BitBench

A Benchmark for Bitstream Computing

Kyle Daruwalla
University of Wisconsin - Madison
Madison, Wisconsin, USA
daruwalla@wisc.edu

Heng Zhuo
University of Wisconsin - Madison
Madison, Wisconsin, USA
hzhuo2@wisc.edu

Carly Schulz
University of Wisconsin - Madison
Madison, Wisconsin, USA
cschulz8@wisc.edu

Mikko Lipasti
University of Wisconsin - Madison
Madison, Wisconsin, USA
mikko@engr.wisc.edu

Abstract

With the recent increase in ultra-low power applications, researchers are investigating alternative architectures that can operate on streaming input data. These target use cases require complex algorithms that must be evaluated under a real-time deadline, but also satisfy the *strict* available power budget. Stochastic computing (SC) is an example of an alternative paradigm where the data is represented as single bitstreams, allowing designers to implement operations such as multiplication using a simple AND gate. Consequently, the resulting design is both low area and low power. Similarly, traditional digital filters can take advantage of streaming inputs to effectively choose coefficients, resulting in a low cost implementation. In this work, we construct six key algorithms to characterize bitstream computing. We present these algorithms as a new benchmark suite: BitBench.

CCS Concepts • **General and reference** → **Evaluation**; *Metrics*; • **Computer systems organization** → *Real-time systems.*

Keywords bitstream, stochastic computing, benchmark, pulse density modulation, deterministic bitstream

ACM Reference Format:
Kyle Daruwalla, Heng Zhuo, Carly Schulz, and Mikko Lipasti. 2019. BitBench: A Benchmark for Bitstream Computing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3316482.3326355

LCTES '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6724-0/19/06…$15.00
https://doi.org/10.1145/3316482.3326355

1 Introduction

Embedded systems that require high performance hardware for signal processing, optimization, and control have historically relied on optimized fixed-point algorithms deployed on low-power microprocessors, digital signal processors, or FPGAs. However, 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. Conventional computing substrates require conversion both on the input side and on the output side in order to effectively interface with physical systems that utilize bitstream representations. Alternatively, the data can be processed efficiently using approaches from stochastic computing (SC), which enables extremely inexpensive arithmetic operations (e.g. a single AND-gate for multiplication). Similarly, the oversampled nature of the data can be leveraged to produce binary-friendly constants with fixed point data processing. We divide these two regimes into **stochastic bitstream** and **deterministic bitstream** computing. In this paper, we present several algorithms (6 kernel benchmarks and 2 end-to-end applications) that are characteristic in typical bitstream processing applications. Additionally, we highlight several interesting features of the bitstream paradigm such as fault tolerance and approximate computing.

Recent advancements in control systems and MEMS fabrication has enabled new power constrained applications such as pico aerial vehicles (PAVs) [1] (e.g. RoboBee [2] [3]). Due to the limited power budget in PAV applications (< 350 mW for the full robot [3]), microcontrollers are not an option for the control system, and current work uses ASICs instead [4]. Prior works [5] and [6] have considered a spiking neural network based controller for stabilizing the flapping insect-scale robot. Similarly, other works such as [3] and [7] have proposed an ASIC that can perform optical flow based control, while keeping the computations in the acceptable power budget for RoboBee. Unfortunately, these ASICs can only perform a limited set of algorithms (e.g. optical flow) that are required to keep the robotic bee in flight.

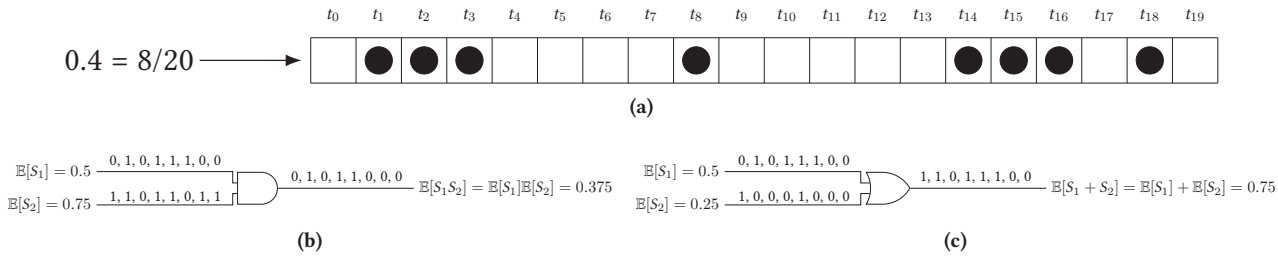

Figure 1. This figure shows how data is represented and computations are performed using stochastic computing. (a) The values 0.4 is represented using a stochastic data representation scheme. The probability of occurrence of a bit at a particular time tick is directly proportional to its value. To represent the value 0.4 over 20 time ticks, there should be 8 random occurrences of a high bit. (b) Since the numbers can be represented as independent streams of bits, 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 multiplication results would be. (c) An example of scaled addition using an OR gate. Notice that the output bitstream does not exactly match the expected result. This occurs due to the randomness of each bit. To produce accurate results, bitstreams should be highly uncorrelated, and the bitstream should be sufficiently long.

Initial work indicates that a RoboBee should be able to perform the duties of biological bees, perform aerial surveillance of crops, collect weather data, and assist search and rescue teams [2], but current implementations do not support these functions, because they lack a computing platform capable of performing these complex algorithms while consuming less than 10% of the available power budget [2]. Stochastic bitstream computing provides an intuitive trade-off between approximation and energy consumption which allows SC algorithms to be easily tuned dynamically to the constraints of these ultra-low power systems. *To address this application space, we present 2 kernels: a least-squares solver and singular value decomposition.*

Similar to stochastic bitstreams, deterministic bitstreams occur naturally in low power signal processing applications. Such bitstreams are prevalent in raw sensor data formats such as PDM audio. Typically, these raw streams are encoded into binary, processed in binary, then re-encoded into a bitstream to drive actuators. By using bitstream processing, we can eliminate the need for energy-hungry data conversion [8]. Furthermore, we show that a bitstream based design can be more storage and resource efficient. *To address this application space, we present 4 kernels: state-variable filter, bi-quad filter, moving average filter (width of 4), and moving average filter (width of 32).*

With these 6 kernels, we present **BitBench**, a **benchmark suite** that provides area/power/energy breakdown for end-to-end applications in the **bitstream** domain. In Sec. 2, we provide background on stochastic computing (SC) and pulse density modulation (PDM), then we introduce two applications that benefits from bitstream computing in Sec. 3. Finally, we provide details about the 6 kernels in Sec. 4, followed by evaluation and results in Sec. 5.

2 Background

In this section, we introduce the key principles such as stochastic computing and pulse-density modulation, which enable much of the work in this paper.

2.1 Stochastic Computing and Algorithms

Stochastic computing is a technique that represents values as continuous streams of random bits [9]. Fig. 1a shows how a unipolar value $\in [0, 1]$ is represented using a stochastic data representation scheme. Prior work that has considered robotics tasks such as optical flow and inverse kinematics use SNNs [10] [11] [12] or approximate computing using neural processing engines [13]. These implementations have relied on offline training of low-precision networks with prior knowledge about the possible set of output values. Stochastic computing provides a distinct advantage over these systems, since we can perform the same inverse kinematics task without any prior knowledge about the possible set of output values.

Even though the digital circuitry to perform stochastic computing is simple [14], these computation techniques expose an energy-efficiency and accuracy trade-off.

2.2 Pulse Density Modulation

In modern audio recording applications, audio is sensed in an oversampled bitstream format called pulse density modulation (PDM). In this bitstream, a larger density of "1" bits is used to represent a larger amplitude. Note that these bitstreams are deterministic. Currently, PDM data is converted to binary samples pulse code modulation (PCM) for processing, then re-encoded as a PDM stream to drive an amplifier circuit. The conversion operations can introduce loss and waste energy.

By keeping audio in PDM format through the entire processing pipeline, we can effectively tune filter coefficients to maximize storage efficiency and reduce required bit precision.

3 End-to-End Applications

Now, we present two end-to-end applications that utilize the kernels discussed later in Sec. 4.

3.1 Drone Stabilization

While many possible applications can be composed from the presented kernels, we will focus on the most critical application for PAVs — stable flight. Given an on-board front-facing camera, the robot must compensate for noise or external force along three axes. Though there are many sensors available on commercial drones that assist stabilization, the weight and power requirements of PAVs implicitly require that only the fewest number of sensors are included in the system. Since a camera is included by default, it would be ideal if the drone could stabilize the PAV with no additional sensors.

A common visual mechanism to detect motion is optical flow. If we can detect minor, unexpected motion, then we can compensate for this motion to keep the drone stable. Fig. 2 provides a system block diagram of this pipeline. In this work, we specifically focus on Lucas-Kanade algorithm for optical flow [15]. Suppose we are given an start frame, I_s, and some external force or noise moves the drone, resulting in a final frame, I_f. We start by computing the spatial (I_x, I_y) and temporal (I_t) derivatives of I_s and I_f. These are given by

$$I_x = K_x * I_s \tag{1}$$
$$I_y = K_y * I_s \tag{2}$$
$$I_t = K_t * I_f - K_t * I_s \tag{3}$$

where K_x, K_y, and K_t are convolution kernels defined as:

$$K_x = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} \quad K_y = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \quad K_t = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

From these, we can define the least-squares problem in Lucas-Kanade optical flow:

$$A = \begin{bmatrix} i_x & i_y \end{bmatrix} \quad B = -i_t$$

where i_x, i_y, and i_t denote vectorized representations of I_x, I_y, and I_t.

Performing optical flow means solving $AX = B$ for the flow vector, X. We can translate this flow vector, X, into the corresponding deflection in pitch or roll, then drive the motors/actuators to control for the deflection.

In Sec. 4.1, we demonstrate how a least-squares problem (written as $AX = B$) can be solved using stochastic bitstreams. Thus, the pipeline in Fig. 2 is converted completely to the bitstream domain. Our results show a power improvement of 8× for optical flow when compared to the ASIC used in the RoboBee [4].

3.2 Hearing Aid

Digital audio filtering has a wide range of uses. Many of the filter kernels in this work are broadly applicable; however, we chose to focus on a case study where low energy consumption is critical:: a hearing aid. A digital hearing aid contains many signal processing steps including noise filtering, frequency shaping, and amplification. The kernels described in Sec. 4.3 are used to address the latter two stages. Fig. 3 illustrates an overview of this system.

Our system is designed with four bandpass filter banks centered at 500 Hz, 1250 Hz, 2875 Hz, and 6000 Hz. These cover the typical range of human auditory deficiencies that need to be amplified [16]. Additionally, most hearing aid users have trouble discerning higher frequencies, so these channels have greater amplification added to them [17]. Thus, our system separates input audio into filter banks, applies selective gain to shape the frequency bins, and sums all the banks back together to generate the output audio.

4 Kernels

4.1 Least Squares Solver

In Sec. 3.1, we already presented one variant of the least-squares (LS) solver kernel (optical flow). Now, we discuss the kernel in more detail, and we present two other variations.

In any least-squares problem, matrices A and B store data about the system (e.g. in optical flow, the spatial and temporal derivatives). Our goal is to estimate the transformation matrix, X, that maps A to B. This relationship is describe in Eq. 4. Many problems in addition to the ones we present are described in this way.

$$AX = B \tag{4}$$

We can calculate X from the following linear least squares problem:

$$\min_X \frac{1}{2}\|AX - B\|_F^2, \tag{5}$$

where $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$, $B \in \mathbb{R}^{m\times p}$, $X \in \mathbb{R}^{n\times p}$ is the transformation matrix, and $\|\cdot\|_F$ is the Frobenius norm.

This system can be solved by the following iterative process:

$$\begin{aligned} X_{k+1} &= X_k + \alpha(-A^\top A X_k + A^\top B) \\ &= (I - \alpha A^\top A)X_k + \alpha A^\top B \\ \text{where } X_0 &:= \alpha A^\top B, \end{aligned} \tag{6}$$

and α is a positive step length. See [18] for a proof of convergence.

The iterative equation, Eq. 6, can be written in terms of a Hopfield neural network [19] [20]. The matrix $\alpha A^\top$ would form the feedforward path weights of Hopfield neural network and $(I - \alpha A^\top A)$ would form the feedback path weights. We refer the reader to [21] for details on how Eq. 6 is implemented as a Hopfield network via SC.

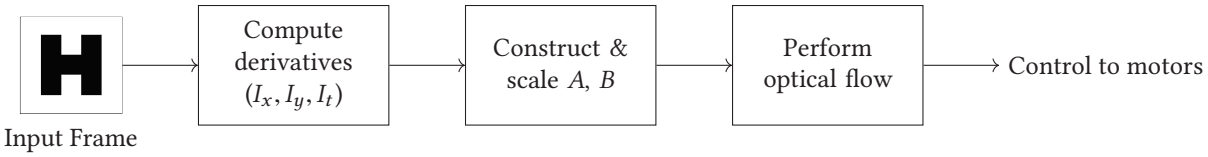

Figure 2. A block diagram of the drone stabilization pipeline. The optical flow block encapsulates the bulk of the compute. A and B are inputs to the optical flow problem as describe Sec. 3.1.

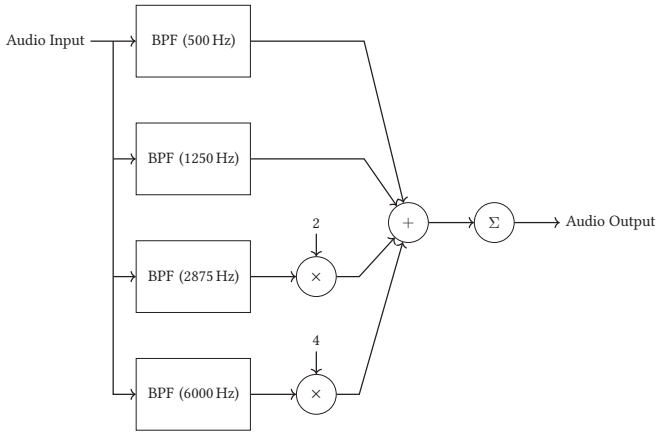

Figure 3. A block diagram of the hearing aid pipeline. Each of the bandpass filters (BPFs) is a kernel in our benchmark suite, and the higher frequencies have larger gain. The Σ indicates a sigma-delta modulator.

Our benchmark suite contains several variations of the least-squares solver described above, with details below.

4.1.1 Object Tracking

In particular, object tracking can be framed as a least squares problem. Consider a set of feature points in an image, $p_1 = (x_1, y_1, 1)$, $p_2 = (x_2, y_2, 1)$, and $p_3 = (x_3, y_3, 1)$ (note the z-dimension is normalized to 1, since we cannot get depth information from an image alone). Similarly, we have another set of paired feature points in a second frame, p_1', p_2', and p_3'. We arrange these features in matrices

$$A = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} x_1' & y_1' & 1 \\ x_2' & y_2' & 1 \\ x_3' & y_3' & 1 \end{bmatrix}$$

We solve for a transformation matrix X such that $AX = B$. X identifies the 2D-affine transformation (translation and rotation) that was applied to point in image A to get image B. By applying this algorithm to subsequent frames of a video stream, we can identify and track a given object. Alg. 1 gives a detailed overview of this process.

4.1.2 Optical Flow

The process of setting up the optical flow problem is described in detail in Sec. 3.1. Alg. 2 provides a step-by-step overview of this process. Yet, in practice, optical flow cannot be applied to entire image at once. A typical approach (which we use) is to divide the image into a grid of "tiles" and process each tile at a time. We do this to provide a more accurate optical flow solution, to reduce the dimension of A and B in the LS problem, and to allow for pipelining. There are other approaches for partitioning the optical flow problem, but we do not cover those here. Instead, we note that these subtleties highlight the potential for future work based on this benchmark suite.

Algorithm 1 Object detection using Harris corner detector for feature extraction

1: Extract coordinates of feature points [22] from initial image into matrix A
2: Extract coordinates of feature points from final image into matrix B
3: Solve for X to to get transformation matrix: $AX = B$

Algorithm 2 Lucas-Kanade Optical Flow (for one tile)

1: Compute x and y spatial derivatives of frame at time t and store them in matrix A
2: Compute temporal derivatives between frame at time t and frame at time $(t + T)$ and store them in matrix B
3: Solve for X to get velocity: $AX = B$

4.1.3 Inverse Kinematics

Inverse kinematics is common in robotics applications where precise control of appendages is required. Consider a robotic arm with two pivot points. The robot is able to estimate the angle at a pivot point of each appendage with respect to a fixed axis. These angles are given as in a vector Θ_k. From the angles, we can calculate x and y coordinates of each pivot point. These are stored in the matrix A. Similarly, we have (x, y) coordinates for the final desired position of the pivot points. We calculate the difference between the current and final coordinates and store them in B. Solving $AX = B$ provides the transformation, X, that will take the current coordinates and transform them into the final coordinates. We update our angles according to $\Theta_{k+1} = \Theta_k + \gamma X$ where γ is a rate of convergence. Smaller γ allows from more precise movement, but it will take the robot longer to reach the

final arm position. Once Θ_{k+1} is calculated, the robot can drive the motors at each pivot point to achieve the required change in angle. Alg. 3 provides step-by-step explanation of this process.

Algorithm 3 Inverse Kinematics

Require: Learning rate, γ
1: **for** $k = 1, 2, \ldots$ **do**
2: Store current value of angles of the arm w.r.t x-axis in matrix Θ_k
3: Compute current x and y coordinates of arm positions and store in matrix A
4: Compute the difference between the final and current positions and store in matrix B
5: Solve for X to to get transformation matrix: $AX = B$
6: Update: $\Theta_{k+1} = \Theta_k - \gamma X$
7: **end for**

4.1.4 Variations of LS

The algorithms presented in Sec. 4.1.1, 4.1.2, and 4.1.3 all use a least-squares problem at their core. This is why we focus on LS as a kernel in our suite, because many applications associated with PAVs can be formulated as LS problems. Table 1 lists the different input matrix sizes for LS variants in Sec. . All the variants are solved using the same iterative process described in Eq. 6.

Table 1. Input matrix sizes for variations of least-square solver.

Variant	Size of A	Size of B
Object Tracking	3×3	3×3
Optical Flow	4×2	4×1
Inverse Kinematics	2×2	2×1

4.2 Singular Value Decomposition

In order to navigate in an open space, a robot can make use of computer vision algorithms known as homography estimation and decomposition. These techniques are commonly used in vision-based control systems [23] [24]. The algorithms utilize basic matrix operations as well as a linear solver (i.e. least squares minimization) and singular value decomposition (SVD). The former operations are already described in Sec. 4.1.

The SVD of a $m \times n$ matrix, A, is defined as

$$A = U \Sigma V^{\top}$$

where $U \in \mathbb{R}^{m \times r}$, $V \in \mathbb{R}^{n \times r}$, and $\Sigma \in \mathbb{R}^{r \times r}$. Here $r = \text{rank}(A)$. U and V's columns are orthogonal unit vectors, and Σ is a diagonal matrix. The columns of U and V are referred to as the left and right singular vectors, respectively, while the elements of the diagonal of Σ are the singular values.

4.2.1 Homography Estimation and Decomposition

A robot with a visual stimulus can travel between two waypoints by observing its surroundings. For a given landmark, the robot navigates between two different points using stored projections of the 3D landmark. Extracting the translation vector from these two projections is known as homography estimation and decomposition. Alg. 4 provides a high level overview of this process. For brevity, we omit discussion of the theory behind homography decomposition. Instead, we simply note that many PAV applications involve finding a transformation between data acquired at two different time steps. The SVD is a useful tool for decomposing this transformation into meaningful components. Additionally, as noted in Step 3 of Alg. 4, certain LS problems (e.g. $AX = \vec{0}$) cannot easily be solved with a traditional solver. Instead, an SVD can be used to find a solution quickly.

Algorithm 4 Homography Estimation and Decomposition Overview (citations per step)

Require: Extracted pairs of feature points
1: Calculate normalization matrix, T [25]
2: Normalize feature points: $x' = Tx$
3: Find homography by solving linear system of equations [26] [27] (can be done using SVD)
4: Reverse normalization: $H = T_B^{-1} H' T_A$
5: Decompose homography by taking SVD of H [23] [24]

4.2.2 Iterative SVD

We will focus on the SVD in Step 5 of Alg. 4. H is a 3×3 matrix, so we can quickly find its SVD by the power iteration method (referred to as the iterative SVD).

Algorithm 5 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: $w_k = A v_{k-1}$
3: $\alpha_k = \|w_k\|_2 = \sqrt{w_k^{\top} w_k}$
4: $u_k = w_k / \alpha_k$
5: $z_k = A^{\top} u_k$
6: $\sigma_k = \|z_k\|_2 = \sqrt{z_k^{\top} z_k}$
7: $v_k = z_k / \sigma_k$
8: **end for**
9: **return** First left/right singular vectors, u_k & v_k, and first singular value, σ_k

Alg. 5 only computes the first left and right singular vectors and first singular value. In order to compute the rest of

the SVD, we remove the first component from by

$$A' = A - \sigma_1 u_1 v_1^\top \tag{7}$$

then apply Alg. 5 to A'. This process is repeated as many times as the rank(A).

4.3 Audio Filters

Four kernels are included to demonstrate the filter applications, two infinite impulse response (IIR) filters and two finite impulse response (FIR) filters.

Both IIR filters are bandpass filters envisioned for separating audio into frequency banks such as in the hearing aid. A more complex application that requires frequency banks is speech recognition. In this scenario, human speech is typically separated into 13 banks. The last band in the bank is typically centered at 8 kHz, so we use this as the center frequency in our kernel experiments.

FIR filters are a more popular class of filters due to their predictable effects on the phase of the output signals. Both FIR filters in the benchmark suite are moving average filters, but any FIR filter can be implemented with the same deterministic bitstream logic.

4.3.1 Digital State Variable Filter

The included digital state variable filter (SVF) [28] is a bandpass filter with a center frequency of $F_c = 8$ kHz. Fig. 4 illustrates the block diagram for this type of filter. Typical PCM data operates at a sampling rate of $F_s = 44.1$ kHz. By Eq. 8, we obtain a filter coefficient of $f = 1.0791$. The coefficient q is chosen according to Eq. 9 for the desired Q-factor.

$$f = 2\sin\left(\frac{\pi F_c}{F_s}\right) \tag{8}$$

$$q = \frac{1}{Q} \tag{9}$$

This traditional implementation results in coefficients that are cumbersome for binary arithmetic. Typically, hardware designs would favor multiplication and division by powers of 2.

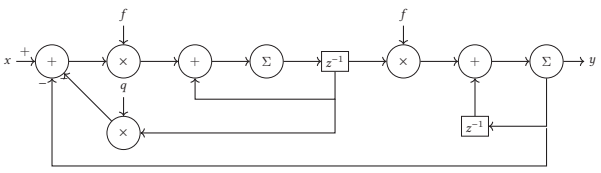

Figure 4. A block diagram of a state-variable filter (SVF). f and q are fixed point constants, and x and y are deterministic bitstreams. Σ indicates a sigma-delta modulator.

By utilizing PDM input, we receive oversampled one-bit data at 3 MHz. Given the oversampled input, we can choose a sampling frequency as some multiple of this base rate, effectively tuning the sampling frequency, F_s. This results in a new design equation described by Eq. 10 where f is chosen ahead of time to be a power of 2, and F_s is determined accordingly [8] [28]. We then downsample to achieve the calculated F_s.

$$F_s = \frac{\pi F_c}{\arcsin(f/2)} \tag{10}$$

For example, in our designs, we choose $f = 0.125 = 2^{-3}$ which results in $F_s = 40.186$ kHz. To achieve this effective sampling frequency, we determine the length of the delay elements in Fig. 4 by Eq. 11.

$$d = \left\lceil \frac{F_{\text{PDM}}}{F_s} \right\rceil \tag{11}$$

For the f and F_s we discuss above, $d = 8$.

We choose q so that it is also a simple combination of powers of 2. For our design, $q = 1.875$. We can also iteratively tune f to minimize delay, since longer delay lengths require more flip-flops. As a result, we get the following final parameters

$$f = 0.0125 \quad q = 1.875 \quad F_s = 3 \times 10^6 \quad d = 1$$

4.3.2 Direct Form I Biquad Filter

The included biquad kernel shown in Fig. 5, is a direct form I biquad filter [29] which is suitable for fixed point applications. The center frequency was also chosen to be 8 kHz — essentially a biquad equivalent of the above SVF implementation. Eq. 12 and 13 contain the necessary coefficient calculations for this filter.

$$\begin{aligned} b_0 &= \alpha & b_1 &= 0 & b_2 &= -\alpha \\ a_0 &= 1 + \alpha & a_1 &= -2\cos(\omega_c) & a_2 &= 1 - \alpha \end{aligned} \tag{12}$$

$$\alpha = \frac{\sin(\omega_c)}{2Q} \quad \omega_c = \frac{2\pi F_c}{F_s} \tag{13}$$

As seen in Eq. 12, most of the constants are simple functions of α, so to get a binary-friendly representation, we carefully choose F_c/F_s in Eq. 13. Note that the length of the delay elements is still adjusted according to Eq. 11. Thus, this system exposes a complex trade-off between arithmetically simple operations and buffer cost (# of FFs). Note that the coefficient a_1 is harder to control, and in most cases, it cannot be made to be a simple binary number.

4.3.3 Moving Average Filter

The moving average filter has two different variants included — a simple 4-tap moving average filter and a more complicated 32-tap moving average filter. The moving average filter takes each output from the taps and averages them. After this operation, their average passes through a sigma-delta modulator that transforms the decimal number back into the bitstream domain. The only coefficient in the moving average filter is $1/n$ where n is the number of taps. If the coefficient of interest is a power of two, the multiplication

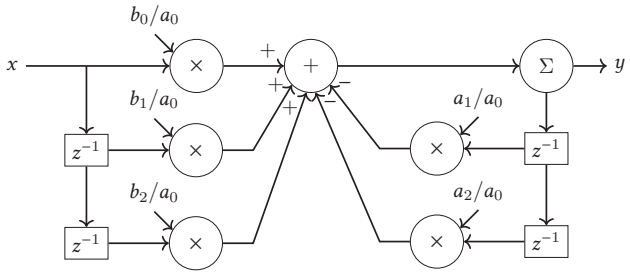

Figure 5. A block diagram of a DF-I biquad filter. a_0, a_1, a_2, b_0, b_1, and b_2 are fixed point constants, and x and y are deterministic bitstreams. Σ indicates a sigma-delta modulator.

is merely a shift operation. Eq. 14 describes this system, and Fig. 6 provides a block diagram.

$$y_k = \frac{1}{n} \sum_{i=k}^{k-(n-1)} x_i \tag{14}$$

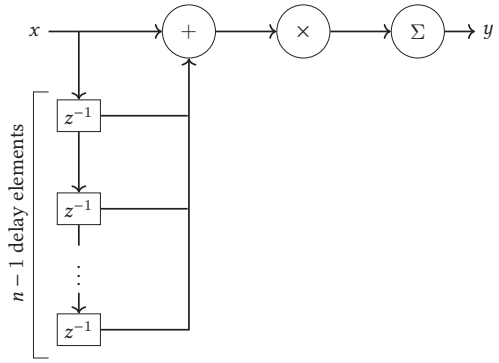

Figure 6. A block diagram of a n-tap moving average (MA) filter. n is a fixed point constant, and x and y are deterministic bitstreams. Σ indicates a sigma-delta modulator.

5 Evaluation & Results

We evaluate each design using Verilog implementations mapped to ultra-low power Lattice FPGAs. Floating point and fixed point baselines are created using Vivado HLS. Below, we will discuss the area, power, and energy results, as well as highlight interesting characteristics of bitstream computing.

5.1 Kernel Area Results

We compare area against a floating point and fixed point baseline when appropriate. As shown in Fig. 7, bitstream implementations of each algorithm consume less area compared to floating point or fixed point implementations. For most FP/FXP implementations, the number of Lattice FPGAs required to partition the design is infeasible (see Table 2); however, all recommended bitstream designs fit on a single Lattice FPGA. By using bitstreams, we are able to take advantage of an FPGA platform well-suited for ultra-low power applications. Still, in Sec. 5.2, we report power and energy numbers for the FP/FXP designs with the large partitioning overhead.

Table 2. Instance count for partitioning FP/FXP designs across Lattice FPGAs. The high instance counts make these designs impractical for mapping across Lattice FPGAs.

Application	FPGA P/N	# of Chips
Iterative SVD (FP)	LM4K	8
Iterative SVD (FXP)	LM4K	15
Linear Solver (OF) (FP)	LM4K	20
Linear Solver (OF) (FXP)	LM4K	14
Linear Solver (IK) (FP)	LM4K	11
Linear Solver (IK) (FXP)	LM4K	3
Linear Solver (OT) (FP)	LM4K	37
Linear Solver (OT) (FXP)	LM4K	7

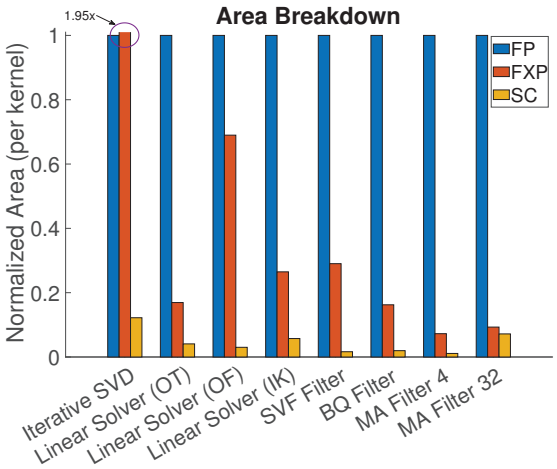

Figure 7. Normalized area consumption relative to floating point implementation. Area is computed as # of LUTs + # of FFs.

5.2 Kernel Power and Energy Results

The energy and power results are shown in Fig. 8. As seen in Fig. 8a, the bitstream designs consume orders of magnitude less power than the FP/FXP alternatives. This gap should be even wider, since we cannot account for the cost of data movement when partitioning across many FPGAs (as is the case for FP/FXP). Fig. 8a also illustrates that dynamic power is a small fraction of the total power in all designs. This is why the implementations enjoy a strong power savings by being mapped to ultra-low power FPGAs which are designed for low leakage current.

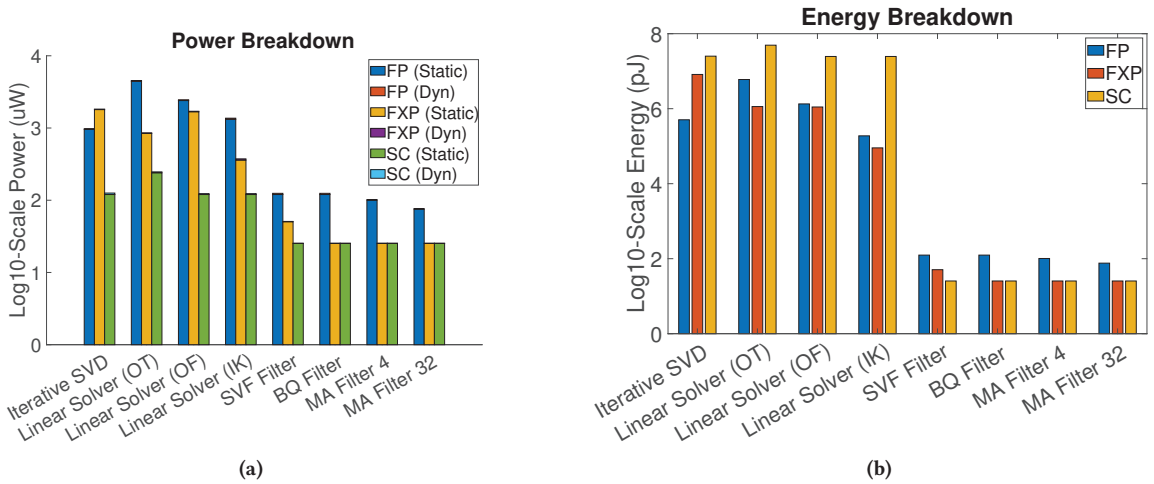

Figure 8. Energy and power consumption for different implementations of the iterative SVD, linear solver, and filters on FPGAs.

Still, the stochastic bitstream applications suffer in Fig. 8b due to a long runtime. There have been proposal for reducing the runtime of stochastic systems [30], and our experiments illustrate that creative latency reduction that does not sacrifice significant accuracy is a fruitful opportunity for designers.

Unlike stochastic bitstreams, deterministic bitstreams do not suffer from a runtime increase, so they enjoy energy savings over the FP/FXP alternatives. Yet, there is still room for improvement. As illustrated in Sec. 4.3, bitstream digital filters expose several tuning knobs to simplify the datapath. Our work only explores some of these possibilities, and we believe there is still much work to be done on the filter kernels.

For reference, our optical flow implementation would consume 375.3 uW of dynamic power and 1.14 uW of static power at a 40 nm technology node (better than the prior work's ASIC [3] [2]); however, we do not suggest using an ASIC for our applications, since an ASIC doesn't allow for flexibility (i.e. it can't be reprogrammed as the PAV's directives change).

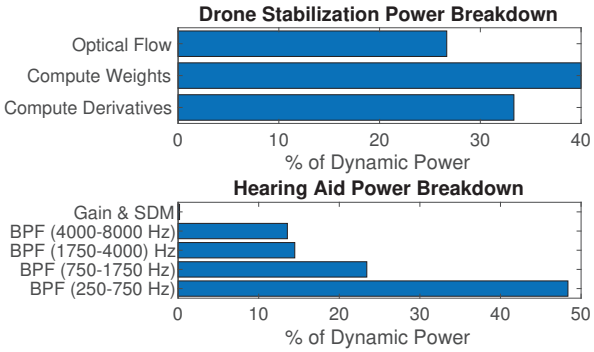

Figure 9. Breakdown of dynamic power in each application. Computing the weights consumes the most area since it involves the largest matrix multiplies. Filters with lower center frequencies consume more power due to longer delay buffer lengths.

5.3 End-to-End Application Analysis

Using the kernels, we can implement the two applications described in Sec. 3. Fig. 9 illustrates the breakdown of dynamic power by application. As seen in the drone stabilization subplot, the linear solver and compute weight blocks occupy a significant percentage of the power consumption. These are the blocks that make up the linear solver (OF) kernel. In the case of the hearing aid, we see that filters with lower center frequencies consume significantly more power since the delay elements are longer. This power breakdown is not fixed; it is dependent on the many trade-off tuning knobs associated with the kernels.

5.4 Fault Tolerance

We also evaluated the fault tolerance of bitstream computing systems in the case of the iterative SVD. Bitstreams are naturally fault tolerant to bit flips, because a single bit does not carry any more significance than other bits. On the other hand, if a bit in the exponent of a floating point number flips, then the error increases exponentially. Fig. 10 illustrates this behavior. To generate the plot, we randomly flipped bits in the input bitstream according to the specified error rate. Similarly, we flipped bits in the floating point input at the

same rate. Bits were uniformly distributed across all bit positions. The loss in the plot is the application loss for the iterative SVD (i.e. $\|Av - \sigma u\|$ which should be zero for a correct solution). Extremely low bit flip rates (e.g. 1% of bits) results in order of magnitude larger error for floating point designs. Beyond this low flip rate, the floating point design performs so poorly that it would not be acceptable for the PAV.

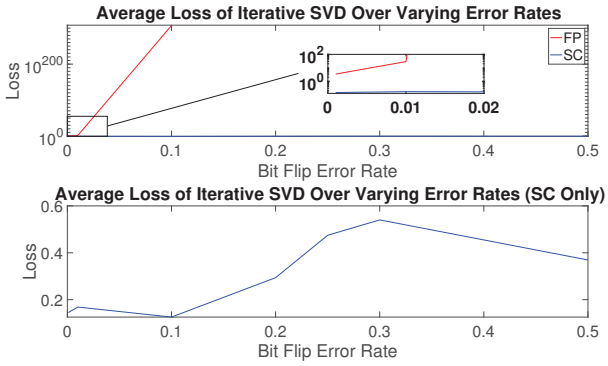

Figure 10. Fault tolerance of the iterative SVD against random bit flips in floating point and stochastic bitstreams. As seen in top subplot, the error rate of the FP implementation quickly increases, but the SC implementation (zoomed in the lower subplot) remains relatively stable.

5.5 Signal-to-Noise Ratio

We evaluated the signal to noise ratio (SNR) of the state variable filter to demonstrate the advantage of intelligently chosen coefficients. In Fig. 11, the SNR of the smart coefficients (by modifying sampling rate according to Eq. 10) remains constant varying fractional width. The naive case, where the coefficients are simply truncated to the desired precision, has worse SNR as the fractional width decreases. The ability to choose our coefficients intelligently was a result of using an oversampled PDM input. With a PCM input, the sample rate would not be tunable, and the filter performance would need to suffer in order to accommodate a simpler datapath.

6 Conclusions

In this work, we demonstrated that bitstream implementations of least squares minimization and iterative SVD are more efficient than their floating point and fixed point counterparts. Furthermore, these algorithms are central to several robotic applications where streaming data is natural. Additionally, we demonstrated how filters on oversampled bitstreams can be designed with efficient low precision coefficients without sacrificing SNR performance. Again, typical audio data begins as an oversampled bitstream, making such filters attractive. We presented our kernels in a packaged benchmark suite — BitBench — which we intend to release alongside this paper, including a complete source-to-HDL automated toolchain. We demonstrate some of the interesting trade-offs and behavior in the benchmark such as fault tolerance and SNR robustness. We hope that the release of this suite will encourage further embedded systems research that explores the opportunities available in the bitstream computing paradigm.

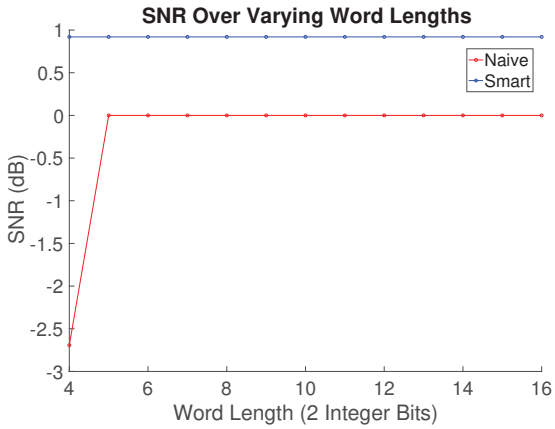

Figure 11. SNR in dB graphed against the # of bits of precision. The smart case modifies the sampling rate to achieve the desired bit precision. The naive case simply truncates the coefficients.

This work was funded in part by NSF awards CCF-1628384 and CCF-1813434 and AFRL award FA9550-18-1-0166.

A Artifact Appendix

A.1 Abstract

Our artifacts contain the necessary Verilog, TCL scripts, and shell scripts to synthesize the results in the paper. We have uploaded all these files as a compressed zip file to Zenodo (DOI: 10.5281/zenodo.2648959). Reviewers can download the source from that permalink which contains a README.md file explaining the procedure to run the experiments and verify the results.

A.2 Artifact Checklist (Meta-Information)

- **Program:** Xilinx Vivado 2018.2 and Matlab R2017b
- **Compilation:** Vivado synthesis tools
- **Output:** CSV data and plots
- **How much disk space required (approximately)?:** 1.8GB
- **How much time is needed to prepare workflow (approximately)?:** None
- **How much time is needed to complete experiments (approximately)?:** 6 hours
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.2648959

A.3 Description

A.3.1 How Delivered

Our artifact is available on Zenodo (10.5281/zenodo.2648959).

A.3.2 Software Dependencies

Ubuntu 16.04 LTS, Bash, Xilinx Vivado 2018.2 and Matlab R2017b.

A.4 Installation

Download the zip archive from Zenodo and uncompress.

A.5 Experiment Workflow

We provide two scripts, run_all.sh and run_hls.sh, to synthesize the SC results and baselines. We also provide Matlab scripts to plot the generated data.

To run the HLS synthesis:

```
$ cd hls_runs
$ ./run_hls.sh
```

To run the SC synthesis:

```
$ ./run_all.sh
```

Detailed instructions in the README.md file in the zip archive.

A.6 Evaluation and Expected Results

Running the provided scripts will generate utilization and power data which can be verified against the provided CSV (details in README.md). The Matlab scripts can be used to reproduce the plots.

References

[1] D. Floreano, J.-C. Zufferey, M.V. Srinivasan, and C. Ellington. Flying insects and robots. *Springer-Verlag Berlin Heidelberg*, 1:316, 2009.

[2] Dario Floreano and Robert J. Wood. Science, technology and the future of small autonomous drones. *Nature*, 521(7553):460–466, 2015.

[3] Pierre Emile Duhamel, Judson Porter, Benjamin Finio, Geoffrey Barrows, David Brooks, Gu Yeon Wei, and Robert Wood. Hardware in the loop for optical flow sensing in a robotic bee. *IEEE International Conference on Intelligent Robots and Systems*, pages 1099–1106, 2011.

[4] 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. A Fully Integrated Battery-Powered System-on-Chip in 40-nm CMOS for Closed-Loop Control of. *IEEE Journal of Solid-State Circuits*, 52(9):2374–2387, 2017.

[5] Taylor S. Clawson, Silvia Ferrari, Sawyer B Fuller, and Robert J Wood. Spiking Neural Network (SNN) Control of a Flapping Insect-scale Robot. In *Conference on Decision and Control, IEEE*, number 55, pages 3381–3388, 2016.

[6] Taylor S. Clawson, Terrence Stewart, Chris Eliasmith, and Silvia Ferrari. An adaptive spiking neural controller for flapping insect-scale robots. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7, 11 2017.

[7] Xuan Zhang, Mario Lok, Tao Tong, Simon Chaput, Sae Kyu Lee, Brandon Reagen, Hyunkwang Lee, David Brooks, and Gu-yeon Wei. A Multi-Chip System Optimized for Insect-Scale Flapping-Wing Robots.

[8] Mikko Lipasti and Carly Schulz. End-to-End Stochastic Computing, Feb 2017.

[9] B. R. Gaines. Stochastic computing. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 149–156, New York, NY, USA, 1967. ACM.

[10] G. Orchard and R. Etienne-Cummings. Bioinspired visual motion estimation. *Proceedings of the IEEE*, 102(10):1520–1536, Oct 2014.

[11] Steve K. Esser, Alexander Andreopoulos, Rathinakumar Appuswamy, Pallab Datta, Davis Barch, Arnon Amir, John Arthur, Andrew Cassidy, Myron Flickner, Paul Merolla, Shyamal Chandra, Nicola Basilico, Stefano Carpin, Tom Zimmerman, Frank Zee, Rodrigo Alvarez-Icaza, Jeffrey A. Kusnitz, Theodore M. Wong, William P. Risk, Emmett McQuinn, Tapan K. Nayak, Raghavendra Singh, and Dharmendra S. Modha. Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, Aug 2013.

[12] S. Menon, S. Fok, A. Neckar, O. Khatib, and K. Boahen. Controlling articulated robots in task-space with spiking silicon neurons. In *5th IEEE RAS/EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 181–186, Aug 2014.

[13] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[14] Armin Alaghi and John P. Hayes. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):92:1–92:19, May 2013.

[15] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[16] Byrne D. and Dillon H. The national acoustic laboratories' (NAL) new procedure for selecting the gain and frequency response of a hearing aid. *Ear and Hearing*, 7(4):257–265, 1986.

[17] Ritwik Dhawan and P. Mahalakshmi. Digital filtering in hearing aid system for the hearing impaired, 2016.

[18] Adi Ben-Israel and A. Charnes. Contributions to the theory of generalized inverses. *J. Soc. Indust. Appl. Math.*, 11(3):55–60, 1963.

[19] G.G. Lendaris, K. Mathia, and R. Saeks. Linear hopfield networks and constrained optimization. *IEEE Trans. Systems, Man and Cybernetics, Part B*, 91:114–118, Feb 1999.

[20] R. Shukla, E. Jorgensen, and M. Lipasti. Evaluating hopfield-network-based linear solvers for hardware constrained neural substrates. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, May 2017.

[21] Rohit Shukla, Soroosh Khoram, Erik Jorgensen, Jing Li, Mikko Lipasti, and Stephen Wright. Computing generalized matrix inverse on spiking neural substrate. *Frontiers in Neuroscience*, 12:115, 2018.

[22] Dermot Kerr, Martin Mcginnity, Sonya Coleman, Qingxiang Wu, and Marine Clogenson. Spiking Hierarchical Neural Network for Corner Detection. *Proceedings of the International Conference on Neural Computation Theory and Applications*, pages 230–235, 2011.

[23] Ezio Malis and Manuel Vargas. Deeper understanding of the homography decomposition for vision-based control. *Sophia*, 6303(6303):90, 2007.

[24] Olivier D. Faugeras and Francis Lustman. Motion and Structure From Motion in a Piecewise Planar Environment. *International Journal of Pattern Recognition and Artificial Intelligence*, 02(03):485–508, 1988.

[25] Richard I. Hartley. In defense of the eight-point algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(6):580–593, jun 1997.

[26] Robert Collins. Lecture 16: Planar Homographies.

[27] Elan Dubrofsky. *Homography Estimation*. PhD thesis, Carleton University, 2009.

[28] Nigel Redmond. The digital state variable filter, 2003.

[29] Robert Bristow-Johnson. Cookbook formulae for audio EQ biquad filter coefficients.

[30] Lifeng Miao and Chaitali Chakrabarti. A parallel stochastic computing system with improved accuracy. In *SiPS 2013 Proceedings*, pages 195–200. IEEE, oct 2013.

[31] Ao Ren, Ji Li, Zhe Li, Caiwen Ding, Xuehai Qian, Qinru Qiu, Bo Yuan, and Yanzhi Wang. SC-DCNN: highly-scalable deep convolutional neural network using stochastic computing. *CoRR*, abs/1611.05939, 2016.

[32] Sean C. Smithson, Kaushik Boga, Arash Ardakani, Brett H. Meyer, and Warren J. Gross. Stochastic computing can improve upon digital spiking neural networks. In *IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation*, pages 309–314, 2016.

[33] Kevin Y. Ma, Pakpong Chiraratttananon, Sawyer B. Fuller, and Robert J. Wood. Controlled Flight of a Biologically Inspired, Insect-Scale Robot. *Science*, (May):603–607, 2013.

[34] Kevin Y. Ma. Robobee, 2015.

[35] Zhenhua Yang, Lixin Tang, and Lingsong He. A New Analytical Method for Relative Camera Pose Estimation Using Unknown Coplanar Points. *Journal of Mathematical Imaging and Vision*, 60(1):33–49, 2018.

[36] A. H. Bentbib and A. Kanber. Block power method for SVD decomposition. *Analele Stiintifice ale Universitatii Ovidius Constanta, Seria Matematica*, 23(2):45–58, 2015.

[37] N. Saraf, K. Bazargan, D. J. Lilja, and M. D. Riedel. Iir filters using stochastic arithmetic. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[38] Bingzhe Li, M. Hassan Najafi, and David J. Lilja. Using stochastic computing to reduce the hardware requirements for a restricted boltzmann machine classifier. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 36–41, New York, NY, USA, 2016. ACM.

[39] Y. Zhang, R. Wang, X. Jiang, Z. Lin, S. Guo, Z. Zhang, Z. Zhang, and R. Huang. Design guidelines of stochastic computing based on finfet: A technology-circuit perspective. In *2017 International Electron Devices Meeting*, IEDM '17, Dec 2017.

[40] Y. Lv and J.P. Wang. A single magnetic-tunnel-junction stochastic computing unit. In *2017 International Electron Devices Meeting*, IEDM '17, 2017.

[41] C. S. Thakur, S. Afshar, R. M. Wang, T. J. Hamilton, J. Tapson, and A. van Schaik. Bayesian estimation and inference using stochastic electronics. *Frontiers in Neuroscience*, 10:104, 2016.

[42] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, and et. al. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, Aug 2013.

[43] IBM Neurosynaptic System Neuron Function Library Reference Manual. Technical report, IBM Corporation, 2016.

[44] B. Moons and M. Verhelst. Energy-efficiency and accuracy of stochastic computing circuits in emerging technologies. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 4(4):475–486, Dec 2014.

[45] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[46] J. Smith. Space-time algebra: A model for neocortical computation. In *2018 International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, USA, June 2018.

[47] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[48] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[49] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.

[50] Xilinx. Vivado design suite user guide-high level synthesis, May 2014.

[51] Sharif University. FPGA logic cells comparison, April 2014.

[52] Lattice semiconductors. DS1048 - iCE40 ultra family data sheet, June 2016.

[53] Wassily Hoeffding. *Probability Inequalities for sums of Bounded Random Variables*, pages 409–426. Springer, 1994.

[54] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. D. Riedel. Computation on stochastic bit streams digital image processing case studies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(3):449–462, March 2014.

[55] Hyeonuk Sim and Jongeun Lee. A new stochastic computing multiplier with application to deep convolutional neural networks. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 29:1–29:6, New York, NY, USA, 2017. ACM.

[56] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, 60(1):93–105, Jan 2011.

[57] M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. H. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January 2018.

[58] H. Kim, J. Yu, and K. Choi. Hybrid spiking-stochastic deep neural network. In *2017 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, April 2017.

[59] A. Nere, A. Hashmi, M. Lipasti, and G. Tononi. Bridging the semantic gap: Emulating biological neuronal behaviors with simple digital neurons. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 472–483, Feb 2013.

[60] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, May 2014.

[61] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[62] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. Buzz: An Extensible Programming Language for Self-Organizing Heterogeneous Robot Swarms. pages 1–12, 2015.

[63] Tiziano Zito. Modular toolkit for Data Processing (MDP): a Python data processing framework. *Frontiers in Neuroinformatics*, 2(January):1–7, 2008.

[64] Devon Jenson and Marc Riedel. A deterministic approach to stochastic computation. *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD '16*, pages 1–8, 2016.

[65] Pai Shun Ting and John Patrick Hayes. Stochastic logic realization of matrix operations. *Proceedings - 2014 17th Euromicro Conference on Digital System Design, DSD 2014*, pages 356–364, 2014.