BitSAD v2: Compiler Optimization and Analysis for Bitstream Computing

KYLE DARUWALLA, HENG ZHUO, ROHIT SHUKLA, and MIKKO LIPASTI,
University of Wisconsin—Madison, USA

Computer vision and machine learning algorithms operating under a *strict* power budget require an alternate computing paradigm. While bitstream computing (BC) satisfies these constraints, creating BC systems is difficult. To address the design challenges, we propose compiler extensions to BitSAD, a DSL for BC. Our work enables bit-level software emulation and automated generation of hierarchical hardware, discusses potential optimizations, and proposes compiler phases to implement those optimizations in a hardware-aware manner. Finally, we introduce population coding, a parallelization scheme for stochastic computing that decreases latency without sacrificing accuracy, and provide theoretical and experimental guarantees on its effectiveness.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Theory of computation →** *Probabilistic computation*; Streaming models;

Additional Key Words and Phrases: Bitstream computing, stochastic computing, pulse density modulation, compiler

ACM Reference format:
Kyle Daruwalla, Heng Zhuo, Rohit Shukla, and Mikko Lipasti. 2019. BitSAD v2: Compiler Optimization and Analysis for Bitstream Computing. *ACM Trans. Archit. Code Optim.* 16, 4, Article 43 (November 2019), 25 pages.
https://doi.org/10.1145/3364999

1 INTRODUCTION

Currently, computer vision (CV) and machine learning (ML) are greatly expanding the efficacy of automated systems and robots. At the same time, fabrication and manufacturing advancements are enabling pico-aerial vehicles (PAVs) [8, 10]: flying robots that are smaller than a human fingernail. Unfortunately, due to *strict* power and resource budgets, microcontrollers are not a viable platform, and current ASICs are only able to perform the necessary functions to keep the PAV in static, stable flight [26]. Enabling advanced CV/ML algorithms on these robots will require a fundamentally different computing paradigm.

Bitstream computing is a low resource, efficient information processing framework. While embedded systems that require high-performance signal processing and control have historically relied on fixed point algorithms, the sensing and actuation interfaces in such systems increasingly rely on *bitstreams*—oversampled, single-bit, sigma-delta-modulated (SDM) representations of data inputs and control outputs. Typical computing substrates require power-hungry data converters

Authors' address: K. Daruwalla, H. Zhuo, R. Shukla, and M. Lipasti, University of Wisconsin—Madison, 1415 Engineering Dr., Madison, WI 53705; emails: {daruwalla, hzhuo2, rshukla3}@wisc.edu, mikko@engr.wisc.edu.

1544-3566/2019/11-ART43
https://doi.org/10.1145/3364999

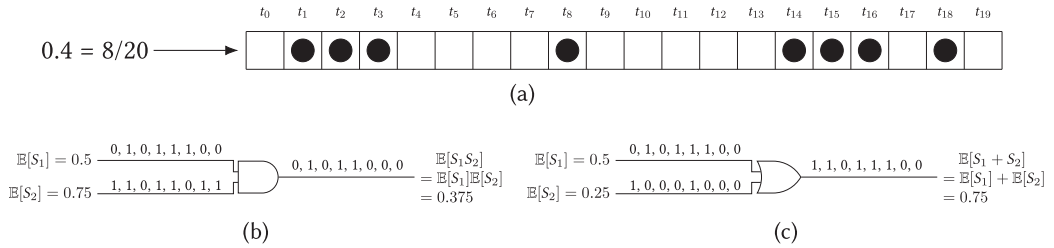


Fig. 1. This figure shows how data are represented and computations are performed using stochastic computing. (a) The probability of the occurrence of a high bit at a particular time tick is directly proportional to the real number's value. To represent the value 0.4 over 20 time ticks, there should be eight random occurrences of a high bit. (b) Given two independent bitstreams, multiplication can be performed using just an AND gate. Because the value is represented over a window of time ticks, the longer the time window, the more accurate results would be. (c) An example of scaled addition using an OR gate. Notice that the output bitstream does not exactly match the expected value. This occurs due to the randomness of each bit. To produce accurate results, inputs should be highly uncorrelated, and the total runtime should be sufficiently large. We address the runtime issue in various ways in Sections 3.1, 3.2, and 6.

to translate between these bitstreams and the fixed point compute format. Instead, bitstream computing directly processes the output from the sensors. To compute on bitstreams, we can apply standard techniques from stochastic computing [11, 12], or we can use fixed point arithmetic but leverage the oversampled data format to create binary friendly constants [15]. We divide these two regimes into *stochastic bitstream* and *deterministic bitstream* computing.

In this article, we extend BitSAD, a domain specific language (DSL) for bitstream computing, by highlighting the interesting design challenges associated with bitstream computing that motivate compiler extensions. First, we introduce **bit-level, cycle-accurate software emulation of bitstream hardware** into BitSAD programs. In particular, we discuss the challenges associated with determining the minimum latency of BC designs to show why bit-level emulation is important. We also add **automatic synthesizable hardware generation** of BitSAD code. Additionally, we highlight code **optimizations that make sense for BC** that are uncommon for traditional computing, and we include **a phase in the compiler that makes area-aware optimizations**. Finally, we introduce **population coding—a new mechanism for producing accurate stochastic computing results while decreasing latency**.

2 BACKGROUND

To discuss the importance of the optimizations and analysis that BitSAD provides, we will provide some background on stochastic computing and pulse density modulation. Each topic applies to *stochastic* and *deterministic* bitstreams, which will be defined in this section. While we briefly introduce some applications to give context for each type of bitstream, we will not delve into the application details yet. Instead, those details will be discussed in conjunction with the compiler and optimization improvements in later sections.

2.1 Stochastic Bitstreams

Stochastic computing is a paradigm for processing information as streams of random bits [11, 12]. At each time step, a given floating point number, p, is represented as a sample, X_t, from a Bernoulli distribution as shown in Equation (1). We will refer to bitstreams defined by Equation (1) as *stochastic bitstreams*. Figure 1(a) illustrates an example of such a bitstream,

$$\mathbb{P}(X_t = 1) = p \qquad \mathbb{P}(X_t = 0) = 1 - p. \tag{1}$$

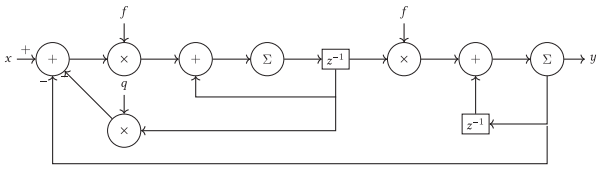


Fig. 2. A block diagram of a state-variable filter (SVF). f and q are fixed point constants, and x and y are deterministic bitstreams. Σ indicates a sigma-delta modulator.

To obtain a floating point estimate of a stochastic bitstream, we can simply average the bits over a window of time, T:

$$p = \mathbb{E}[X_t] \approx \overline{p} = \frac{1}{T}\sum_{t=1}^{T} X_t. \tag{2}$$

By operating on stochastic bitstreams, we can perform complex operations such as multiplication using simple logic gates (as shown in Figure 1(b) and (c)). Moreover, References [2, 12, 20] have implemented even more complex operations such as vector norms, matrix-matrix multiply, and the Moore-Penrose pseudoinverse on stochastic bitstreams.

The challenge, then, is to effectively integrate the work done on individual stochastic bitstream operators into end-to-end applications. For example, computer vision algorithms rely on matrix-matrix products and pseudoinverses, so synthesizing a CV algorithm into a complete stochastic computing design is possible; however, it is not attractive to most engineers, since mapping an algorithm to simulation code and then to hardware and, finally, testing the system and iterating the design is error-prone and labor-intensive. To alleviate this, BitSAD provides the `SBitstream` data type and a complete set of matrix operations to work with stochastic bitstreams.

2.2 Deterministic Bitstreams

Deterministic bitstreams are similar to stochastic bitstreams, except each bit, X_t, is deterministically generated instead of drawn from a probability distribution. Pulse density modulation (PDM), a common audio encoding, is a deterministic bitstream [15]. In this format, a higher density of "1"s in the bitstream indicates a larger amplitude. Normal digital audio data uses pulse coded modulation (PCM) at a sample rate of 44.1 kHz, while PDM data is oversampled at 3 MHz. We can leverage this disparity to create small, efficient audio signal processing pipelines.

Consider the state-variable filter (SVF) shown in Figure 2. Equation (3) defines the coefficients used by the filter,

$$f = 2\sin\left(\frac{\pi F_c}{F_s}\right) \qquad q = \frac{1}{Q}. \tag{3}$$

The center frequency, F_c, is specified by the application and cannot be changed. But the sample frequency, F_s, can be controlled in a PDM system (fixed at 44.1 kHz in PCM), since 3 MHz is a much higher sample rate than the application requires. By controlling F_s, PDM systems can choose low-precision coefficients that are powers of 2, leading to smaller and more efficient filters [7].

To enable building deterministic bitstream systems, BitSAD provides the `DBitstream` data type and a set of operations on it.

2.3 BitSAD v1

BitSAD v1 is a domain-specific language for bitstream computing developed by Daruwalla et al. during the first Workshop on Unary Computing (WUC'19) at ISCA [6]. Built using Scala, it enables Matlab/Python-like syntax to describe bitstream computing algorithms. Furthermore, it leverages

Scala's compiler plugin support to automatically generate hardware from user code. Listing 1 is a sample BitSAD program from the original paper. It implements a stochastic bitstream algorithm to compute the singular-value decomposition of a matrix. We will not go into the details of the algorithm in this section. Instead, we are presenting a sample program for comparison with a BitSAD v2 program in Section 4.

```
trait Parameters {
  val m: Int
  val n: Int
}

object DefaultParams extends Parameters {
  val m = 2
  val n = 2
}

case class Module (params: Parameters) {
  // Define outputs
  val outputList = List(("v", params.n, 1),
                        ("u", params.m, 1),
                        ("sigma", 1, 1))

  def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
          (Matrix[SBitstream], Matrix[SBitstream], SBitstream)
      = {
    // Update right singular vector
    var w = A * v
    // :/ is fixed-gain division
    // functionally the same as division
    // more efficient implementation for dividing by a constant
    // refer to BitSAD v1 paper for more details
    var wScaled = w :/ math.sqrt(params.m)
    var u = wScaled / Matrix.norm(wScaled)

    // Update left singular vector
    var z = A.T * u
    var zScaled = z :/ math.sqrt(params.n)
    var sigma = Matrix.norm(zScaled)
    var _v = zScaled / sigma

    (u, _v, sigma)
  }
}
```

Listing 1. BitSAD v1 program for iterative Singular Value Decomposition (SVD).

BitSAD algorithms are defined by a `loop()` function that is a programmatic description of the dataflow graph of an algorithm. Since Scala is a programming language built on top of Java, developers can write code that uses Java-like syntax and features (i.e., Scala supports notions of classes, objects, and abstract classes called traits). At the same time, Scala is functional, so it has convenient frameworks like map-reduce, anonymous functions, and currying. The purpose of this work is not to discuss the features of Scala; instead, we suggest that users familiar with Java and/or Python will find themselves capable of writing Scala and, subsequently, BitSAD code easily. Within the body of the `loop()` function, users may want to express complex algorithms with matrix operators. Though this is not available in Scala, BitSAD v1 includes a `Matrix` class to make operating

with matrices as simple as writing Matlab code (which is a matrix-first language). We refer readers to Reference [6], the original paper, for more details on basic syntax. Understanding the full list of available syntax will not be required for this article.

We will also save more details about the structure of the program for Section 4 and instead point out some key improvements presented by BitSAD v2:

(1) **Bit-level software emulation:** Consider Lines 21 and 30 in the code above. Each line is a separate invocation of the * operator that maps to two distinct multipliers in hardware. Different hardware multipliers produce outputs dependent on the *specific* inputs passed to them. Currently, BitSAD v1 does not capture this distinction, and so it is unable to provide accurate simulations of potential errors in a program. We discuss the possible errors and how we address them with bit-level, cycle-accurate emulation in Section 3.
(2) **Functional composition:** Currently, a designer must specify their entire algorithm in a single `loop()` function body. It is not possible to break a system into sub-systems that are each designed and tested in separate Scala files and then integrate all the sub-systems together in a hierarchical fashion. BitSAD v2 improves upon this by allowing functional composition as described in Section 4.4.
(3) **Code optimizations:** BitSAD v1 generates hardware for a user's program exactly as specified. Often, the most natural way to express an algorithm is not the most efficient in terms of hardware implementation. To address this, we discuss some code optimizations on deterministic bitstreams in Section 5.
(4) **Stochastic computing latency:** The original work did not address the high latency associated with stochastic bitstream designs. We introduce a new encoding scheme called population coding in Section 6, and we provide language-level support for it in BitSAD v2.

BitSAD v1 presents a major step forward in enabling designers unfamiliar with bitstream computing to leverage its resource efficient paradigms. BitSAD v2 improves on this work by introducing frameworks critical to design complex bitstream computing systems. For the rest of this article, BitSAD v2 will be referred to as just BitSAD, while we will still explicitly mention BitSAD v1 when referenced.

3 SOFTWARE EMULATION

By default in BitSAD v1, all `SBitstream` data types are simulated using floating point numbers. In other words, `SBitstream` operates only as a wrapper around the underlying floating point value that the bitstream represents. Under this framework, it is possible to detect simple behavior, such as saturation, that occurs when operating with bitstreams, but bit-level, cycle-accurate simulation is not possible. Thus, BitSAD v1 is useful for verifying the correctness of algorithms at a high level, but it fails to inform the user of issues that may arise at the hardware level. For example, if two bitstreams are correlated, then most stochastic circuits will not produce the correct output. Yet, if we only represent the bitstreams by their floating-point equivalents and not the individual bits at each time step, then we will not realize that they are correlated. This will only become apparent when the designer runs the physical hardware, which operates on physical bitstreams, and the output is incorrect. Tracking down the source of this error will require dataflow changes at the program-level, but debugging this mistake using a Verilog simulator is painful and inefficient. In this section, we provide further motivation for bit-level emulation, and we introduce a mechanism into BitSAD to support such behavior.

3.1 The Latency Issue

A common critique of stochastic computing is that getting an accurate result requires many cycles of operation. This is intuitive from Equation (2). By the law of large numbers, we expect that the empirical average approximates the mean well as $T \to \infty$, but as we show in Section 3.1.1, it is not easy nor reasonable to analytically determine how large T should be for complex designs. Yet, to estimate T in software, the language must support bit-level, cycle-accurate simulations. Without bit-level simulation, the estimate of T does not capture any of the dynamic stochastic behavior of bitstreams (e.g., random fluctuations [12]). Using current design tools, a developer needs to manually write code to emulate the bit-level behavior of their design, or they must create hardware and simulate it. Needing to wait until hardware testing to determine whether a design will meet a real-time deadline is impractical, and it makes bitstream computing unattractive to system designers.

3.1.1 Effects of Numerical Scale on Convergence. The issue of determining the overall latency of a stochastic computing design is input dependent. A simple analysis shows that the number of samples, T, required for a stochastic bitstream to converge to its floating point value within a relative tolerance is dependent on the magnitude of the number being represented. To see this effect, we will define the error of a stochastic bitstream as

$$\mathbb{P}\left(\left|\frac{1}{T}\sum_{t=1}^{T} X_t - p\right| > \epsilon\right) \tag{4}$$

for some $\epsilon > 0$. Here, ϵ represents the absolute error between our empirical estimate and the true floating point value, p, that the bitstream should represent. Since X_t is a random variable, we can never be sure that the error is always less than ϵ, but we can bound the probability that an error occurs using Hoeffding's inequality [13]:

$$\mathbb{P}\left(\left|\frac{1}{T}\sum_{t=1}^{T} X_t - p\right| > \epsilon\right) \leq 2e^{-2T\epsilon^2}. \tag{5}$$

We will assume for now that $T = 100$ is fixed.

Now, suppose $p \approx 1$. If we want a relative error of 10% in our final result, then this implies that $\epsilon = 0.1$. So, we see that

$$\mathbb{P}\left(\left|\frac{1}{T}\sum_{t=1}^{T} X_t - p\right| > \epsilon\right) \leq 2e^{-2(100)(0.1)^2} \approx 2(0.1353).$$

However, if $p \approx 0.01$, then a relative error of 10% implies that $\epsilon = 0.001$. Our bound changes to

$$\mathbb{P}\left(\left|\frac{1}{n}\sum_{t=1}^{n} X_t - p\right| > \epsilon\right) \leq 2e^{-2(100)(0.001)^2} \approx 2(0.9998).$$

Note that Hoeffding's inequality provides a two-sided tail bound (due to the absolute value in Equation (5)), so we use the notation $2(0.1353)$ to reference the probability of error on each tail. In other words, $\mathbb{P}\left(\frac{1}{T}\sum_{t=1}^{T} X_t - p > \epsilon\right) \leq 0.1353$ and the $\mathbb{P}\left(\frac{1}{T}\sum_{t=1}^{T} X_t - p < -\epsilon\right) \leq 0.1353$. But the probability of either event happening is the sum of the two cases (i.e., $2(0.1353)$).

So, for the same relative error requirement, depending on the value being represented, ϵ changes, and, consequently, the probability of error changes. We can see how the bound changes for varying $p \in [0.01, 1]$ in Figure 3(a) below. Even at around $p \approx 0.6$, there is already a 50% chance that our bitstream is wrong. Thus, for 10% relative error, any estimate for $p < 0.6$ is likely to be wrong.

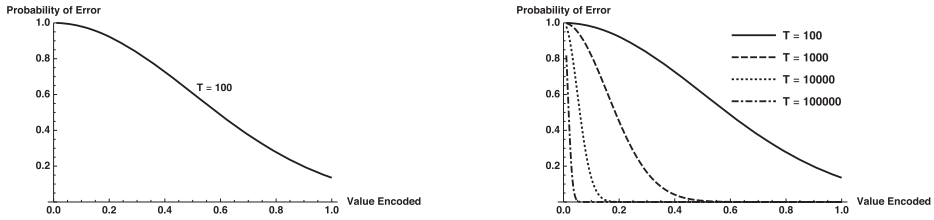


Fig. 3. (a) Error Bound for varying $p \in [0.01, 1]$ and (b) varying T as well. As seen in the plots, smaller numbers result in a higher probability of error. This can be controlled by increasing the number of samples, T, acquired.

An obvious solution to this issue is to increase T, but as seen in Figure 3(b), we need $T \geq 100000$ before most of the values of p are estimated with a low probability of error. If the correct choice of T can vary between 1,000 and 100,000 depending on the floating point value being encoded, then it is difficult for a designer to correctly reason about the number of samples required for their algorithm.

3.2 Simulatable Code

To address the issue of estimating T, we have added bit-level emulation to BitSAD. First, for every operator, +, −, *, /, and so on, we have developed classes that emulate the bit-level behavior of the hardware unit for the respective operator. It is important to perform emulation at the hardware module level, because the chosen implementation of an operator can effect how long the module takes to converge (e.g., Reference [25] implements a faster division and square root circuit than in References [11, 12]).[1]

Yet, we do not want users to have to instantiate objects to express functional semantics. Instead, programmers should expect to write code the same as before in Listing 1. By default, this code is processed using float point numbers; however, the definition of an operator, such as * in the example, receives more arguments than just the two operands. These additional arguments are in the form of a `SimulationId`, which allows us to uniquely identify each invocation of an operator on a specific pair of inputs. For example, the function definition of * actually looks like Listing 2.

```
1// Scala allows multiple parameter lists to support currying
2def *(that: SBitstream)(implicit id: SimulationId): SBitstream =
3  (id.lhs, id.rhs) match {
4    // default case where bit-level simulation is not required
5    case ("", "") => SBitstream.times(this, that)
6    case (thisName, thatName) => {
7      // create result SBitstream using floating-point times() function
8      var z = SBitstream.times(this, that)
9      // select the correct instance of the hardware module emulator
10     var op = SBitstream.findOperator(thisName, thatName, "times")
11     // evaluate the bit-level result and push it onto the result
12     z.push(op.evaluate(List(x.pop, y.pop)))
13     return z
14   }
15}}
```

Listing 2. Simulatable implementation of *.

[1]See Section 7 about support for different implementations.

Thus, * is just a wrapper that exercises the correct hardware emulator instance for that specific pair of operands. The key to making this transparent to users is the `implicit` keyword available in Scala. Implicit parameters are specified in their own parameter list to support currying. Currying is a functional programming semantic that allows a programmer to leave off parameters. For example, `x.*(y)` is a function that takes one parameter, namely the `SimulationId` parameter that we did not specify. So, to actually evaluate the result of *, we would have to specify this final parameter as shown in Listing 3. Of course, this is inconvenient for users. To alleviate this burden, we use the `implicit` keyword. If a parameter list is declared as implicit, then the Scala compiler will attempt to infer what value to insert for the missing parameter. In the class definition of `SBitstream`, we specify that this implicit value should default to `SimulationId("", "")`. Thus, by default, with no changes to a program written in BitSAD v1, we support the same floating-point based simulation as before.

```
1// explicitly stating the second parameter list
2var b = (A * x)(SimulationId("A", "x"))
3var c = (A * c)(SimulationId("A", "c"))
```

Listing 3. Bit-level execution.

To support bit-level simulation with minimal user overhead, we introduce a Scala macro, `simulatable`, to transform an expression or block of expressions into bit-level code. Scala macros allow language designers to manipulate the syntax tree before it is parsed by the compiler. Thus, we can transform code that looks like Listing 4 to Listing 3 with a simple macro call as highlighted below.

```
1simulatable({ // code in simulatable() auto-inserts the SimulationIds
2  var b = A * x
3  var c = A * b
4})
```

Listing 4. Bit-level execution with simulatable macro.

So, while it is not always possible to analytically determine the number of cycles until convergence of a bitstream, using BitSAD with bit-level simulation, a designer can empirically determine the value of T. Moreover, since BitSAD supports control flow logic for test code, one could easily write a loop that exits when some user-defined error is below a threshold, then print the number of iterations executed. Thus, the designer can determine experimentally the number of cycles for sufficient application tolerance. For example, in Listing 5, we instantiate a module for computing the SVD, which is a complex algorithm. Analytically evaluating T would be infeasible, but we can write some test code to experimentally determine T (e.g., Lines 28–42 in Listing 5). Specifically, note Line 41 highlighted in red. Here, we are running the algorithm for as many cycles as required to reach the specified application tolerance. At the end of the program, we print T to see how many cycles it took to reach the required application tolerance.

Furthermore, note that there are no restrictions to what can be written outside of the `Module` definition. So, our test code can be as general and flexible as Scala allows, and since Scala is a general-purpose functional programming language, the degree of flexibility is extremely high. For instance, we could wrap all the code in Listing 5 in a loop and test our module over many trials of randomly generated inputs and then print out the average cycle count over all trials. Thus, with

minimal effort, BitSAD enables designers to get accurate estimates of the latency of their bitstream designs.

```
package IterativeSVD
// ... skipping lines that define imports/module params
//     they are the same as Lst. 1
case class Module (params: Parameters) { /* define module, see Lst. 1 */}
object IterativeSVD {
  final val epsilon = 0.1
  def main(args: Array[String]) {
    // ---- TEST SETUP CODE ---- //
    val m = 2
    val n = 2
    // Generate inputs
    var ADouble = 2 * Matrix.rand[Double](m, n) - 1
    var vDouble = Matrix.rand[Double](n, 1)
    vDouble = vDouble / Matrix.norm(vDouble)
    // Calculate scaling
    val alpha = 2 * max(Matrix.norm(ADouble, "inf"), Matrix.norm(ADouble, "L1"))
    ADouble = ADouble / alpha
    // Convert to bitstream
    var A = ADouble.toSBitstream
    var v = vDouble.toSBitstream

    var err = 1.0
    var sigma = SBitstream(1.0)
    var u = Matrix[SBitstream](m, 1)
    // instantiate module
    var dut = Module(DefaultParams)
    // ---- TEST SETUP CODE ---- //
    // ---- EXPERIMENT RUN CODE ---- //
    var T = 0
    do {
      dut.loop(A, v) match {
        case (uNew, vNew, sigmaNew) => {
          u = uNew;
          v = vNew;
          sigma = sigmaNew;
        }
      }
      // Update error
      err = Matrix.norm(alpha * (A.toDouble * v.toDouble - u.toDouble * sigma.
          value * math.sqrt(n)))
      T += 1
    } while(err > epsilon) // running until we reach app tolerance
    // ---- EXPERIMENT RUN CODE ---- //
    println(f"  err: $err%.4f")
    println(f"  T:   $T%d")
  }
}
```

Listing 5. Example test harness in BitSAD.

4 HARDWARE GENERATION

The most powerful feature in BitSAD is the ability to translate a program to hardware without user intervention. To illustrate this, we will briefly introduce an algorithm, the power iteration method

for computing the singular value decomposition (SVD) of a matrix, and then we will describe the algorithm in BitSAD and translate the program to hardware.

4.1 Iterative SVD

Singular value decomposition is a critical component of computer vision applications such as homography estimation and decomposition. These algorithms are used to extract motion information for robots from two views of the same object at different locations. As a result, being able to compute the SVD is essential to vision-based path planning and navigation. Algorithm 1 provides an overview of the power-iteration method for computing an SVD.

ALGORITHM 1: Iterative SVD

Require: Input matrix $A \in \mathbb{R}^{m \times n}$ and initial guess $v_0 \in \mathbb{R}^n$

1: **for** $k = 1, 2, \ldots$ (until convergence) **do**
2: $\quad w_k = A v_{k-1}$
3: $\quad \alpha_k = \sqrt{w_k^\top w_k}$
4: $\quad u_k = w_k / \alpha_k$
5: $\quad z_k = A^\top u_k$
6: $\quad \sigma_k = \sqrt{z_k^\top z_k}$
7: $\quad v_k = z_k / \sigma_k$
8: **end for**
9: **return** First left/right singular vectors, u_k & v_k, and first singular value, σ_k

4.2 Iterative SVD as a BitSAD Module

As shown below, Listing 6 is a BitSAD program that implements Algorithm 1. Lines 1–9 define the parameters of the hardware. This is not mandatory, but it is recommended, since it allows the user to define a *module* that encapsulates a single algorithm. The module definition is given on lines 11–39. To define a module in BitSAD, the user must specify a `loop()` function, and a list of the output signal names, `outputList`.

We call the reader's attention to lines 17–30, which implement the iterative SVD algorithm. Compare these lines to the mathematical definition in Algorithm 1. The code essentially mimics the math—this is a good example of the intuitive, one-to-one mapping algorithm designers can expect when writing BitSAD code.

4.3 Generating Hardware

BitSAD ships with a Scala compiler plugin that parses a program like Listing 6. The `Module` class is parsed for a `loop` function. We extract the dataflow graph of the program as in Figure 4(a). Then, for each node in the graph, we generate the corresponding Verilog to implement synthesizable hardware for that operation. Figure 4(b) shows the top-level schematic for the hardware that implements the graph in Figure 4(a). Since stochastic bitstreams represent positive numbers between zero and one, a single block, like matrix-matrix multiply, is implemented by several hardware units to keep track of signed computation. The intent of Figure 4 is to illustrate how complex a hardware implementation of Algorithm 1 can be and how BitSAD abstracts this detail away from the designer.

Figure 5 provides a detailed view of the flow of the BitSAD compiler, and it explains our additions to the compiler.

```
1 trait Parameters {
2   val m: Int
3   val n: Int
4 }
5 object DefaultParams extends Parameters {
6   val m = 2
7   val n = 2
8 }
9 case class Module (params: Parameters) {
10   // Define outputs
11   val outputList = List(("v", params.n, 1),
12                         ("u", params.m, 1),
13                         ("sigma", 1, 1))
14   def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
15           (Matrix[SBitstream], Matrix[SBitstream], SBitstream)
16       = simulatable({
17     // Update right singular vector
18     var w = A * v
19     // :/ is fixed-gain division
20     // functionally the same as division
21     // more efficient implementation for dividing by a constant
22     // refer to BitSAD v1 paper for more details
23     var wScaled = w :/ math.sqrt(params.m)
24     var u = wScaled / Matrix.norm(wScaled)
25
26     // Update left singular vector
27     var z = A.T * u
28     var zScaled = z :/ math.sqrt(params.n)
29     var sigma = Matrix.norm(zScaled)
30     var _v = zScaled / sigma
31
32     (u, _v, sigma)
33   })
34 }
```

Listing 6. BitSAD program for iterative SVD.

4.4 Functional Composition and Modules

As mentioned in Section 4.2, BitSAD is designed to express algorithms as modules. This is analogous to traditional hardware design, where the full system is broken into a hierarchy of smaller sub-systems, and each sub-system is individually implemented. There can be several layers of sub-systems in the hierarchy. For example, consider a designer building a deterministic bitstream system to implement a hearing aid [7] that is composed of several filters. In Listing 7, a separate BitSAD program already implements a bi-quad filter in `PDMBandpassFilterBQ.scala`. Line 2, highlighted in red, imports this module into the hearing aid program. Then, Lines 19–22, also highlighted in red, instantiate four instances of the bi-quad filter module. Now, in our `loop()` function, we can call each instance's `loop()` function to evaluate that instance. Thus, BitSAD v2 extends hardware generation to support *functional composition*, which allows designers to create several layers of hierarchy. This is a **major improvement to the language, as it enables designers to logically break down complex systems**. Under this framework, users gain the flexibility to test sub-systems individually, allowing them to quickly find the source of errors. Furthermore, we can estimate the latency for each sub-system separately to determine if a specific module is a bottleneck in a design.

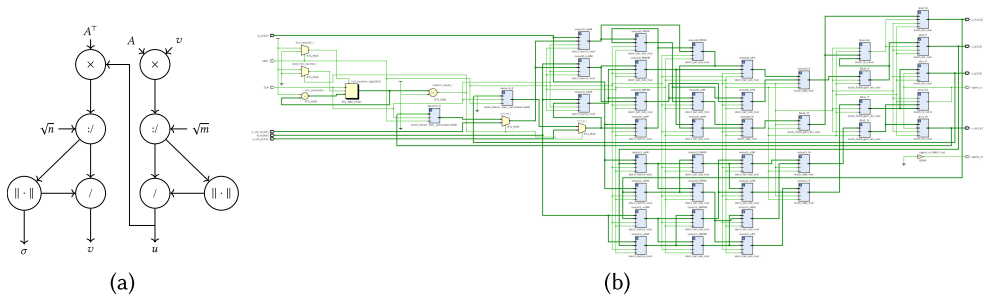


Fig. 4. (a) A DFG of the example code. (b) Top-level RTL schematic generated by BitSAD compiler. Blue blocks in RTL diagram are high-level blocks (e.g., matrix multiply of positive matrices).

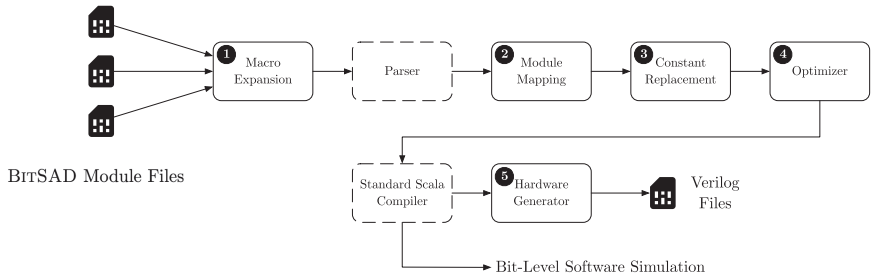


Fig. 5. A diagram of the compiler flow for BitSAD. The user can specify several hardware modules in different Scala files. The `loop` function supports functional composition—that is, a module can make a call to another module's `loop` function. Standard Scala compiler phases are in dashed boxes, and our additions are in solid boxes. ❶ Uses macros to modify the syntax tree before the compiler to make calls to operator wrappers syntactically correct. It also applies population coding (discussed in Section 6). ❷ Module mapping parses the various Scala files to determine the hardware RTL hierarchy. ❸ Finds expressions (potentially complex) that are known statically and then evaluates and replaces them in the code. ❹ Performs the optimizations discussed in Section 5. ❺ Recursively parses the `loop` function of the top-level module to generate the corresponding Verilog.

4.5 Hardware Results

To illustrate the benefits of bitstream computing, we use BitSAD to generate Verilog for several kernels from the BitBench suite [7]. Floating point (FP) and fixed point (FXP) baselines are created using Vivado High-Level Synthesis (HLS). All Verilog designs are synthesized using Xilinx Vivado 2018.2, and we map the resulting designs to Lattice FPGAs. Bitstream computing is beneficial in applications where few resources are available and the power budget is *strict*. This setting makes ultra-low power FPGAs, such as Lattice FPGAs, a fitting choice for these applications. Many of the floating point and fixed point designs consume too many resources to fit on a Lattice FPGA, but we give the baselines the benefit of the doubt and artificially partition the designs across multiple FPGAs. The iterative SVD and linear solver designs are implemented with stochastic bitstreams and run for 200,000 clock cycles. The final application error is 1%, which is the same tolerance required by the FP and FXP designs. Stochastic bitstreams are generated using an LFSR, but we do not include these in the hardware costs. As mentioned in the benchmark [7], these kernels are meant to be end-to-end stochastic circuits, so the LFSRs would not exist in real systems, and they are only a side-effect of our testing environment. The SVF, BQ, and MA filters are deterministic

```
import bitstream.types._, bitstream.simulator.units._
import PDMBandpassFilterBQ.{Module => filter_bq_top, Parameters => BQParameters}
trait Parameters { /* skipped for brevity */ }
object DefaultParams extends Parameters {
  val f1Params = new BQParameters {
    val b0 = 0.5
    val b1 = 0
    val b2 = -0.5
    val a1 = 0.000000000000000122515
    val a2 = 0.0625
    val delay = 1563
  }
  val f2Params = new BQParameters { /* similar to f1Params, skipped */ }
  val f3Params = new BQParameters { /* similar to f1Params, skipped */ }
  val f4Params = new BQParameters { /* similar to f1Params, skipped */ }
}
case class Module (params: Parameters) {
  // internal units
  var filter1 = filter_bq_top(params.f1Params)
  var filter2 = filter_bq_top(params.f2Params)
  var filter3 = filter_bq_top(params.f3Params)
  var filter4 = filter_bq_top(params.f4Params)
  var sdm = SDM()
  // Define outputs
  val outputList = List(("y", 1, 1))
  def loop(x: Bit): Int = {
    // we can call each submodule's loop() function to evaluate it
    var y_f1 = filter1.loop(x)
    var y_f2 = filter2.loop(x)
    var y_f3 = filter3.loop(x)
    var y_f4 = filter4.loop(x)
    // Accumulate all filters
    var y = sdm.evaluate(y_f1 + y_f2 + 2 * y_f3 + 2 * y_f4)
    y
}}
```

Listing 7. BitSAD program for hearing aid.

bitstream designs, and they are evaluated on one second of audio data [15]. Deterministic bitstreams are generated by sampling the original audio at 3 MHz.

Figure 6(a), (b), and (c) show the area, power, and energy results, respectively. As expected, the bitstream computing designs consume fewer resources and, as a result, consume less power as well. Yet, due to the latency issue, the stochastic computing designs consume more energy. This can be addressed using various techniques, such as our population coding in Section 6 or using operators that leverage correlation [25]. This issue is still an active topic of research, and we hope our results and BitSAD will push researchers to examine it more closely.

5 CODE OPTIMIZATION

Creating a language whose syntax is functional, like software, but with the end-goal of generating hardware, presents a unique set of potential optimizations. In particular, since BitSAD encourages users to write their code exactly like the mathematical definition, the written programs can contain several unnecessary operations that could be algebraically reduced. Typical compilers perform algebraic-like optimizations, such as constant propagation, constant folding, and strength reduction, but not very aggressively. There are many reasons for this, the foremost being numerical

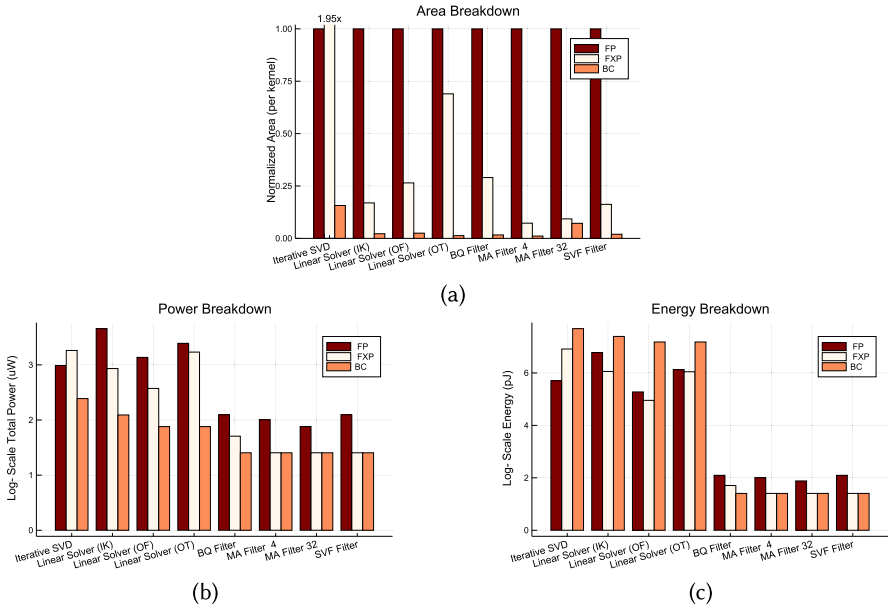


Fig. 6. (a) Normalized area consumption relative to floating point implementation. Area is computed as # of LUTs + # of FFs. ((b) and (c)) Power and energy consumption for different implementations of the iterative SVD, linear solver, and filters on FPGAs. FP is floating-point, FXP is fixed-point, and BC is bitstream computing.

stability. When an expression can contain varied types, algebraically manipulating the syntax tree can result in different type conversions from the original expression, while reordering operations can affect cumulative precision, leading to numerical instability. Moreover, the benefits provided by such optimization are may not be significant for typical programs, and for high-performance code, it is almost always better to leave such optimizations to the user.

Yet, BitSAD code is different. First, the cost of extra operations is much higher—additional hardware units, meaning more area and power. Furthermore, as we will show, depending on the type of the inputs to an operation, the corresponding hardware module can have widely different costs. Last, since we have detailed information about the hardware executing the algorithm, we can make safe transformations that do not compromise stability. It is worth noting that the optimizations in this section apply to deterministic bitstreams. Optimizations might be possible for stochastic bitstreams as well, but we leave that discussion to Section 7.

5.1 Optimization Example — State Variable Filter

Consider the state variable filter, first shown in Figure 2 in Section 2.2. A BitSAD program implementing the dataflow graph exactly as given is shown in Listing 8 (important lines highlighted in red). DFGs like the one in Figure 2 are common among signal processing engineers for illustrating a given digital filter design.

Unfortunately, the most natural way for a designer to express their filter is not the most efficient implementation computationally. First, notice the multiplication by f on line 7, which is over an additive expression. On a bitstream substrate, distributing the multiplication might be more efficient. Indeed, the expression `x - d2 - q * d1_old` is a fixed point number, so `f * (x - d2 - q * d1_old)` is a fixed point multiplication. In contrast, if we distribute the multiplication,

```
val d1_old = delay1.pop; val d2_old = delay2.pop // Get delay buffer values
// Update SDM outputs
val d2 = sdm2.evaluate(f * d1_old + d2_old)
val d1 = sdm1.evaluate(f * (x - d2 - q * d1_old) + d1_old)
delay1.push(d1); delay2.push(d2) // Push new values into delay buffers
```

Listing 8. Example code for SVF.

`f * x - f * d2 - (f * q) * d1_old`, we gain more multiplies, but each multiply is between a fixed point constant and a bitstream. Since a deterministic bitstream is just ±1 at a give time step, each multiply now amounts to a mux that chooses to the flip the sign of the fixed point constant or not. The resulting hardware consumes significantly lower area. This kind of optimization, where expressions are manipulated to use cheaper operators, is commonly referred to as *strength reduction* in classical compiler literature. We adopt the same nomenclature for this bitstream computing variant. Listing 9 provides an optimized version of the original example (we skip the extraneous lines for brevity).

```
val d2 = sdm2.evaluate(f *- d1_old + d2_old)
val d1 = sdm1.evaluate(f *- x -- f *- d2 -- (f *- q) *- d1_old + d1_old)
```

Listing 9. Example code for SVF with strength reduction.

Still, this code can be further optimized. In particular, the sub-expression `-(f * q) * d1_old + d1_old` represents the addition of same variable multiplied by constants ($-(f*q)$ and 1). Since constants are known at compile time (and more importantly, static in the generated hardware), we can "combine like terms" to reduce the number of operations performed on `d1_old`. Listing 10 shows the final version of the original example using this new optimization we refer to as *algebraic simplification*.

```
val d2 = sdm2.evaluate(f *- d1_old + d2_old)
val d1 = sdm1.evaluate(f *- x -- f *- d2 + (1 -- f *- q) *- d1_old)
```

Listing 10. Example code for SVF with strength reduction and algebraic simplification.

Furthermore, it is worth noting that the specific application of algebraic simplification in Listing 10 would not be possible with the previous strength reduction optimization. In a traditional compiler setting, the distribution of f over the additive expression would have been avoided to eliminate concerns about numerical stability. In our setting, this transformation is safe, because the resulting multipliers are muxes to flip the sign bit that are numerically accurate for all values of f. But more importantly, a single multiply is always cheaper than several multiplies in a traditional setting. Simply put, the difference in cost of operators depending on input type is not present in normal code, but it is abundant in bitstream computing.

5.2 Optimization Example—Moving Average Filter

While the optimizations just introduced present clear value in the case of the SVF, there are situations where such optimizations may hurt the results. Consider a moving average (MA) filter that

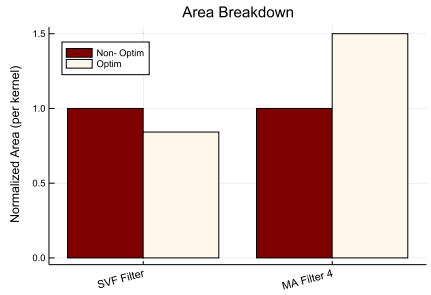


Fig. 7. Normalized area (# LUTs + # FFs) with and without optimizations. In some cases, optimization helps (SVF Filter), but in some cases, it hurts (MA Filter 4). BitSAD uses standard cell library area metrics to estimate the area of the optimized and non-optimized designs, then it only applies the optimization when it is useful.

is simply described by Equation (6),

$$y_k = \frac{1}{n} \sum_{i=k}^{k-(n-1)} x_i. \tag{6}$$

Listing 11 is a program the implements Equation (6) for $n = 4$. In this example, all terms in the additive expression are bitstreams. This is different from the SVF example where each term is a fixed point number (e.g., `f * x` is FXP since `f` is FXP). Since each term is a bitstream, with the exception of the initial addition, we are adding a bitstream and FXP at each node in the expression. While the addition is still FXP addition, the bitstream is mapped to the fixed point representation (either +1 or −1) using a mux. As a result, most of the bits in the bitstream operand are hard-wired at the time of hardware synthesis. In contrast, when we distribute `(1 / n)` over the additive tree, each term becomes a fixed point number, and every bit is considered a variable.

```
1 // Get delay buffer values
2 val d1_old = delay1.pop
3 val d2_old = delay2.pop
4 val d3_old = delay3.pop
5
6 // Update SDM outputs
7 val y = sdm1.evaluate((1 / n) * (x + d1_old + d2_old + d3_old))
8
9 // Push new values into delay buffers
10 delay1.push(x)
11 delay2.push(d1_old)
12 delay3.push(d2_old)
```

Listing 11. Example code for MA filter of length 4.

This presents a problem, because when the bits are hard-wired, the synthesis tool is able to perform LUT reduction to reduce the area consumption (the details of the Xilinx synthesis tool are not openly available, but we can observe this behavior based on the output reports). In contrast, for the SVF example, the addition expression operated on variable fixed point numbers (no bits hard-wired), so our optimization did not prevent the synthesis tool from a potential resource reduction. Figure 7 illustrates this difference. Thus, it is important to consider the hardware area costs associated when deciding when to apply strength reduction.

5.3 Optimization Implementation

To perform these optimizations, we introduce a new phase into the Scala compiler that runs immediately after the syntax tree is generated. This phase manipulates the tree as described in Section 5.1 and passes it on to the remaining phases of the compiler. To effectively reason about when strength reduction and algebraic simplification are possible, we synthesize all the operators using a TSMC 45-nm standard cell library. We then provide the resulting area costs to the compiler phase in a CSV file. Even though BitSAD typically targets FPGAs, we use a standard cell library, because this gives us an accurate relative area estimate between operators. In contrast, FPGA synthesis tools can have widely different implementations of small designs such as individual operators. For this reason, we consider area estimates of individual operators from FPGA synthesis tools as inaccurate. Using the standard cell metrics, the optimizer performs a first-order estimate of the area with and without an optimization, and it only executes the optimization when the area is lowered.

6 POPULATION CODING

As mentioned in Section 3.1, stochastic computing designs suffer from higher latency than floating point and fixed point designs. To alleviate this, we introduce a parallelization approach that takes inspiration from biology—*population coding*. Neurons typically perform approximate computation, so the brain uses multiple sets of similarly connected neurons to perform the inaccurate computation many times in parallel. The aggregate result across all populations is considered more accurate.

This behavior can easily be mapped to stochastic bitstreams by recalling the definition in Equation (1) in Section 2.1. The floating point number being represented, p, is not encoded in time but in the mean of a probability distribution. If we can *generate extra samples of this distribution every time step*, then we can approximate this mean more quickly. To do this, population coding arranges samples of this distribution in both time and space:

$$\bar{p} = \frac{1}{T_{\text{pop}}} \sum_{t=1}^{T_{\text{pop}}} \frac{1}{N} \sum_{i=1}^{N} X_{i,t} \qquad \text{where } T_{\text{pop}} = T/N. \tag{7}$$

Instead of instantiating a single instance of an SC compute block, we can instantiate N populations. The input to each population is an independent sample of the original input created using decorrelators [1]. Decorrelators are units that accept an input stochastic bitstream and produce an output bitstream with the same mean that is statistically independent of the input bitstream. In other words, decorrelators are a mechanism to generate independent and identically distributed (i.i.d.) samples of our input bitstreams.

Finally, to evaluate the resulting output, we can average the output across all N populations and then average that result over time (shown in Equation (7)). This process is illustrated in Figure 8(a). In Equation (2), the output was the average of T samples. To obtain the same accuracy in the population coded version (i.e., take the average over the same number of samples), we need $T_{\text{pop}}N = T$. For example, if $T = 200000$ clock cycles is too long in terms of latency, then we can instantiate 10 populations of the compute block. Now, the entire system is run for only $T_{\text{pop}} = 200{,}000/10 = 20{,}000$ clock cycles, but the output accuracy is equivalent to the single population implementation.

We provide experimental verification of population coding Section 6.2. The differences between our approach to parallelization and prior work is discussed in Section 6.1, and Section 6.3 discusses how our approach can reduce the hardware cost of parallelization. Finally, Section 6.4 provides theoretical guarantees about population coding.

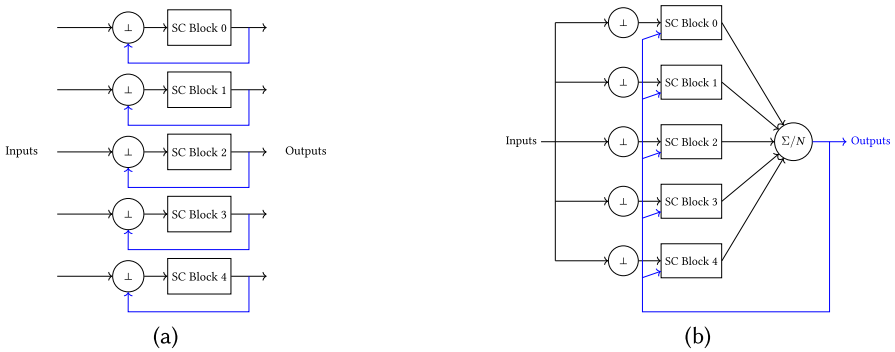


Fig. 8. (a) A block diagram of population coding applied to a generic stochastic computing (SC) block naively. Each population works in parallel, and the results are average across all populations *after* the algorithm terminates. This is a streaming version of prior work discussed in Section 6.1. (b) A block diagram of population coding applied to a generic SC computing block. The $\perp$ units indicate decorrelators and the Σ/N indicates an averaging unit. By averaging stochastically, we automatically decorrelate the output from the input, and we do not need decorrelators on the feedback path (highlighted in blue). Discussed in Sections 6.3 and 6.4.

6.1 Differences between Population Coding and Parallel Encoding

The generation of new samples (via decorrelators) differentiates our approach to parallelizing stochastic circuits with respect to Miao and Chakrabarti's work [18]. Parallel encoding of stochastic circuits organizes T samples into N subsets of size T/N. There are N stochastic circuits, and each stochastic circuit processes one subset. Since each subset is of length T/N, parallel encoding of stochastic circuits reduces the latency linearly, much like population coding. *But this is only reasonable when the inputs and outputs of a system are binary.* The prior work assumes that conversion between binary and stochastic bitstreams happens at the input and output of any stochastic system. This makes parallel encoding feasible, because all the samples are known *a priori* and rearrangement is possible. Yet, as References [5, 10] argue, there exist ultra-low power applications where the sensory data is already a bitstream. Under these constraints, using binary data is an unnecessary and costly bottleneck, but without the ability to generate all T samples prior to compute, the parallel encoding scheme would not be possible. In other words, we cannot arrange T samples into N subsets as parallel encoding requires, because in end-to-end streaming applications, only a single sample (as opposed to T samples) is available to the system at each time step. Population coding avoids this by using decorrelators to generate *new* i.i.d. samples.

We note that there are existing parallel encoding schemes for deterministic computing [9, 19, 24], which are discussed in more detail in Section 8.

6.2 Experimental Results of Population Coding

Population coding is verified by our experimental results shown in Figure 9(a). We apply population coding to the singular value decomposition block generated by Listing 6. As the number of populations increases, the time to convergence decreases linearly. Eventually, increasing the number of populations will not help convergence as shown by the 50 population and 100 population curves in Figure 9(a). This is because there is a minimum iteration count required for each sub-block to converge.

Figure 10(a) shows the area results for the stochastic computing designs from Figure 6(a) with population coding applied, and Figure 10(b) shows the power results. As expected, increasing the number of populations consumes more area and power, but the results are still an order of magnitude lower than the FP/FXP designs.

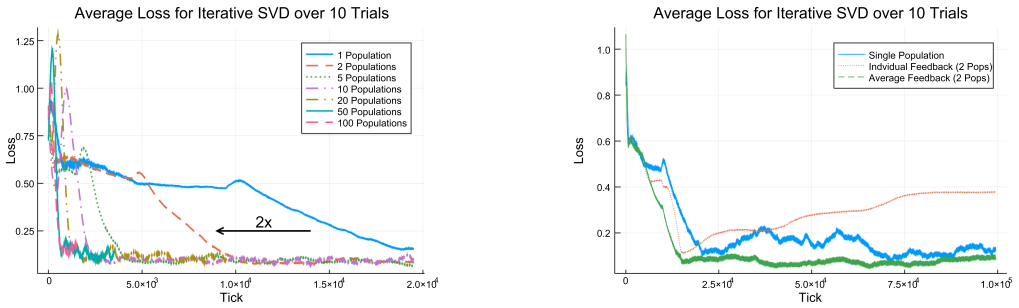


Fig. 9. (a) Average loss for iterative SVD for varying populations. As the number of populations increases by n, the time to convergence decreases by n. (b) Iterative SVD loss over 100,000 iterations with no decorrelators on feedback paths. The implementation in Figure 8(a) diverges (blue curve), but Figure 8(b) converges (green curve).

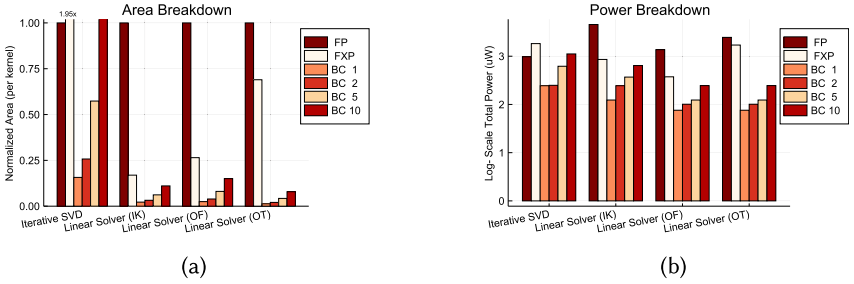


Fig. 10. (a) Area consumption of stochastic bitstream designs with population coding applied ($N = 10$). (b) Power consumption of stochastic bitstream designs with population coding applied. FP is floating-point, FXP is fixed-point, and BC is bitstream computing.

Last, recall in Section 3.1.1 that the magnitude of the floating point numbers being estimated can increase the required number of samples to produce an accurate result. This is especially challenging when the inputs are scaled to prevent saturation. In these cases, population coding is a practical way to process more samples without increasing latency.

6.3 Removing Decorrelators on Feedback Paths

Figure 8(a) illustrates a naive implementation of population coding. As mentioned in the prior section, decorrelators are used to ensure each population sees an independent sample of the input bitstream distributions. However, instead of averaging across populations at the end of the computation, we can average stochastically every time step as shown in Figure 8(b) (using a stochastic averaging unit [1]). When the average across populations is computed using a stochastic averaging unit, the output of the unit is decorrelated from the inputs to each SC block. As a result, there is no need for decorrelators on the feedback paths. This results in significant area and power savings, since each decorrelator consists of an linear feedback shift register (LFSR) that consumes many flip-flops. When the output is a vector or matrix, a decorrelator is required for every element, so the power consumption associated with the LFSRs is a significant contribution to the overall dynamic power. Figure 9(b) shows experimentally that naively removing the decorrelators on the feedback paths causes the loss to diverge for the implementation in Figure 8(a) but converges with

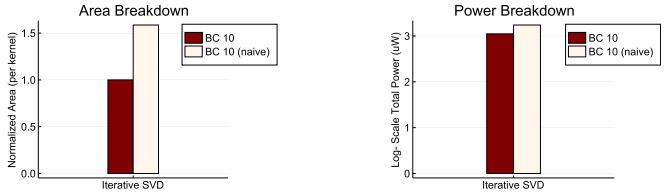


Fig. 11. (a) Area and (b) power with the smart vs. naive implementations of population coding. The smart implementation allows us to remove decorrelators on the feedback paths, which decreases the area cost.

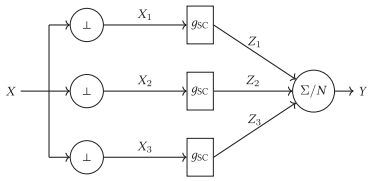


Fig. 12. An "unrolled" population coded stochastic circuit with averaging.

the implementation in Figure 8(b). Figure 11 shows the area and power reduction for the iterative SVD design by removing decorrelators.

6.4 Mathematical Analysis of Population Coding with Stochastic Averaging

In the previous section, we demonstrated empirically that stochastic averaging provides a decorrelating effect that leads to proper convergence. Here, we provide a mathematical foundation for that observation. Consider the "unrolled" population coded stochastic circuit in Figure 12. The output of the circuit, Y, is usually fed back into each population; by removing this feedback path, we are unrolling the recurrent behavior of the circuit and only examining the feedfoward behavior. All labeled nodes in the circuit are Bernoulli random variables. g_{SC} is a function that implements any stochastic circuit. Our goal is to show that Y is sufficiently decorrelated from X as the number of populations increases.

Theorem 6.1. *Given a stochastic circuit implemented by function g_{SC} and input random variable, $X \sim \text{Ber}(p)$, let f_X be the probability mass function (PMF) of X, and let $X_1, \ldots, X_N \overset{i.i.d.}{\sim} f_X$ be N decorrelated copies of X. Define $Z_i = g_{SC}(X_i)$ as the output of the i-th population, and define $Y \sim \frac{1}{N}\sum_{i=1}^{N} Z_i$ as the stochastic average over all N populations. Let $\text{Cov}(X, Y)$ be the covariance between the input and output distributions. Then,*

(1) if $g_{SC}(x) = 0$ or $g_{SC}(x) = 1$ (constant circuit), $\text{Cov}(X, Y) = 0$ (independent of N)
(2) if $g_{SC}(x) = x$ (identity circuit), $\text{Cov}(X, Y) = p(1-p)$ (independent of N)
(3) if $g_{SC}(x) = \neg x$ (inversion circuit), $\text{Cov}(X, Y) = -p(1-p)$ (independent of N)
(4) for conditional identity/inversion circuits (e.g., $g_{SC}(x)$ is identity if $x = 0$ but is an arbitrary function for $x = 1$) as $N \to \infty$,
  (a) when $g_{SC}(x) = 0$ if $x = 0$ or $g_{SC}(x) = 1$ if $x = 1$ (but arbitrary in the other case), $0 < \text{Cov}(X, Y) < p(1-p)$
  (b) when $g_{SC}(x) = 1$ if $x = 0$ or $g_{SC}(x) = 0$ if $x = 1$ (but arbitrary in the other case), $-p(1-p) < \text{Cov}(X, Y) < 0$
(5) else, $\text{Cov}(X, Y) \to 0$ as $N \to \infty$.

In other words, for all stochastic circuits but the trivial cases, population coding with stochastic averaging reduces the correlation of the input and output, which means we can safely feed the output back into each population without the need for decorrelators. Specifically, the input and output are asymptotically independent of each other.

Proof. *(This is a sketch. The full proof can be found in supplemental material.)* Start by noting that each output, Z_i, is a Bernoulli random variable drawn according to a distribution induced by the function g_{SC}. More specifically, there exists two conditional distribution, $f_{Z|X}(z \mid X = 0)$ and $f_{Z|X}(z \mid X = 1)$ that describe the probability that $Z_i = z$ given that $X = 0$ or $X = 1$, respectively. These conditional distributions are sufficient to capture any function g_{SC}. Furthermore, these distributions must be Bernoulli, since Z_i is a Bernoulli random variable; so we can characterize the function, g_{SC}, by two parameters, p_{z_0} and p_{z_1}, such that

$$f_{Z|X}(z \mid X = 0) \sim \text{Ber}(p_{z_0}) \qquad f_{Z|X}(z \mid X = 1) \sim \text{Ber}(p_{z_1}).$$

With these parameters, we begin by using the definition of covariance and conditional probability to derive an expression for the input-output covariance:

$$\begin{aligned}\text{Cov}(X, Y) = \left[p^2(1-p)(p_{z_1} - p_{z_0}) + p(1-p)p_{z_0}\right] \sum_{k=0}^{N} p_{z_0}^k (1-p_{z_0})^{N-k} - p_{z_1}^k (1-p_{z_1})^{N-k} \\ - p(1-p) \sum_{k=\lceil N/2 \rceil}^{N} p_{z_0}^k (1-p_{z_0})^{N-k} - p_{z_1}^k (1-p_{z_1})^{N-k}. \end{aligned} \tag{8}$$

Note that $p_{z_0}, p_{z_1} \in [0, 1]$. Thus, we are concerned with how the covariance behaves over this square domain. We show results (1), (2), and (3) of the theorem by directly substituting $(p_{z_0}, p_{z_1}) = (0, 0), (0, 1), (1, 0), (1, 1)$, which are the corners of the square, into Equation (8). Then, by restricting $p_{z_0}, p_{z_1} \in (0, 1)$ (the interior of the square), we can rewrite the summation in Equation (8) as

$$\sum_{k=0}^{N} p_{z_0}^k (1-p_{z_0})^{N-k} - p_{z_1}^k (1-p_{z_1})^{N-k} = (1-p_{z_0})^N \sum_{k=0}^{N} \left(\frac{p_{z_0}}{1-p_{z_0}}\right)^k - (1-p_{z_1})^N \sum_{k=0}^{N} \left(\frac{p_{z_1}}{1-p_{z_1}}\right)^k.$$

As $N \to \infty$, the expression is easily identified as the product of a convergent sequence, $\lim_{N\to\infty}(1 - p_{z_0})^N = 0$, and convergent geometric series, $\sum_{k=0}^{\infty} \left(\frac{p_{z_0}}{1-p_{z_0}}\right)^k$. Applying the same technique to the other summation in Equation (8), we get the following limit when $p_{z_0}, p_{z_1} \in (0, 1)$:

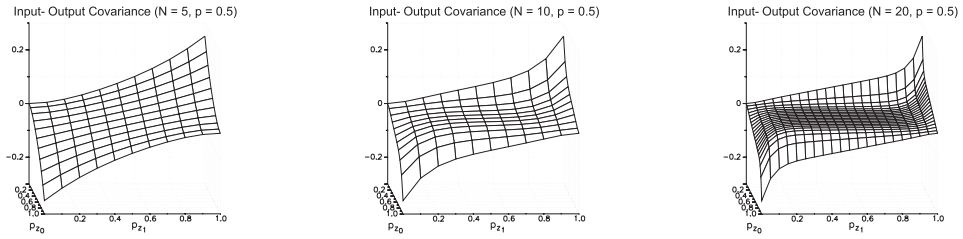
$$\lim_{N\to\infty} \text{Cov}(X, Y) = 0.$$

This proves result (5) of the theorem. Similar techniques can be applied to the borders of the square—fixing one variable and substituting it into Equation (8), simplifying, and then taking the limit. Thus, we can bound the covariance in the border cases to prove result (4) of the theorem. □

Though the techniques used to prove Theorem 6.1 are not enlightening, the process produces Equation (8), which can be qualitatively analyzed to understand how the covariance behaves for different choices of stochastic circuits. As shown in Figure 13, for 20 or more populations, the covariance is essentially zero for the vast majority of stochastic circuits. In other words, we do not require N to be extremely large to confidently feed the output back into the circuit.

6.5 Adding Population Coding Support to BitSAD

The syntax for population coding support in BitSAD is similar to the `simulatable` macro. The `populations` macro takes two arguments—the number of populations and an expression or block of expressions to which population should be applied. The macro then replicates each expression



(a) (b) (c)

Fig. 13. Input-output covariance for (a) 5 populations, (b) 10 populations, and (c) 20 populations. In (c), the majority of the surface is a plane at zero. So, for most stochastic circuits, 20 populations may be sufficient to guarantee decorrelation. Changing the input probability, p, varies the height of the peak at the extrema, so we only show $p = 0.5$, which is the maximum height.

as many times as required, and it gives unique names to all the inputs and outputs. It also adds a call to a stochastic averaging unit to aggregate the outputs. Listing 12 details the `populations` syntax (highlighted in red).

One of the benefits of BitSAD is that designers do not need to explicitly decorrelate their variables. The compiler automatically inserts decorrelators as needed. But our experiments in Section 6.3 show that certain operators, such as a stochastic averaging unit, make decorrelators redundant. The hardware generation mechanism in our compiler plugin uses dataflow analysis to track which edges in a DFG are correlated and apply decorrelators as necessary. This is done by assigning an index set to each input. The index set of the output edge from a node is the union of the index sets of the incoming edges. Units like a decorrelator or stochastic averaging unit "clear" the index set by assign the output edge to a set with new unused index. After assigning index sets to every edge, we can detect missing decorrelators by finding nodes where intersection of the index sets of the incoming edges is non-null. See Figure 14 for a graphical representation of this process.

```
1 def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
2       (Matrix[SBitstream], Matrix[SBitstream], SBitstream)
3       = simulatable(populations(2, {
4   // normal loop code, skipped for brevity
5 }))
```

Listing 12. Population coding syntax in BitSAD.

6.6 Optimal Granularity of Population Coding

While population coding allows a designer to reduce the latency of their design without sacrificing accuracy, for large designs, the cost of replicating the entire system can be significant. It also is not clear that replicating the entire system is required. We study this behavior by considering the DFG in Figure 15, which is a subset of the full iterative SVD algorithm.

Our goal is to evaluate the following:

$$w = \frac{Av}{\sqrt{m}} \qquad u = \frac{w}{\|w\|_2}. \tag{9}$$

We start by creating eight populations of Figure 15, and we measure the time until

$$\|u_{\text{bitstream}} - u_{\text{floating-point}}\|_2 \leq 0.01.$$

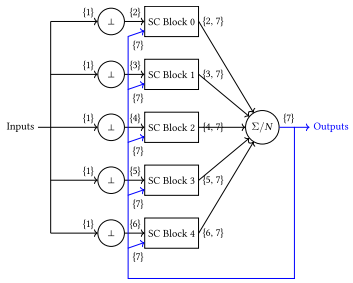


Fig. 14. Dataflow analysis to identify correlation of edges in a DFG. An outgoing edge's index set is the union of the incoming edges' index sets. Certain units clear index sets to indicate decorrelation (e.g., a decorrelator assigns a brand new index set regardless of input).

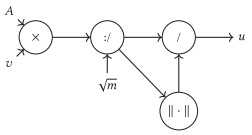


Fig. 15. An example DFG to understand population coding granularity. This is a subset of the iterative SVD algorithm in Figure 4(a).

Table 1. A Comparison of Designs with Population Coding at Different Granularities

Variant	Average Cycles till Convergence	Area (# LUTs + # FFs)
Eight Populations (Full Design)	19,781	6,807
Eight Populations (Multiplier Only)	13,613	4,484

But not all the nodes in Figure 15 take a long time to converge. In this case, the matrix multiply is a more complex block than the other nodes, and it is the bottleneck for convergence. To test this hypothesis, we instantiate eight populations of the matrix multiply, average across the populations, then feed the averaged result to the remaining DFG. Table 1 compares these designs. Even though the latter design only applies population coding to the multipliers, it is able to achieve the lower latency across 10 trials of random inputs, and it consumes approximately 2/3 the area. So, when applying population coding to a design, it is important to identify the true bottlenecks to convergence and only apply population coding to those blocks as opposed to the whole design. Using BitSAD, we can efficiently determine which blocks to add population coding to.

7 FUTURE WORK

Our work only begins to touch the potential intricacies of bitstream computing. There is still work to be done to understand potential extensions of the optimizations we proposed, as well as new optimizations that we have not discovered. The varied types and operators in a single bitstream computing program have widely different costs, and so there are many opportunities for automated code improvement. Here, we focus on one such opportunity: automated selective population coding. In future work, we would like to automate the process of finding the optimal granularity for population coding as discussed in Section 6.6.

Suppose that we have the latency for each node in a DFG. This can be obtained by running a user-provided benchmark of inputs for the node. For the example discussed in the article, we would start by considering population coding applied to the full design. Then, we would greedily

reduce the populations for the node consuming the most area, while making sure the maximum latency across all nodes remains below a user specified real-time deadline. If a node cannot be reduced by population any further, then we move onto the node with the next-largest area. We keep applying this process until all nodes have been passed over.

We also explore the problem of optimal decorrelator placement. As mentioned in Section 6.3, certain units will automatically decorrelate their inputs. Given a DFG with no decorrelators and all the index sets assigned, we would like to determine the optimal position to insert a decorrelator to maximize the number of edges that become decorrelated. This could be framed as a min-cut problem, and the compiler could heuristically solve the min-cut problem to determine the decorrelator placement. Reference [23] addresses a similar problem where they decorrelate bitstreams with delay blocks and propose an algorithm to find an optimal placement of delay blocks. Since the approach to decorrelation is different between Reference [23] and our proposal, we believe that our approach might still prove to be useful.

Last, we intend to refactor the source code for our compiler plugin to improve modularity, so that it will be easier for researchers to contribute their own operators and corresponding hardware modules to BitSAD.

8 RELATED WORK ON DETERMINISTIC BITSTREAMS

Other works [9, 14, 19, 24] have proposed "deterministic bitstreams" (similar to the deterministic bitstreams in this article but not PDM format) to reduce latency. Unfortunately, this work only describes how to implement single or a handful of arithmetic operations and does not discuss how to extend the work to cascaded, generic, dependent, and/or recurrent operations, such as those required for the algorithms explored in this article. To the best of our understanding, this would require buffering between each operation to regenerate the deterministic encoding. For this reason, we feel this scheme is less flexible than population coding, which seamlessly enables streaming computation in both feedforward and feedback configurations. While deterministic bitstreams are useful (see Section 2.2), we do not believe they are useful for reducing latency of *general* designs. Instead, like described by prior work, we see their value for highly specialized designs that consume extremely low resources. Bridging the gap between these designs and the flexible, large-scale designs of this article through language development (like BitSAD) is an interesting and promising endeavor for future work.

9 CONCLUSION

This work proposes several extensions to BitSAD, a DSL for bitstream computing. We add bit-level software simulation, automatic hierarchical hardware generation, and several code optimizations to the compiler. Furthermore, we introduce population coding to alleviate the latency problems associated with stochastic computing, and we add language-level support for population coding. In our future work, we discuss several possibilities for further interesting compiler automation of decisions for the user. We hope our work illustrates the new challenges available to compiler designers when working with bitstream computing DSLs, and it motivates the need to further study this domain to make bitstream computing viable and practical.

REFERENCES

[1] 2016. *IBM Neurosynaptic System Neuron Function Library Reference Manual.* Technical Report. IBM Corporation.

[2] Armin Alaghi and John P. Hayes. 2013. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.* 12, 2s (2013), 1–19. DOI: https://doi.org/10.1145/2465787.2465794

[3] A. H. Bentbib and A. Kanber. 2015. Block power method for SVD decomposition. *Anal. Sti. Univ. Ovid. Const. Ser. Mat.* 23, 2 (2015), 45–58. DOI: https://doi.org/10.1515/auom-2015-0024

[4] Bradley D. Brown and Howard C. Card. 2001. Stochastic neural computation I: Computational elements. *IEEE Transactions on Computers* 50, 9 (2001), 891–905. DOI: 10.1109/12.954505

[5] Taylor S. Clawson, Silvia Ferrari, Sawyer B. Fuller, and Robert J. Wood. 2016. Spiking neural network (SNN) control of a flapping insect-scale robot. In *Proceedings of the IEEE Conference on Decision and Control*. 3381–3388.

[6] Kyle Daruwalla and Heng Zhuo. 2019. BitSAD: A domain-specific language for bitstream computing. In *Proceedings of the 1st ISCA Workshop on Unary Computing*.

[7] Kyle Daruwalla, Heng Zhuo, Carly Schulz, and Mikko Lipasti. 2019. BitBench: A benchmark for bitstream computing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*. ACM Press, New York, NY, 177–187. DOI: https://doi.org/10.1145/3316482.3326355

[8] Pierre Emile Duhamel, Judson Porter, Benjamin Finio, Geoffrey Barrows, David Brooks, Gu Yeon Wei, and Robert J. Wood. 2011. Hardware in the loop for optical flow sensing in a robotic bee. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, 1099–1106. DOI: https://doi.org/10.1109/IROS.2011.6048759

[9] S. Rasoul Faraji and Kia Bazargan. 2019. Hybrid binary-unary hardware accelerator. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference on (ASPDAC'19)*. ACM Press, New York, NY, 210–215. DOI: https://doi.org/10.1145/3287624.3287706

[10] Dario Floreano and Robert J. Wood. 2015. Science, technology and the future of small autonomous drones. *Nature* 521, 7553 (2015), 460–466. DOI: https://doi.org/10.1038/nature14542

[11] B. R. Gaines. 1969. Stochastic computing systems. In *Advances in Information Systems Science*. Springer US, Boston, MA, 37–172. DOI: https://doi.org/10.1007/978-1-4899-5841-9_2

[12] Warren J. Gross and Vincent C. Gaudet. 2019. *Stochastic Computing: Techniques and Applications*. DOI: https://doi.org/10.1007/978-3-030-03730-7

[13] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association* 58, 301 (1963), 13–30. DOI: 10.2307/2282952

[14] Devon Jenson and Marc Riedel. 2016. A deterministic approach to stochastic computation. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD'16)*, 1–8. DOI: https://doi.org/10.1145/2966986.2966988

[15] Mikko Lipasti and Carly Schulz. 2017. End-to-end stochastic computing. In *Proceedings of the 1st Workshop on Pioneering Processor Paradigms*.

[16] Kevin Y. Ma, Pakpong Chirarattananon, Sawyer B. Fuller, and Robert J. Wood. 2013. Controlled flight of a biologically inspired, insect-scale robot. *Science* 340, 6132 (May 2013), 603–607. DOI: https://doi.org/10.1126/science.1231806

[17] Ezio Malis and Manuel Vargas. 2007. Deeper understanding of the homography decomposition for vision-based control. Research Report. RR-6303, INRIA, inria-00174036v3. pp. 90.

[18] Lifeng Miao and Chaitali Chakrabarti. 2013. A parallel stochastic computing system with improved accuracy. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS'13)*. IEEE, 195–200. DOI: https://doi.org/10.1109/SiPS.2013.6674504

[19] Soheil Mohajer, Zhiheng Wang, and Kia Bazargan. 2018. Routing magic: Performing computations using routing networks and voting logic on unary encoded data. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*, 77–86. DOI: https://doi.org/10.1145/3174243.3174267

[20] Rohit Shukla, Erik Jorgensen, and Mikko Lipasti. 2017. Evaluating hopfield-network-based linear solvers for hardware constrained neural substrates. In *Proceedings of the International Joint Conference on Neural Networks*, 3938–3945. DOI: https://doi.org/10.1109/IJCNN.2017.7966352

[21] Rohit Shukla, Soroosh Khoram, Erik Jorgensen, Jing Li, Mikko Lipasti, and Stephen Wright. 2018. *Computing Generalized Matrix Inverse on Spiking Neural Substrate*. 115 pages.

[22] Pai Shun Ting and John Patrick Hayes. 2014. Stochastic logic realization of matrix operations. In *Proceedings of the 2014 17th Euromicro Conference on Digital System Design (DSD'14)*, 356–364. DOI: https://doi.org/10.1109/DSD.2014.75

[23] Pai Shun Ting and John Patrick Hayes. 2016. Isolation-based decorrelation of stochastic circuits. In *Proceedings of the 34th IEEE International Conference on Computer Design (ICCD'16)*. 88–95. DOI: https://doi.org/10.1109/ICCD.2016.7753265

[24] Zhiheng Wang, Soheil Mohajer, and Kia Bazargan. 2018. Low latency parallel implementation of traditionally-called stochastic circuits using deterministic shuffling networks. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'18)*, 337–342. DOI: https://doi.org/10.1109/ASPDAC.2018.8297346

[25] Di Wu and Joshua San Miguel. 2019. In-stream stochastic division and square root via correlation. In *Proceedings of the 56th Annual Design Automation Conference (DAC'19)*. 1–6.

[26] Xuan Zhang, Mario Lok, Tao Tong, Sae Kyu Lee, Brandon Reagen, Simon Chaput, Pierre-Emile J. Duhamel, Robert J. Wood, David Brooks, and Gu-Yeon Wei. 2017. A fully integrated battery-powered system-on-chip in 40-nm CMOS for closed-loop control of. *IEEE J. Solid-State Circ.* 52, 9 (2017), 2374–2387. DOI: https://doi.org/10.1109/JSSC.2017.2705170

Received June 2019; revised August 2019; accepted September 2019