if the computational error is less or equal to μ.

VI. Approximate floating-point multiplier

Section V gives us a rule on how to choose the unit round off μ according to the inputs L and U in the form of Eq. 20.

We need to introduce a way on how to determine the number of relevant mantissa bits depending on μ.

An approximate floating-point multiplier as proposed by [4] allows us to ignore mantissa bits of the bit representation of the operands of a multiplication. Let the length of the bit representation of the mantissa be m and the approximation level be l. By approximation level l we mean, that the bit length of the mantissa of the bit representation of each operand of a multiplication is shortened by l bits, leaving a mantissa length of $m - l$ bits. Higham gives detailed information about the standard model for floating-point operations in [18]. It is stated, that operations are to be performed as if they were calculated with infinite precision and then the unit roundoff μ is to be applied. Let $\alpha, \beta \in \mathbb{R}$. Then according to [18], Theorem 2.2 their floating-point representation $(\alpha)_a$ and $(\beta)_a$ is given by

$$(\alpha)_a = \alpha(1 + \delta_\alpha), \qquad (\beta)_a = \beta(1 + \delta_\beta).$$

Let $\delta = \max(|\delta_\alpha|, |\delta_\beta|)$ be an upper bound for δ_α and δ_β, then a bound for the error of the product of $(\alpha)_a$ and $(\beta)_a$ induced by floating-point arithmetic is given by

$$|(\alpha)_a(\beta)_a - \alpha\beta| \leq |\alpha\beta(1 + 2\delta + \delta^2) - \alpha\beta|.$$

Since this bound is reached if $\alpha = \beta$, we can conclude that $\mu \geq 2\delta + \delta^2$. Solving the equation for δ gives

$$\boxed{\delta \leq -1 + \sqrt{1 + \mu}}. \tag{21}$$

μ can be determined by Eq. 20.

We can now calculate an upper bound δ using Eq. 21 and transform it directly to the level of inaccuracy we can tolerate in our approximate multiplier, according to [18] p.39, a general bound for δ is given by

$$\delta \leq \frac{1}{2}b^{1-m},$$

with b being the used radix. If we shorten the length of the mantissa by l bits, it changes to

$$\delta \leq \frac{1}{2}b^{1-(m-l)}$$

and thus

$$\boxed{l \geq \log_b 2\delta + m - 1}. \tag{22}$$

Eq. 22 enables us to calculate how many mantissa bits of the operands of a multiplication can be ignored in order to be sure that the rounding error of the result will be bounded by μ. δ can be calculated using Eq. 21, while μ is calculated using Eq. 20 for any given bound for the backward error.

VII. Experimental Evaluation

In this section, we present the experimental evaluation on real life problems. For approximation, we use as explained before an approximate floating-point multiplier which ignores the last l mantissa bits of the operands. Based on this approximation scheme, we calculate the level of approximation with our proposed reversed flow, i.e. based on the application specific error bound we determine the number of mantissa bits to be dropped for each problem. To demonstrate the effectiveness of our approach we show the speedup achieved by approximation compared to the non-approximated calculation.

A. Experimental Setting and Experimental Procedure

There is a large collection of matrices appearing in different applications called *SuiteSparse Matrix Collection* [28]. We use a subset of this collection which consists of square matrices of different sizes, and use GNU Octave© 4.0.2 to calculate their LU-Factorization. In addition, we determine the feasible values for the approximation level l using the equations from our three main results of Section VI (visualized by boxes). We use the Euclidean norm in our experiments. Right-hand sides are computed by multiplying the matrices with a constant vector filled with ones. Finally, we utilize the approximate floating-point multiplier to solve the equation using the precalculated LU-Factorization.

A ZedBoard™ and Xilinx Vivado© 2016.3 are used to realize the approximate floating-point multiplier which is written in VHDL. First, we design our HW to calculate the results using full machine accuracy. Afterwards we calculate the necessary computational accuracy for each matrix to get a backward error below different error bounds, which are $0.01, 0.10, 0.30$ and 0.50. We then redesign our HW according to the calculated required computational accuracy using the relations developed in the previous section.

An intuitive interpretation of the values chosen as error bounds is as follows: Recall that we actually bound $\frac{\|\Delta x\|}{\|\hat{x}\|}$ since this is an upper bound for the backward error (see Eq. 13). Limiting this to 0.01 means that the norm of the error in the calculated result cannot be larger than 1% of the norm of the calculated result itself. This explanation is for general norms. Now to give a concrete example, consider the ∞-norm (this norm just finds the largest entry in a given vector): Limiting $\frac{\|\Delta x\|_\infty}{\|\hat{x}\|_\infty}$ to 0.01 means that the largest deviation in the calculated result vector may not be larger than 1% of its largest entry.

In order to use the approximate floating-point multiplier, we first need to calculate how many mantissa bits l may be dropped with respect to each application specific error bound for each problem. After calculating the feasible unit round off μ which is an upper bound for the worst-case computational accuracy, using Eq. 20 and subsequently δ based on Eq. 21, we round the result l of Eq. 22 down to the nearest integer.

B. Experimental Results

Table I summarizes the number of dropped bits l for different application specific error bounds. The first column denotes the name of the benchmark problem. The second column gives the dimension and the third column the condition number κ of the corresponding matrix. Since all matrices are square, we only give the size of one dimension in the second column of the table. The remaining columns give the feasible values for l for each error bound 0.01, 0.10, 0.30, 0.50, respectively.

As can be seen in Table I larger matrices and matrices with higher condition numbers tend to have smaller feasible values for l and thus allow less approximation. This was expected due to the form of Eq. 20. The size of the matrix n and the condition number κ are both part of the denominator and the larger these values become, the smaller the resulting unit round off μ will be. Furthermore, larger matrices tolerate fewer errors, since they propagate and accumulate with each step of solving the systems L and U. Looking at Lemma 1 it becomes clear that in general matrices with high condition numbers tend to allow a smaller disturbance ΔA which is induced by our approximation during the solving process.

TABLE I
Number of dropped mantissa bits l (approximation level) for different application specific error bounds

Benchmark	size	condition number	Dropped bits l for error bounds 0.01	0.10	0.30	0.50
tols2000	2000	5991449.6	11	15	16	17
G2	800	4600.5	13	16	17	18
G3	800	2556.1	13	17	18	19
steam2	600	3783125.2	14	17	19	19
tols1090	1090	1831083.5	14	17	19	20
ukerbe1_dual	1866	7451.0	15	18	19	20
tols340	340	203453.9	19	22	24	24
wang2	2903	23055.4	19	22	24	25
wang1	2903	20323.0	20	22	24	25
Trefethen_2000	2000	15517.6	20	23	25	26
str_200	363	15106.1	22	25	27	28
str_400	363	6430.7	23	27	28	29
tols90	90	20232.0	24	27	29	30
Trefethen_200	200	1090.7	27	30	32	33
Trefethen_20	20	63.1	35	38	39	40
Tina_AskCog	11	19.8	35	39	40	41

To evaluate the efficiency of our approach, we compare the run-time needed to calculate the results using floating-point operations with double precision (golden non-approximated results; cf. [25]) to the run-time needed to solve the problems using the proposed reversed approximate HW design flow. Table II gives information about the run-time saved by approximation. Again, the first column denotes the name of the benchmark. The second column shows the run-time needed when using a floating-point multiplier with double precision. The remaining four columns give information about

TABLE II
COMPARISON BETWEEN THE COMPUTATION TIMES FOR DIFFERENT ERROR BOUNDS

Benchmark	double precision [ms]	speedup in [%] for error bounds 0.01	0.10	0.30	0.50
tols2000	193.740	<0.1	0.3	0.2	0.3
G2	1189.331	<0.1	4.8	9.7	9.7
G3	1198.436	<0.1	9.7	9.7	9.7
steam2	77.842	3.7	7.7	7.8	7.8
tols1090	60.949	0.4	0.7	0.4	0.8
ukerbe1_dual	602.988	7.1	7.2	7.1	7.1
tols340	9.294	4.0	3.9	2.7	2.7
wang2	2740.468	8.4	8.5	5.1	8.5
wang1	2740.786	8.5	8.6	5.1	8.5
Trefethen_2000	5433.548	9.6	14.1	9.5	14.1
str_200	16.465	5.9	5.9	5.9	8.8
str_400	18.088	9.3	6.4	9.4	16.4
tols90	3.713	6.5	10.4	23.4	18.6
Trefethen_200	59.179	9.7	20.1	24.7	20.2
Trefethen_20	0.654	30.9	29.4	37.5	40.4
Tina_AskCog	0.090	21.1	27.8	30.0	30.0

the percentage of time saved when setting the error bound to $0.01, 0.10, 0.30$ and 0.50, respectively. The values for l as given in Table I are used, such that the l least significant bits of the operands are dropped by the approximate floating-point multiplier.

It can be seen that increased usage of approximation results in lower run-times. Allowing a backward error of 0.01 has reduced the run-time by more than 30% for the problem *Trefethen_20*. The amount of reduction achieved by approximation does not solely dependent on the approximation level. Problem *str_200* allowed an approximation level of 22 for the error bound 0.01 and achieved a speedup of 5.9%, while problem *ukerbe1_dual* could only tolerate an approximation level of 15, but achieved a speedup of 7.1% for the same error bound.

This has to be due to the nature of the problem itself. If we compare *tols2000* to *Trefthen_2000*, then we can recognize that almost no speedup was achievable due to approximation for *tols2000*, while for *Trefthen_2000* a notable speedup could already be achieved for small error bounds, even though both matrices have the same size. The LU-Factorization of *tols2000* is sparse, which results in many multiplications being skipped due to one factor being 0, while both, matrix L and matrix U, of the LU-Factorization of *Trefthen_2000* have about three orders of magnitude more non zero entries than the LU-Factorization of *tols2000*. This results in many more multiplications compared to other operations, making the usage of an approximate multiplier a lot more efficient. As a rule of thump for the efficiency of the usage of an approximate multiplier, one could compare the computation time of solving a problem to its size, when using full accuracy. If there are many zero entries in the LU-Factorization, then the computation time will be low, because little operations are needed. If there are many non zeros, then many multiplications are needed and the computation time rises, which can be reduced significantly by the use of an approximate multiplier.

VIII. CONCLUSION

In this paper we have proposed to reverse the typical design flow for approximate hardware. In the reversed flow we start with the application and determine the required computational accuracy such that the computational error of the result is below the application specific error bound. In contrast to the result of the traditional HW design flow, the used approximate HW is correct by construction.

We have demonstrated the new reversed flow for a first application, i.e. LU-Factorization. In the experiments we were able to demonstrate that a significant speed-up for solving linear equations can be achieved while guaranteeing the application specific error bound. As a consequence, the user can chose the accuracy required for the concrete application context.

REFERENCES

[1] N. Zhu, W. L. Goh, and K. S. Ye, "An enhanced low-power high-speed adder for error-tolerant application," in *ISIC*, 2009, pp. 69–72.
[2] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*. New York, NY, USA: ACM, 2012, pp. 820–825.
[3] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *DAC*, 2015, pp. 86:1–86:6.
[4] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," in *TVLSI*, vol. 8, No. 3, Jun. 2000, pp. 273–285.
[5] C.-H. Lin and I.-C. Lin, "High accuracy approximate multiplier with error correction." in *ICCD*, 2013, pp. 33–38.
[6] S. Venkataramani, A. Sabne, V. J. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: systematic logic synthesis of approximate circuits," in *DAC*, 2012, pp. 796–801.
[7] S. Froehlich, D. Große, and R. Drechsler, "Error bounded exact BDD minimization in approximate computing," in *ISMVL*, 2017, pp. 254–259.
[8] ——, "Approximate hardware generation using symbolic computer algebra employing Gröbner basis," in *DATE*, 2018, pp. 889–892.
[9] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of AIGs for error tolerant applications," in *ICCAD*, 2016.
[10] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *MICRO*, 2003.
[11] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Designing a processor from the ground up to allow voltage/reliability tradeoffs," in *(HPCA-16)*, Jan. 2010, pp. 1–11.
[12] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "MACACO: modeling and analysis of circuits for approximate computing," in *ICCAD*, Nov. 2011, pp. 667–673.
[13] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Precise error determination of approximated components in sequential circuits with model checking," in *DAC*, 2016.
[14] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, "BDD minimization for approximate computing," in *ASP-DAC*, 2016, pp. 474–479.
[15] H. Anton and C. Rorres, *Elementary Linear Algebra: Applications Version*, 11th ed. John Wiley & Sons, 2014.
[16] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994.
[17] ——, *The Algebraic Eigenvalue Problem*. New York, NY, USA: Oxford University Press, Inc., 1988.
[18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.
[19] ——, "How accurate is gaussian elimination?" Department of Computer Science, Cornell University, Ithaca, NY, USA, Tech. Rep., 1989.
[20] C. de Boor and A. Pinkus, "Backward error analysis for totally positive linear systems," *Numerische Mathematik*, vol. 27, no. 4, pp. 485–490, 1976.
[21] F. Stummel, "Perturbation theory for evaluation algorithms of arithmetic expressions," *Mathematics of Computation*, vol. 37, no. 156, pp. 435–473, 1981.
[22] ——, "Forward error analysis of gaussian elimination," *Numer. Math.*, vol. 46, no. 3, pp. 365–395, Sep. 1985.
[23] Q. Zhang, F. Yuan, R. Ye, and Q. Xu, "Approxit: An approximate computing framework for iterative methods," in *DAC*. New York, NY, USA: ACM, 2014, pp. 97:1–97:6.
[24] D. S. Watkins, *Fundamentals of Matrix Computations*. New York, NY, USA: John Wiley & Sons, Inc., 1991.
[25] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE Computer Society, Aug. 2008.
[26] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 4th ed. J. Hopkins Uni. Press, 2013.
[27] S. Gratton, P. Jiránek, and X. Vasseur, "Energy backward error: interpretation in numerical solution of elliptic partial differential equations and behaviour in the conjugate gradient method." CERFACS, Tech. Rep., 2013.
[28] T. Davis, "The suitesparse matrix collection," http://www.cise.ufl.edu/research/sparse/matrices/.