Digit Elision for Arbitrary-accuracy Iterative Computation

He Li, James J. Davis, John Wickerson and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London, London, SW7 2AZ, United Kingdom
{h.li16, james.davis, j.wickerson, g.constantinides}@imperial.ac.uk

***Abstract*—We recently proposed the first hardware architecture enabling the iterative solution of systems of linear equations to accuracies limited only by the amount of available memory. This technique, named ARCHITECT, achieves exact numeric computation by using online arithmetic to allow the refinement of results from earlier iterations over time, eschewing rounding error. ARCHITECT has a key drawback, however: often, many more digits than strictly necessary are generated, with this problem exacerbating the more accurate a solution is sought. In this paper, we infer the locations of these superfluous digits within stationary iterative calculations by exploiting online arithmetic's digit dependencies and using forward error analysis. We demonstrate that their lack of computation is guaranteed not to affect the ability to reach a solution of any accuracy. Versus ARCHITECT, our illustrative hardware implementation achieves a geometric mean 20.1× speedup in the solution of a set of representative linear systems through the avoidance of redundant digit calculation. For the computation of high-precision results, we also obtain an up-to 22.4× memory requirement reduction over the same baseline. Finally, we demonstrate that solvers implemented following our proposals can show superiority over conventional arithmetic implementations by virtue of their runtime-tunable precisions.**

1. Introduction & Motivation

Consider a numerical program whose result is obtained by iteratively calculating a sequence of approximations of the form $\boldsymbol{x}^{(k+1)} = f\big(\boldsymbol{x}^{(k)}\big)$, where $f \in \big(\mathbb{R}^N \to \mathbb{R}^N\big)$ is a computable real function. For demonstration, assume we wish to solve the toy equation

$$x^{(k+1)} = 1/8 - x^{(k)}/7$$

for x from $x^{(0)} = 0$. This is a unidimensional example of the Jacobi method, a classical stationary iterative algorithm.

Each approximant is conventionally generated starting from the least-significant digit (LSD), leading to a computation pattern such as that shown in Figure 1a. A key shortcoming with this conventional method is that the numerical precision must be fixed beforehand. Choosing the right precision is non-trivial, particularly with respect to hardware implementation. If it is too high, the circuit may be unnecessarily slow and power-consuming, while if it is too low, the criterion for convergence may never be reached.

In our previous work, ARCHITECT [1]—the first method for implementing iterative computations to arbitrary accuracies in hardware—we addressed this issue. ARCHITECT employs online arithmetic, in which approximants are calculated starting from the most-significant digit (MSD). Approximants can thus be calculated concurrently, *e.g.* following the digit pattern shown in Figure 1b. Crucially, precision does not need to be fixed *a priori*; the zig-zag pattern advances deeper into the iteration-precision space until a satisfactorily accurate result is obtained.

Unfortunately, the ARCHITECT method is inefficient because the triangular shape traced out involves the computation of many more digits than are actually needed. In the bottom-left corner lie high-significance digits of later approximants; these generally become stable over time, so we call them *don't-change* digits. In the top-right corner lie low-significance digits of early approximants; these are generally unimportant, thus we call them *don't-care* digits.

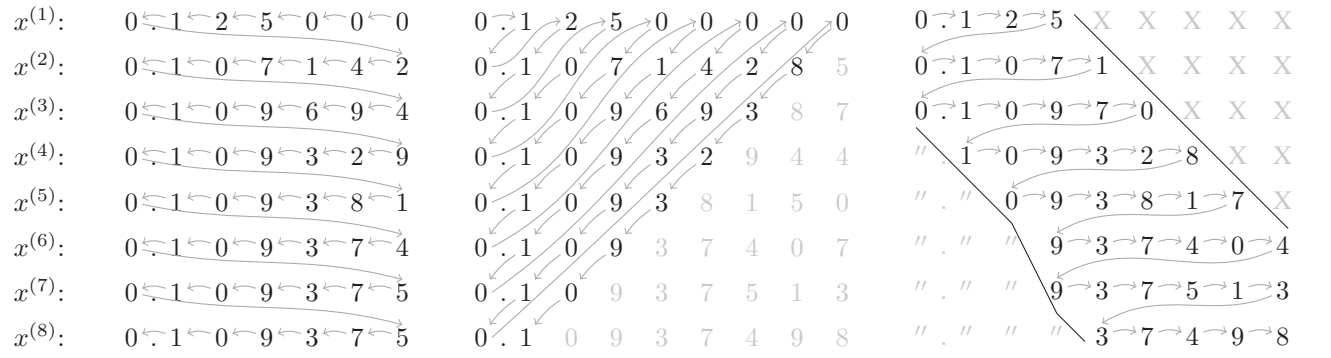
In this paper, we propose a novel method for implementing stationary iterative calculations to arbitrary accuracies that refines the ARCHITECT technique by avoiding the unnecessary computation of these don't-change and don't-care digits, arriving at a digit pattern such as that shown in Figure 1c. We make the following novel contributions:

- Theoretical analysis of MSD stability within any online arithmetic-implemented iterative method, facilitating runtime detection of don't-change digits.
- A theorem for the optimal rate of LSD growth per iteration within stationary iterative methods, thereby enabling the preclusion of don't-care digit computation. With the appropriate preconditions, this is proven to have no bearing on the chosen method's ability to reach a solution of any accuracy.
- An exemplary hardware implementation of our proposal using the Jacobi method.
- Performance evaluations of our demonstrative architecture, drawing comparison against ARCHITECT, the state-of-the-art arbitrary-precision hardware iterative solver, and conventional fixed-precision equivalents. We observe a mean 12.8× reduction in the number of digits generated to solve a set of representative linear systems, showing a geometric mean 20.1× speedup over ARCHITECT.

2. Background

2.1. Custom-precision Arithmetic

Applications requiring very high precisions have become increasingly popular in recent years [2]. For exam-



(a) Conventional arithmetic with approximants calculated LSD first.

(b) ARCHITECT, MSD first.

(c) Our proposal, MSD first, ignoring don't-change and don't-care digits.

Figure 1: Alternative digit-calculating strategies for the solution of $x^{(k+1)} = 1/8 - x^{(k)}/7$. Arrows show the order of digit generation. In Figure 1c, $''$-marks indicate don't-change and Xes don't-care digits, with solid lines representing the bounds of these two regions.

ple, today, hundreds of digits of precision are required in atomic system simulations and electromagnetic scattering theory calculations, while Ising integrals and elliptic function evaluation need thousands of digits [3]. In experimental mathematics, Poisson equation computations frequently require results to tens or hundreds of thousands of digits of precision [4]. Standard numeric datatypes, such as double- or even quadruple-precision floating point, are no longer sufficient in an increasing number of scenarios.

Interest in the hardware acceleration of high-precision operations, in particular those within iterative algorithms, is growing [5]. Field-programmable gate arrays (FPGAs) represent ideal platforms for their realisation thanks to their flexible fabrics, devoid of the costs and lead times associated with full-custom implementation. Many open-source libraries, *e.g.* FloPoCo [6] and VFLOAT [7], alongside those provided by vendors, are available for custom-precision arithmetic hardware generation. Mixed-precision iterative solvers, in which precisions can be selected from a set as required at runtime, have also been proposed [8], [9].

Each of the aforementioned proposals requires precision—or precisions—to be determined *a priori*. In many cases, this is not a trivial task; making the wrong choice often means having to throw the calculations already done away and starting from scratch with higher precision, wasting both time and energy in doing so. In our work, we are particularly interested in hardware architectures which allow precision to be increased over time without having to restart computation or modify the circuitry.

2.2. Online Arithmetic

Achieving arbitrary-precision computation with fixed hardware requires MSD-first input consumption and output generation. A suitable proposal for this, widely discussed in the literature, is *online arithmetic* [10]. By employing redundancy in their number representations, all online operators are able to function in MSD-first fashion. Online operators are classically serial, however efficient digit-parallel (unrolled) implementations targetting FPGAs have been developed as well [11]. We make use of both digit-serial and -parallel online operators in this work, employing the *de facto* standard signed-digit number representation.

Of particular significance to the material presented in this paper is the concept of *online delay*. When performing an online operation, the digits of its result are generated at the same rate as its input digits are consumed, but the result is delayed by a fixed number of digits, denoted δ. That is, the first (*i.e.* most-significant) q digits of an operator's result are wholly determined by the first $q + \delta$ digits within each of its operands [10]. The value of δ is operator-specific, but is typically a small integer; for radix 2, delays of online addition and multiplication are $\delta_+ = 2$ and $\delta_\times = 3$, respectively. When chaining operators to form a datapath, its total online delay, δ_Σ, is the highest cumulative delay through the complete circuit [12].

2.3. ARCHITECT: Arbitrary-precision Constant-hardware Iterative Compute

Arbitrary-precision online operators were realised by Zhao *et al.* [12], who wished to allow runtime tuning of precision for iterative calculations. Of course, precision is not the only factor affecting the accuracy of an iterative algorithm's result; the number of iterations performed, K, is also crucial. Where Zhao *et al.*'s proposal required separate hardware to be instantiated for each iteration, ARCHITECT removed the need to determine, and fix, K at compile time. Therein, we presented the first iterative solver that allowed both precision *and* iteration count to be tuned at runtime; the accuracy of results that can be obtained is limited only by the size of the available memory.

Zhao *et al.*'s designs and ARCHITECT both achieved arbitrary-precision functionality by breaking arbitrary-precision numbers into fixed-precision 'chunks' and processing them sequentially. We use the ARCHITECT notation in this paper, regarding a p-digit number as comprising $n = \lceil p/U \rceil$ chunks, each U digits wide. The chunks, and the digits within each chunk, are indexed—from most to least significant—using the variables $c \in \{0, 1, \cdots, n-1\}$ and $u \in \{0, 1, \cdots, U-1\}$, respectively. Chunk width U determines the size of parallel operators used within more

complex units—*e.g.* adders in the case of multipliers—and the width of the required storage elements. Finally, approximants are indexed with $k \in \{0, 1, \cdots, K-1\}$, with $k = 0$ indexing the initial guess of the solution being sought.

Simultaneous growth of k and p was achieved for ARCHITECT by using a Cantor pairing function (CPF) [13] to collapse the three dimensions k, c and u into the two-dimensional tuple $(\text{CPF}(k, c), u)$. This is suitable for addressing memory since $\text{CPF}(k, c)$ can grow without bound while, by design, $0 \leq u < U$.

As exemplified in Figure 1b, while the generation of all LSDs shown is strictly necessary to achieve exact computation in every iteration, as ARCHITECT does, iterative algorithms do not ordinarily require this to achieve convergence. By purposefully allowing rounding error, but carefully bounding the amount introduced in each approximant, arbitrary-accuracy results can still be obtained while generating far fewer digits than necessitated by ARCHITECT. It was this observation—coupled with that regarding MSD stability—that inspired the work conducted for this paper.

3. Theoretical Analysis

3.1. Don't-change Digit Elision

The first optimisation we make to the ARCHITECT method is to avoid the recalculation of don't-change digits. This is the name we give to high-significance digits of later approximants that have stabilised.

The concept behind this optimisation is straightforward. To calculate the digits of approximant k, we begin by examining the digits of the previous two approximants. If these approximants are equal in their most-significant $q+\delta_\Sigma$ digits, it is guaranteed that approximant k will be equal to its two predecessors in its first q digits. Hence, we do not calculate them, skipping directly to digit q's generation.

The soundness of this optimisation can be justified by appealing to the digit dependencies of online arithmetic. Figure 2 provides some graphical intuition. Given that each approximant depends only on the value of its immediate predecessor, and recalling the definition of online delay from Section 2.2, we emphasise that the first q digits of one approximant depend only upon the first $q + \delta_\Sigma$ digits of the previous approximant [10]. Hence, if approximants $k-2$ and $k-1$ are equal in their first $q + \delta_\Sigma$ digits, approximant k is guaranteed to be equal to them in its first q digits.

3.2. Don't-care Digit Elision

Let us now turn to the issue of don't-care digit avoidance. We use this term to refer to low-significance digits in earlier approximants which do not prohibit the chosen iterative method's convergence. Herein, we present a don't-care analysis applicable to any stationary iterative method: Jacobi, Gauss-Seidel, successive overrelaxation, *etc.*

Consider a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A} \in \mathbb{R}^{N \times N}$. A stationary iterative method for its solution is defined as

$$\boldsymbol{M}\boldsymbol{x}^{(k+1)} = \boldsymbol{N}\boldsymbol{x}^{(k)} + \boldsymbol{b}, \tag{1}$$

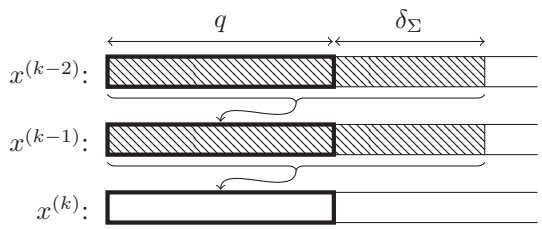


Figure 2: A proof sketch showing why it is sound to omit don't-change digits. If the two hatched regions contain the same $q + \delta_\Sigma$ digits, the three thick boxes are guaranteed to contain the same q digits, hence approximant k's calculation can begin at digit q.

with $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$ and $\boldsymbol{M}$ non-singular [14]. Letting $\boldsymbol{G} = \boldsymbol{M}^{-1}\boldsymbol{N}$, we obtain

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{M}^{-1}\boldsymbol{b}. \tag{2}$$

Approximant by approximant, we have

$$\begin{aligned}
\boldsymbol{x}^{(1)} &= \boldsymbol{G}\boldsymbol{x}^{(0)} + \boldsymbol{M}^{-1}\boldsymbol{b} \\
\boldsymbol{x}^{(2)} &= \boldsymbol{G}^2\boldsymbol{x}^{(0)} + \boldsymbol{G}\boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{M}^{-1}\boldsymbol{b} \\
&\vdots \\
\boldsymbol{x}^{(k+1)} &= \boldsymbol{G}^{k+1}\boldsymbol{x}^{(0)} + \sum_{i=0}^{k} \boldsymbol{G}^i\boldsymbol{M}^{-1}\boldsymbol{b}
\end{aligned} \tag{3}$$

starting from some initial guess $\boldsymbol{x}^{(0)}$.

Lemma 1. *If $\boldsymbol{A}$ is strictly diagonally dominant*[1]*, then $\sum_{i=0}^{\infty} \boldsymbol{G}^i = (\boldsymbol{I} - \boldsymbol{G})^{-1}$ [15].*

Thus,

$$\lim_{k\to\infty} \boldsymbol{G}^k\boldsymbol{x}^{(0)} = \boldsymbol{0}. \tag{4}$$

Applying Lemma 1 a second time, this observation allows us to conclude from (3) that

$$\lim_{k\to\infty} \sum_{i=0}^{k} \boldsymbol{G}^i\boldsymbol{M}^{-1}\boldsymbol{b} = \boldsymbol{x}.$$

Hence, (1) will converge to $\boldsymbol{x}$ for any choice of $\boldsymbol{x}^{(0)}$.

Lemma 2. *$\boldsymbol{x}$ is a fixed point of the iteration, i.e. $\boldsymbol{x} = \boldsymbol{G}^{k+1}\boldsymbol{x} + \sum_{i=0}^{k} \boldsymbol{G}^i\boldsymbol{M}^{-1}\boldsymbol{b}$ $\forall k$ [14].*

Introducing rounding error $\boldsymbol{\epsilon}_k$, as we propose to via truncation of each approximant, (2) becomes

$$\hat{\boldsymbol{x}}^{(k+1)} = \boldsymbol{G}\hat{\boldsymbol{x}}^{(k)} + \boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{\epsilon}_{k+1}$$

1. Our don't care analysis—in particular, (6)—necessitates strict diagonal dominance of $\boldsymbol{A}$ to ensure convergence. We do not consider this to be a hindrance, however, since strict diagonal dominance, which is computationally cheap to verify, is commonly used to ensure solubility when employing stationary iterative methods [16].

or, expressed per approximant,

$$\begin{aligned}\hat{\boldsymbol{x}}^{(1)} &= \boldsymbol{G}\hat{\boldsymbol{x}}^{(0)} + \boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{\epsilon}_1\\ \hat{\boldsymbol{x}}^{(2)} &= \boldsymbol{G}^2\hat{\boldsymbol{x}}^{(0)} + \boldsymbol{G}\boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{G}\boldsymbol{\epsilon}_1 + \boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{\epsilon}_2\\ &\vdots\\ \hat{\boldsymbol{x}}^{(k+1)} &= \boldsymbol{G}^{k+1}\hat{\boldsymbol{x}}^{(0)} + \sum_{i=0}^{k}\boldsymbol{G}^i\boldsymbol{M}^{-1}\boldsymbol{b} + \sum_{i=0}^{k}\boldsymbol{G}^i\boldsymbol{\epsilon}_{k+1-i} \quad (5)\end{aligned}$$

from some finite-precision initial guess $\hat{\boldsymbol{x}}^{(0)}$.

Defining computation error $\boldsymbol{e}^{(k)} = \boldsymbol{x} - \hat{\boldsymbol{x}}^{(k)}$, subtraction of (5) from the equality given in Lemma 2 results in

$$\boldsymbol{e}^{(k+1)} = \boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)} - \sum_{i=0}^{k}\boldsymbol{G}^i\boldsymbol{\epsilon}_{k+1-i},$$

wherein $\boldsymbol{e}^{(k)}$ captures errors due to the finiteness of both the iteration count *and* the precision. We wish to minimise this value. Since we cannot minimise $\boldsymbol{e}^{(k)}$ directly, we seek to minimise its upper bound instead. Taking norms,

$$\left\|\boldsymbol{e}^{(k+1)}\right\|_\infty \leq \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + \sum_{i=0}^{k}\|\boldsymbol{G}\|_\infty^i \left\|\boldsymbol{\epsilon}_{k+1-i}\right\|_\infty \quad (6)$$

since $\|\boldsymbol{G}\|_\infty < 1$. We ensure that $\|\boldsymbol{\epsilon}_{k+1-i}\|_\infty \leq r^{-d_{k+1-i}}$ by controlling the precision of each approximant's computation, expressed as a number of radix-r digits d_j.

For neatness, let $g(i)$ denote the maximum error introduced in approximant i:

$$g(i) = \|\boldsymbol{G}\|_\infty^i \, r^{-d_{k+1-i}}.$$

Defining $\boldsymbol{d}$ to be a column vector whose j^{th} element is d_{k+1-i} and assuming an available 'budget' of total digits D for computation, we aim to find

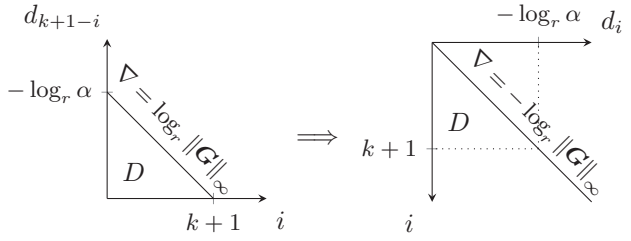
$$\begin{aligned}\min_{\boldsymbol{d}}\ & f(\boldsymbol{d}) = \sum_{i=0}^{k} g(i)\\ \text{subject to } & h(\boldsymbol{d}) = -D + \sum_{i=0}^{k} d_{i+1} = 0,\end{aligned} \quad (7)$$

thereby determining the optimal allocation of the available digits per approximant.

Theorem 1 (The optimal error distribution is uniform)**.** *The optimisation in* (7) *is achieved when* $g(i)$ *is a constant independent of* i*.*

Proof. Via the Karush-Kuhn-Tucker conditions [17], the optimal $\boldsymbol{d}$, $\boldsymbol{d}^*$, is obtained when

$$\begin{aligned}\nabla f(\boldsymbol{d}^*) + \lambda\nabla h(\boldsymbol{d}^*) &= \boldsymbol{0}\\ h(\boldsymbol{d}^*) &= 0\end{aligned}$$



(a) Sketch of (8). **(b)** Don't-care line.

Figure 3: Deriving the gradient of the don't-care line. Figure 3b was arrived at by transforming d_{k+1-i}, featured in Figure 3a, into d_i, after which it was rotated clockwise by 90° to match the presentation used in Figure 1. Since d_i does not feature k, Figure 3b's x-intercept (here, the origin) can be chosen arbitrarily.

for some multiplier λ. We have

$$\nabla f(\boldsymbol{d}) = \begin{pmatrix}\left(\|\boldsymbol{G}\|_\infty^k \ln r^{-1}\right) r^{-d_1}\\ \left(\|\boldsymbol{G}\|_\infty^{k-1} \ln r^{-1}\right) r^{-d_2}\\ \vdots\\ \left(\|\boldsymbol{G}\|_\infty^0 \ln r^{-1}\right) r^{-d_{k+1}}\end{pmatrix}$$

$$\nabla h(\boldsymbol{d}) = \begin{pmatrix}1\\1\\\vdots\\1\end{pmatrix},$$

i.e.

$$-\left(\|\boldsymbol{G}\|_\infty^i \ln r\right) r^{-d_{k+1-i}} + \lambda = 0.$$

Therefore, the optimisation in (7) is achieved when $g(i) = {}^{\lambda}/_{\ln r}$, a constant independent of i, as required. □

Setting $g(i) = \alpha$ and taking logs, we obtain

$$d_{k+1-i} = i\log_r \|\boldsymbol{G}\|_\infty - \log_r \alpha. \quad (8)$$

We present the transformation of this function to our required don't-care line graphically in Figure 3. From Figure 3b, we can infer that the optimal gradient of the don't-care line is $-\log_r \|\boldsymbol{G}\|_\infty$, and is therefore independent of α. The line's x-intercept is analogous to the precision with which one wishes to begin computation, so is user-defined.

Theorem 2 (Error can be arbitrarily minimised)**.** $\lim_{D,k\to\infty} \left\|\boldsymbol{e}^{(k)}\right\|_\infty = 0$.

Proof. As $D \to \infty$, $d_{k+1-i} \to \infty\ \forall i$. Consequently, $g(i) \to 0\ \forall i$. From (4), we can infer that

$$\lim_{k\to\infty} \boldsymbol{G}^k = \boldsymbol{0}.$$

Thus, from (6),

$$\lim_{D,k\to\infty} \left\|\boldsymbol{e}^{(k)}\right\|_\infty = 0,$$

as required. □

From Theorem 2, we can be sure that convergence to any accuracy is always achievable when $\boldsymbol{A}$ is strictly diagonally dominant, given enough iterations, for any choice of $\hat{\boldsymbol{x}}^{(0)}$.

4. Implementation

Armed with the analyses presented in Section 3, we are now in a position to design suitable control and storage infrastructure to support the efficient generation of digits within an MSD-first iterative solver.

4.1. Control Logic

For every element of $\hat{\boldsymbol{x}}^{(k)}$, $\hat{x}_v^{(k)}$, three factors determine the number of its digits that must be calculated:

- *The initial guess*. We assume that Figure 3's don't-care line is placed such that the minimum number of digits needed to represent $\hat{x}_v^{(0)}$ exactly are used. Thus, each approximant requires a minimum of
$$\chi_v = -\log_r \mathrm{ULP}\Big(\hat{x}_v^{(0)}\Big)$$
digits for its representation, where ULP returns the unit in the last place of its argument.
- *Don't-change digits*. Don't-change analysis is implemented through the comparison of successive approximants. Mathematically, this is represented as
$$\psi_v(k) = \max\Big(\mathrm{LZC}\Big(\Big|\hat{x}_v^{(k-1)} - \hat{x}_v^{(k-2)}\Big|\Big) - \delta_\Sigma,\ 0\Big),$$
in which LZC produces a leading-zero count. $\psi_v(k)$ returns the number of MSDs that are guaranteed to be stable within approximant k. Once digits in a particular position have been determined to be stable, in practice there is no longer a reason to revisit that position. Since the number of stable MSDs grows monotonically, we are able to optimise the successive-approximant comparison by ignoring the first $\psi_v(k-1)$ MSDs during iteration k. A counter is all that is required to implement this.
- *Don't-care digits*. The don't-care line's gradient is indicative of the number of additional digits to calculate per approximant. Hence, within approximant k, we *do* care about
$$\omega(k) = \lceil -k \log_r \|G\|_\infty \rceil$$
more digits than were contained in the initial guess. We always round towards $+\infty$ to ensure that we do not inadvertently neglect to calculate required LSDs.

Combining these, we arrive at

$$\beta_v(k) = \begin{cases} \chi_v + \omega_v(k) & \text{if } k < 2 \\ \chi_v - \psi_v(k) + \omega_v(k) & \text{otherwise,} \end{cases}$$

where $\beta_v(k)$ reveals the number of digits to produce per approximant. The index of the first digit to produce within each approximant is given by $\psi_v(k)$, which we only begin to evaluate once $k \geq 2$; prior to this, calculation always commences from the MSD.

We designed a parameterisable finite-state machine (FSM) to sequence digit propagation through a pipeline of online operators. Beginning with the consumption of approximant $k = 0$ (the user-supplied initial guess), it sweeps through $\beta_v(k)$ digits per approximant, incrementing k after each calculation. Once $k \geq 2$, don't-change digits start to be evaluated, shifting the start of each approximant's calculation by $\psi_v(k)$ digits away from the MSD.

From a user's perspective, no additional information needs to be supplied to take advantage of don't-change and don't-care digit elision. The system to be solved, defined by $\boldsymbol{A}$ and $\boldsymbol{b}$, along with $\hat{\boldsymbol{x}}^{(0)}$, are all that is required.

4.2. Memory

Since our don't-care analysis tells us the number of digits we need to compute per approximant *a priori*, we are able to compute approximants, in full, sequentially. This was demonstrated visually in Figure 1c. To enable don't-change digit avoidance, however, we require access to *one* previously computed approximant. This is one, rather than two, since the generation of the current approximant (k) can be used to infer the number of stable digits in the yet-to-be-computed approximant $k+1$. It is therefore safe for us to overwrite approximant $k-2$ with approximant k.

Since the number of memory words (chunks) required for each approximant grows over time, we opted to segment a single, flat memory for the storage of $\hat{\boldsymbol{x}}^{(k)}$ into two halves: one for even approximants, and one for odd. The correct memory bank for approximant k is therefore selected by simply evaluating $k \bmod 2$.

Output digit storage is not our only memory-related concern. Don't-change digit elision requires the ability to restart computation from arbitrary digit indices, thus we must store the internal *residues* of earlier iterations' operations.

Without delving into the minutiae of these mechanisms, it is important to note that residue storage requirements grow more slowly than those of $\hat{\boldsymbol{x}}^{(k)}$. With more thorough analysis, we believe that the majority of this additional storage can be eliminated: a task we leave to future work.

As a result of these sources of increasing storage burden, the size of the instantiated memories will determine the maximum precision and number of iterations (and consequently accuracy) to which one can compute. It is for this reason that we refer to the work presented in this paper, and ARCHITECT, as fixed *compute*-resource architectures. Our arbitrary-accuracy claim is subject to the availability of sufficient memory to solve the particular problem at hand.

5. Evaluation

Hardware performance evaluations were conducted to investigate how our method compares to both ARCHITECT and conventional LSD-first equivalents. For the latter, we compared against the performance of datapaths made up of parallel-in, serial-out arithmetic operators, as was done to evaluate ARCHITECT, for which precision must be set at compile time. These function in a similar digit-serial fashion to those used for ARCHITECT and this work.

Compared to iterative computation architectures constructed using conventional arithmetic (LSD-first) operators, we expect ours to compare favourably due to:

- *There being no need to determine, or fix, precision in advance*. MSD-first computation circumvents the need to begin computation from the LSD, avoiding both under- and over-budgeting compute resources for a given problem. The same hardware can thus be used for problems that would ordinarily require distinct architectures in order to solve efficiently.
- *Don't-change digit elision*. Online arithmetic grants us the ability to declare MSDs to be stable. This is generally not possible in LSD-first architectures, in which carries can propagate from LSD to MSD.
- *Don't-care digit elision*. Our theoretical analysis facilitates the growth of precision over time. This allows us to focus exclusively on digits known to be of value, thereby increasing efficiency. This is difficult to achieve in LSD-first architectures, in which every approximant must ordinarily be calculated to a maximum (worst-case) precision.

Versus ARCHITECT, we hypothesise that iterative solvers constructed following the proposals presented in this paper will achieve greater performance thanks to:

- *Don't-change and don't-care digit elision*. Beyond the benefits outlined above, additional performance gains can be obtained through digit generation avoidance in MSD-first architectures. Of the two classes, we expect don't-care digit avoidance to have the greatest effect since digit generation requires more time the lower the significance of the digit. This is a property of the hardware used for operators which are themselves inherently iterative, *e.g.* multipliers, and applies equally to ARCHITECT and this work.
- *A more efficient digit computation pattern*. As exemplified in Figure 1b, ARCHITECT refines earlier approximants as needed in order to reach further into the iteration-precision space. Switching between iterations incurs stalling penalties which our proposal limits. Since our don't-care line is monotonic, we have no need to revisit earlier iterations, as was demonstrated in Figure 1c.
- *Significantly lower memory usage*. Closely related to the previous point, our computation pattern permits us to discard older approximants and their residues. This allows us to do away with ARCHITECT's CPF, with memory use scaling only with precision.

All of our hardware implementations targetted a Xilinx Virtex UltraScale FPGA (XCVU190-FLGB2104-3-E), with Vivado 16.2 used for compilation. Results obtained in hardware were compared for correctness against a golden software model executed in MATLAB.

5.1. Jacobi Method Benchmark

We used the Jacobi method, a well known stationary iterative algorithm, as a case study for the proposals presented in this paper. Jacobi is of the form expressed in (1), with $\boldsymbol{M} = \mathrm{diag}(\boldsymbol{A})$. Without loss of generality, we used matrix size $N = 2$, radix $r = 2$, chunk width $U = 8$ and online delay $\delta_\Sigma = 5$ in this paper, constructing the datapath shown in Figure 4. This is identical in structure to that used in our previous work [1]. Digit sequencing and storage were realised using an FSM and memory architecture constructed as outlined in Section 4.

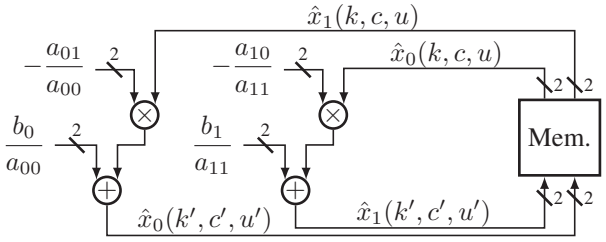


Figure 4: 2D Jacobi method benchmark datapath. Adders and multipliers are radix-2 signed-digit online operators.

Mirroring the experiments conducted to evaluate the performance of ARCHITECT, we solved systems of the form

$$\boldsymbol{A}_m = \begin{pmatrix} 1 & 1-2^{-m} \\ 1-2^{-m} & 1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}, \quad \boldsymbol{x}^{(0)} = \boldsymbol{0},$$

with elements of $\boldsymbol{b}$ selected from a uniform distribution bounded to $[0, 1)$. This facilitated direct comparison.

The criterion $\|\boldsymbol{A}_m\boldsymbol{x} - \boldsymbol{b}\| < \eta$, with $0 < \eta \leq 1$, was used at runtime to establish convergence. The conditioning of $\boldsymbol{A}_m$ was controlled using $m \in \mathbb{R}_{>0}$. This range of m guaranteed solubility since $\boldsymbol{A}_m$ was always strictly diagonally dominant. Higher m results in a higher condition number $\kappa(\boldsymbol{A}_m)$, indicating that higher precision will be required to achieve convergence.

5.2. Performance Comparison

To begin, we set accuracy bound $\eta = 2^{-6}$ and investigated how the conditioning of $\boldsymbol{A}_m$ affected the performance of our proposal relative to conventional LSD-first arithmetic. Figure 5 illustrates the speedup in solve time, both for our proposal and for ARCHITECT. Note that speedups below unity are slowdowns.

Figure 5a compares our proposal and ARCHITECT against LSD-first arithmetic with a fixed precision of 30 bits (LSD-30). This precision is over-budgeted for the solution of well conditioned matrices, hence we see that, when m is small, both ARCHITECT and our proposal can compute more quickly. ARCHITECT requires $m \lessapprox 0.013$ in order to beat LSD-30, whereas our proposal only requires $m \lessapprox 0.11$. Figure 5b demonstrates that if LSD-first arithmetic is given an under-budgeted precision of just eight bits (LSD-8), then only arbitrary-precision iterative solvers can solve systems with $m > 2$. Even if LSD-8 could run indefinitely, it would never be able to converge to an accurate-enough result.

We remark that the values that appear in Figure 5 are not particularly meaningful since we are comparing our proposal against unoptimised baselines, rather than against the state of the art in LSD-first arithmetic. Nonetheless, our emphasis here is that, however optimised a conventional implementation is, if its precision is sufficiently over- or under-budgeted, we will eventually outperform it.

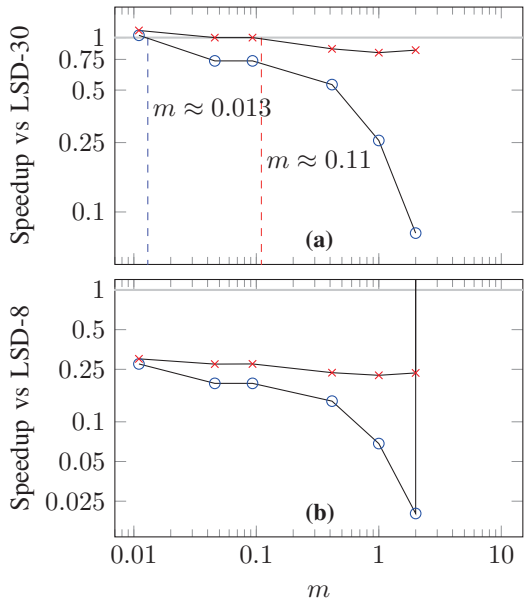


Figure 5: A comparison of our proposal against conventional LSD-first arithmetic. (a) shows how the conditioning of input matrix $\boldsymbol{A}_m$ affects the solve time of both ARCHITECT (—○—) and our proposal (—×—) compared to LSD-30. ARCHITECT computes faster than LSD-30 only when $m \lessapprox 0.013$, but our proposal beats LSD-30 when $m \lessapprox 0.11$. (b) shows that even though both ARCHITECT and our proposal lead to a slowdown compared to LSD-8, there is nevertheless a point—at $m > 2$—whence LSD-8 does not converge at all, hence our speedup (and ARCHITECT's) is infinite.

To further evaluate the performance of our proposal, Figure 6 presents a side-by-side comparison of solving different linear systems using our method and ARCHITECT. The results show that, generally, as m increases, more time is required in order to achieve a sufficiently accurate result. Most strikingly, our method demonstrates high efficiency with corresponding decreases in computation time being found versus ARCHITECT. When computing a well conditioned matrix with $m = 0.046$, our design is 1.77× faster than ARCHITECT, while for $m = 6$ it is some 2500× faster. Our proposal outperformed ARCHITECT in all tested cases, with a geometric mean 20.1× speedup obtained.

5.3. Scalability Comparison

To evaluate the scalability of our proposed method, Figure 7 shows how the requested accuracy, controlled by η, affects the solve time and the number of computed digits. We can see from Figure 7a that ARCHITECT requires increasingly more time to reach a solution than our proposal as the requested accuracy increases. With low accuracy requirements, such as $\eta = 2^{-4}$, our method is 1.82× faster than ARCHITECT. In the case of high accuracy, such as when computing to $\eta = 2^{-256}$, our method is 193× faster. From a more fundamental perspective, Figure 7b shows the relationship between the requested accuracy and the total number of digits calculated. Our method calculates 1.04×

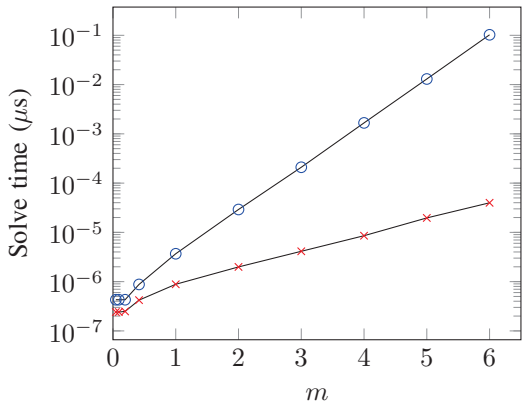


Figure 6: How the conditioning of $\boldsymbol{A}_m$ affects the solve time of ARCHITECT (—○—) and our proposal (—×—).

fewer digits than ARCHITECT with $\eta = 2^{-4}$, increasing to 44.1× fewer digits when $\eta = 2^{-256}$.

Figures 7a and b also demonstrate how don't-care and don't-change digit avoidance individually affect computation time and the number of computed digits. As expected, don't-care digit elision leads to the majority of our design's efficiency savings over ARCHITECT. The gap between the don't-change plus don't-care and don't-care-only lines widens as η reduces, however, indicating that consideration of don't-change digits becomes more important with higher accuracy requirements. This makes sense since, as η falls, more iterations are required to achieve convergence, thus affording more opportunity for don't-change digit elision.

Finally, Figure 7c shows the minimum number of on-chip memory blocks that need to be instantiated in order for our proposed solver and ARCHITECT to reach particular accuracies. For lower-accuracy cases ($\eta \geq 2^{-32}$), both designs require approximately the same amount of memory, although our proposal is slightly inferior due to don't-change detection overheads. The advantages over ARCHITECT's memory addressing explained in Section 4.2 come to the fore with higher accuracy requirements. For the lowest η tested, 2^{-256}, we observed a 22.4× memory reduction.

6. Conclusion & Future Work

In this paper, we proposed a new methodology for the creation of iterative numeric solvers in hardware in which the achievable result accuracy is bounded only by the size of the available memory. Our work relies on online arithmetic operators in order to generate results from most-significant digit first. Efficiency over ARCHITECT, the state-of-the-art arbitrary-accuracy hardware solver, was achieved using digit dependency and forward error analyses to identify both stable and unimportant—don't-change and don't-care—digits, excluding them from calculation. Our don't-change analysis holds for any iterative method, and was realised in hardware using simple runtime detection logic. For the identification of don't-care digits, we presented a theorem for stationary iterative methods allowing many low-significance digits to be ignored without impacting upon the solver's ability to reach a result of any accuracy.

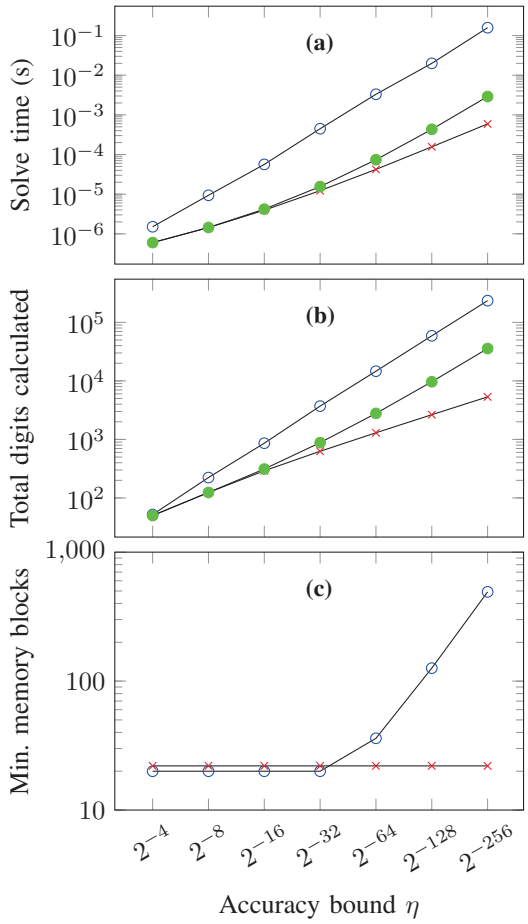


Figure 7: How the requested accuracy bound η affects (a) the solve time, (b) the total number of digits calculated and (c) the minimum memory requirement for ARCHITECT (—○—), our proposal (—×—) and ours with don't-care digit omission only (—●—), with $m = 1$.

We evaluated our proposals using the Jacobi method. Versus ARCHITECT, we showed a mean 12.8× reduction in the number of digits generated in order to solve a set of differently conditioned matrices to the same accuracy. In those cases, elision of redundant digit calculation led to a geometric mean 20.1× speedup. The monotonicity of our don't-care line allows us to eliminate the need to revisit and refine earlier approximants, leading to an up-to 22.4× memory footprint reduction. Making more efficient use of a given-sized memory enables us to advance much deeper into the iteration-precision space than ARCHITECT allowed. Finally, versus a fixed-precision Jacobi solver constructed from conventional LSD-first arithmetic operators, we demonstrated that our proposal is able to solve more difficult linear systems, as ARCHITECT did, and that we require less solve time when a system is well conditioned.

In the future, we will attempt to extend our don't-care digit analysis to other iterative algorithms, including Krylov subspace methods such as conjugate gradient descent. We will explore the possibility of more aggressive don't-change digit analysis, likely to be algorithm-specific, that may allow us to do away with the runtime detection proposed in this work, thereby achieving further performance improvements. We are particularly keen to see if it is possible to obtain the same rates of growth in MSD stability and LSD significance, thus achieving parallel don't-change and don't-care lines. Doing so would enable the creation of efficient, fixed compute-resource hardware with no bounds on accuracy, and may allow us to generalise the e-method [18].

Acknowledgements

This work was supported by the EPSRC (grant numbers EP/P010040/1 and EP/K034448/1), Imagination Technologies and the Royal Academy of Engineering. The authors wish to thank Matei Istoan and Ian McInerney for their helpful suggestions. Supporting data for this paper are available online at https://doi.org/10.5281/zenodo.1219704.

References

[1] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides, "ARCHITECT: Arbitrary-precision Constant-hardware Iterative Compute," in *International Conference on Field Programmable Technology*, 2017.

[2] G. Constantinides, A. Kinsman, and N. Nicolici, "Numerical Data Representations for FPGA-based Scientific Computing," *IEEE Design & Test of Computers*, vol. 28, no. 4, 2011.

[3] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision Computation: Mathematical Physics and Dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, 2012.

[4] D. H. Bailey and J. M. Borwein, "High-precision Arithmetic in Mathematical Physics," *Mathematics*, vol. 3, no. 2, 2015.

[5] M. Benzi, T. M. Evans, S. P. Hamilton, M. L. Pasini, and S. R. Slattery, "Analysis of Monte Carlo-accelerated Iterative Methods for Sparse Linear Systems," *Numerical Linear Algebra with Applications*, vol. 24, no. 3, 2017.

[6] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, 2011.

[7] X. Fang and M. Leeser, "Open-source Variable-precision Floating-point Library for Major Commercial FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 9, no. 3, 2016.

[8] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-performance Mixed-precision Linear Solver for FPGAs," *IEEE Transactions on Computers*, vol. 57, no. 12, 2008.

[9] M. Jaiswal and H. So, "Area-efficient Architecture for Dual-mode Double Precision Floating Point Division," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 2, 2017.

[10] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Elsevier, 2004.

[11] K. Shi, D. Boland, and G. A. Constantinides, "Efficient FPGA Implementation of Digit Parallel Online Arithmetic Operators," in *International Conference on Field Programmable Technology*, 2014.

[12] Y. Zhao, J. Wickerson, and G. A. Constantinides, "An Efficient Implementation of Online Arithmetic," in *International Conference on Field Programmable Technology*, 2016.

[13] M. Lisi, "Some Remarks on the Cantor Pairing Function," *Le Matematiche*, vol. 62, no. 1, 2007.

[14] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. Siam, 2002.

[15] E. Cheney and D. Kincaid, *Numerical Mathematics and Computing*. Nelson Education, 2012.

[16] J. D. Hoffman and S. Frankel, *Numerical Methods for Engineers and Scientists*. CRC Press, 2001.

[17] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[18] M. D. Ercegovac, "A General Hardware-oriented Method for Evaluation of Functions and Computations in a Digital Computer," *IEEE Transactions on Computers*, vol. C-26, no. 7, 1977.