Scaled Population Arithmetic for Efficient Stochastic Computing

He Zhou, Sunil P Khatri, Jiang Hu and Frank Liu

Abstract— We propose a new Scaled Population (SP) based arithmetic computation approach that achieves considerable improvements over existing stochastic computing (SC) techniques. First, SP arithmetic introduces scaling operations that significantly reduce the numerical errors as compared to SC. Experiments show accuracy improvements of a single multiplication and addition operation by $6.3\times$ and $4.0\times$, respectively. Secondly, SP arithmetic erases the inherent serialization associated with stochastic computing, thereby significantly improves the computational delays. We design each of the operations of SP arithmetic to take $\mathcal{O}(1)$ gate delays, and eliminate the need of serially iterating over the bits of the population vector. Our SP approach improves the area, delay and power compared with conventional stochastic computing on an FPGA-based implementation. We also apply our SP scheme on a handwritten digit recognition application (MNIST), improving the recognition accuracy by 32.79% compared to SC.

I. Introduction

Approximate computation is an approach with an emphasis on area and power efficiency, while sacrificing accuracy. For certain classes of applications that are tolerant to computational errors, approximate computation can achieve better area and power characteristics compared with exact arithmetic. Hence, it has shown promising application in scientific computing [1], machine learning [2], signal processing [3], and real-time systems [4].

Popular techniques for approximation computing include the following: precision scaling [5], inexact or faulty hardware [6], voltage over-scaling [7], and skipping tasks and memory accesses [8]. Among these techniques, stochastic computing [9] is a non-conventional arithmetic scheme for area-efficient implementation of error-tolerant applications. Stochastic computing has received renewed interest due to, among other reasons, the degrading reliability of recent VLSI fabrication processes, its purported decrease in power and energy, and its robustness to bit-flip errors. In stochastic computation, values are represented by binary bit streams, and the arithmetic operations can be processed by simple logic circuits, such as OR/AND gates for addition and multiplication, respectively.

However, classical stochastic computing, which is abbreviated as SC in this sequal, has its own limitations. First, although it was claimed that SC has a high error tolerance to bit flips [10], its accuracy depends heavily on the *density* and the randomness of the 1's in the binary bit-stream [11]. To the best of our knowledge, the error of SC have not been quantified to date. This paper presents an error analysis for the proposed scaled population arithmetic as well as SC. Second, since SC uses a population-based representation alone for all numbers, it can only represent numbers in $[0, 1]$. The limitation can be problematic when overflow occurs in the operations, especially in addition. The third limitation of SC is the runtime complexity. Although the arithmetic operation units consist of only OR/AND logic gates, the supporting units, e.g., the random number generator (RNG) and the shuffler, have a runtime complexity of $\mathcal{O}(k)$, where k is the number of bits in SC representation. These weaknesses limit the applicability of SC in real world applications.

In order to alleviate the above limitations, we propose a new Scaled Population (SP) arithmetic based computation which achieves fast, approximate computation with a low area/power overhead and improved accuracy. SP arithmetic uses some of the basic ideas of SC, but with three key enhancements: a) the inherent serialization in SC is avoided; b) the errors of SC are significantly reduced by providing a scaling (exponent) term in SP arithmetic; and c) the range of numbers that can be represented by SP is much larger than what is possible in SC. The key design goal of SP arithmetic is that each operation be computed using $\mathcal{O}(1)$ gate delays (as opposed to clock cycles). Unlike SC, SP never allows any operation which requires a serial traversal of the bits of the operand. The SP arithmetic achieves a dramatic speedup over SC.

Our proposed SP computation greatly improves the accuracy of a single multiplication and addition operation by $6.3\times$ and $4.0\times$, compared with SC. Our experimental results show that for addition and multiplication, our SP approach uses $7.13\times$ and $3.75\times$ fewer LUTs than conventional floating point number based arithmetic circuit, respectively. We also test our approach in the scenarios of matrix inner product and image classification using MNIST dataset. Our approach achieved a 32.79% improvement over SC in terms of the accuracy.

The key contributions of this paper are:

- Introduction of SP, with larger range, better error and reduced delay than SC.
- Achieving constant delay for all operations, and design for speed and accuracy over SC.
- Quantifying the errors of SP arithmetic and SC.
- Applying the SP approach on simple addition/multiplication, matrix inner product and MNIST classification.

II. Stochastic Computing and Previous Works

Stochastic computing is an approximate arithmetic approach that allows area-efficient circuit implementation for some operations on fractional numbers. Consider a fractional number $P_x \in [0, 1)$. In conventional binary number representation, it is represented as $X = x_1x_2...x_k$ such that $P_x = \sum_{i=1}^{k} 2^{-i}x$. In stochastic computing, by contrast, it is represented by a $\Pi-$bit vector π, where $|\pi| \leq \Pi$ bits are randomly chosen to be 1, so that $P_x = \frac{|\pi|}{\Pi} \in [0, 1]$.

In [9], [12], the key elements of stochastic computing, including circuit implementations and a comparison with analog computing, are introduced. One prominent benefit of stochastic computing is the very low area cost in implementing certain arithmetic operations. Fig. 1 and Fig. 2 show examples

H. Zhou, J. Hu and S. P. Khatri are with the Department of Electrical and Computer Engineering, Texas A&M University.

F. Liu is with the the IBM Austin Research Laboratory.

of multiplication and addition operations, respectively. The area advantage is clearly evident. The computing results depend on the number and the locations of 1's in the bit-streams, and therefore are usually inaccurate. Moreover, the 1's are required to be randomly located in each bit-stream.

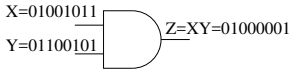


Fig. 1. Multiplication: $\frac{4}{8} \times \frac{4}{8} = \frac{2}{8}$

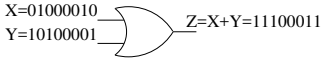


Fig. 2. Addition: $\frac{2}{8} + \frac{3}{8} = \frac{5}{8}$

The work of [13] shows how to realize subtraction in stochastic computing by using a multiplexer (MUX) and a NOT gate. In [12], addition and subtraction approaches are introduced to solve the overflow problem. Although this approach solved the problem of overflow, it is a serial process, i.e., only one bit in π is processed at a time. Hence it can easily form a performance bottleneck. A stochastic division circuit design, called CORDIV, is proposed in [14].

In the basic form of stochastic computing, only numbers in $[0, 1]$ are allowed. In [11], multiple representation schemes are reported to overcome this limitation. The bipolar format increases the operands' range to [-1,1] [15]. In [16], the numerical value of a bit-stream representation is no longer population-based, but interpreted as the ratio of 1's to 0's, which increases the range to $[0, +\infty]$. Although these approaches increase the range of the operands, they require a more complicated design for the arithmetic operation circuits, while our SP arithmetic achieves a large range for number representation while ensuring that all operations incur only $\mathcal{O}(1)$ gate delays.

Since the accuracy of stochastic computation relies highly on the randomness of the 1's in the bit-stream, data shuffling has been used in SC [17], through random number generators. However, these approaches either introduce more overhead with respect to runtime, area and power, or are not able to introduce enough randomness. By contrast, our SP approach makes use of several multi-level Linear-Feedback Shift Register (LFSR) based shuffler, which improves both efficiency and randomness.

In spite of the considerable studies on stochastic computing, the research attention on quantifying its accuracy and error characteristics has been surprisingly light. In particular, there is a lack of a systematic investigation on the errors due to the densities of 1's in the bit-stream of the SC number representation. A key contribution of our SP arithmetic is to fill this void and remarkably improve the accuracy over SC.

III. Scaled Population Arithmetic

A. Number Representation

The number representation in Scaled Population (SP) arithmetic is an enhancement of that in SC, with a scaling (exponent) term such that it can cover a range beyond $[0, 1]$. Specifically, the SP representation of a number x is an M-bit tuple $x = \{\sigma, \pi\}$, where σ is a Σ-bit scaling term and π is a Π-bit population vector such that $M = \Sigma + \Pi$. The numerical value of x is $\frac{|\pi|}{\Pi} \times 2^{(\sigma - \Sigma_0)} \simeq x$, where $|\pi|$ is the number of 1's in the population vector π, and Σ_0 is a constant, typically chosen to be $2^{(\Sigma-1)}$. The reason that we include the Σ_0 constant is to allow the value of the scaling term in the SP representation (i.e. $2^{(\sigma-\Sigma_0)}$) to be smaller than 1, so that we can have increased resolution, allowing us to increase the density of the population vector without changing the numerical value of x. We note that the 1's in the population vector π are uniformly distributed, which has the similar characteristic with SC [12]. The SP representation described above only handles positive numbers. However, augmenting SP to handle signed computation can be easily accomplished by adding a sign bit.

For example, if $\{\sigma, \pi\} = \{110, 1011010101\}$ then $|\pi| = 6$, $\Pi = 10$, $\Sigma = 3$ and $\sigma = 6$. Hence the numerical value of the SP number x is $\frac{6}{10} \times 2^2$, which equals 2.4. Note that the inclusion of the scaling term is something that SC does not have. The SP number representation not only covers a much larger range of numbers, but also, and more importantly, facilitates arithmetic operations that have improved computing accuracy, as will be elaborated as follows.

B. Arithmetic and Supporting Operations

In this section, we will describe two most commonly used arithmetic operations, multiplication and addition, followed by a description to supporting operations such as shuffling, density checking and scaling.

Fig. 3 is a top level block diagram of the proposed SP arithmetic system. The input operands are represented as conventional binary numbers X and Y. The generators convert X and Y to the SP format, e.g., $x = \{\sigma, \pi\}$. Then, the operands x and y in SP format are fed into the arithmetic processing units, such as adder and multiplier, for computation.

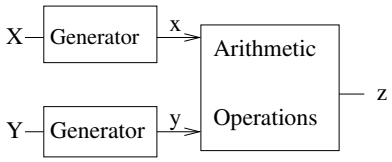


Fig. 3. The top level view of SP scheme.

In designing SP-based arithmetic circuits, we ensure that each operation incurs $\mathcal{O}(1)$ gate delays. In particular, any computation that requires us to iterate over the M bits of an SP vector (which requires serialization) is avoided. Note that all the operations described below are approximate in nature. Also, the computations on the scaling term are efficient, since Σ is a very small value.

1) Multiplication: Multiplication uses an AND gate as the underlying function, as in SC. In [12], it was proved that an AND gate is able to achieve multiplication when the operands are presented in population-based vectors. In SP arithmetic, we propose a scaling operation, to be performed prior to each multiplication in order to improve the computation accuracy. This improvement is based on the observation that the multiplication accuracy is higher when there are more 1's in the population vectors of the operands.

Lemma III.1. *Consider multiplication between x and y, with the result being z. The computational error ε from the AND gate-based multiplication decreases when $\frac{|\pi_x|}{\Pi}$ or $\frac{|\pi_y|}{\Pi}$ increase.*

Proof. Let $p_x = \frac{|\pi_x|}{\Pi}$, $p_y = \frac{|\pi_y|}{\Pi}$ and $p_z = \frac{|\pi_z|}{\Pi}$. Then each bit in π_x, π_y and π_z is 1 with a probably of p_x, p_y and p_z, respectively. For the i^{th} bit, ideally $p_z = p_x \times p_y$. Hence, the

error of the i^{th} bit in π_z occurs when the probability of it being 1 is not $p_x \times p_y$, i.e., the error at the i^{th} bit in π_z is $\varepsilon_i = (1 - p_x \times p_y)$. When p_x or p_y increases, ε_i decreases. Therefore, considering the entire population vector, when p_x or p_y increases, ε will decrease as well. □

Based on Lemma III.1, the average error of multiplication with p_x and p_y varying over the interval $[0, 1]$ is $\int_0^1 \int_0^1 (1 - p_x p_y) dp_x dp_y = 0.75$.

The scaling term in the SP number representation allows us to control the density of population vectors by scaling. A *density checker unit* and a *scaling unit* are needed together to perform the density control. Also, the randomness of the distribution of 1's in the population vectors affects the accuracy as well. The more uniformly randomly the 1's are distributed, the more accurate the result is. Therefore, a *shuffle unit* is additionally needed in our design for achieving randomness. The key elements for the SP multiplication are shown in Fig. 4.

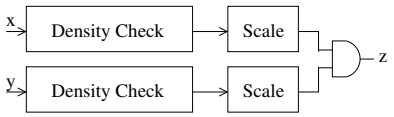


Fig. 4. The SP-based Multiplication

Consider multiplication between x and y. We first check if the population vectors of the two numbers are dense enough, i.e. $|\pi_x| \geq T_1$ and $|\pi_y| \geq T_1$, where T_1 is a sufficiently high fraction. Typically, $T_1 \sim 0.7 \times \Pi$ is a good value according to our experimental results. Such density checking is performed by the density checker unit. If not dense enough, the population vectors are scaled to make them dense enough, and corresponding changes are made to the scaling terms. Now we compute $\mathsf{z} = \mathsf{x} \times \mathsf{y}$ as $\pi_z = \pi_x \& \pi_y$, and $\sigma_z = \sigma_x + \sigma_y - \Sigma_0$.

2) Addition: In our approach, addition is approximately achieved by using an OR gate. Suppose we want to add x and y. A high accuracy requires that the population vectors of both numbers have a density lower than a threshold T_2.

Lemma III.2. *Suppose we want to add x and y, with the result being z. The error ε of the OR-based addition decreases when $\frac{|\pi_x|}{\Pi}$ or $\frac{|\pi_y|}{\Pi}$ decreases.*

Proof. Let $p_x = \frac{|\pi_x|}{\Pi}$, $p_y = \frac{|\pi_y|}{\Pi}$ and $p_z = \frac{|\pi_z|}{\Pi}$. Then, each bit in π_x, π_y and π_z is 1 with a probably of p_x, p_y and p_z, respectively. The OR operation performed on π_x and π_y leads to $p_z = p_x + p_y - (p_x \times p_y)$. For the i^{th} bit in π_x and π_y, the error at the i^{th} bit in π_z is $\varepsilon_i = p_x \times p_y$. In other words, the error of the i^{th} bit in π_z occurs when the i^{th} bits in π_x and π_y are both 1's. From the above equation, when p_x or p_y decrease, ε_i decreases. Thus, considering the entire population vector, when p_x or p_y decreases, ε will decrease. □

According to Lemma III.2, the average error of addition with p_x and p_y varying over the interval $[0, 1]$ is $\int_0^1 \int_0^1 (p_x p_y) dp_x dp_y = 0.25$.

However, unlike multiplication, as long as the 1's in the population are uniformly distributed, we don't have to use a density check or scaling unit for addition. Instead, we perform *skewed* addition, where each operand occupies the different halves of the π bits. Therefore, in *skewed* addition, it is guaranteed that no matter what the density of the operands' population vectors is, the 1's in the two operands will never be aligned at the same bit position. To perform the skew, we use 2 Π-bit masks m_x and m_y, where m_x has the left $\frac{\Pi}{2}$ bits set to 1, and m_y has the right $\frac{\Pi}{2}$ bits set to 1. The result will be $\pi_z = (\pi_x \& m_x) | (\pi_y \& m_y)$, with $\sigma_z = \sigma_x + 1 = \sigma_y + 1$. If $\sigma_x \neq \sigma_y$, then the scaling unit will be used to adjust the density of the population vectors of x and y. Fig. 5 shows an example of how *skewed* addition is processed. In Fig. 5, $x = \{01, 00111001\}$ and $y = \{01, 10000010\}$. The numerical values of x and y are $2^{(1-2^{(2-1)})} \times \frac{4}{8} = 0.25$ and $2^{(1-2^{(2-1)})} \times \frac{2}{8} = 0.125$, respectively, assuming $\Sigma_0 = 2$. After the $skewed$ addition, the numerical value of the result $z = \{01, 00110010\}$ is $2^{(2-2^{(2-1)})} \times \frac{3}{8} = 0.375$, which matches the theoretical addition.

Note that $skewed$ addition relies on the randomness of the distribution of 1's in the population vector. If the 1's in the population vector are not uniformly distributed, $\pi_x \& m_x$ or $\pi_y \& m_y$ will not have half of the 1's in π_x or π_y approximately, which will introduce an error into the final addition result.

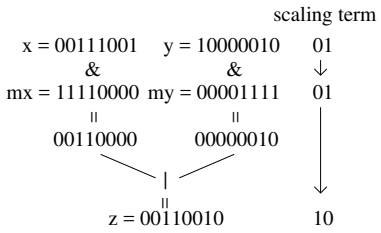


Fig. 5. SP-based *Skewed* Addition

3) Generator: The generate operation converts a conventional binary number to the SP format. The generators used in SC have a computational complexity that is proportional to the bit stream length [18]. Our approach generates a π population vector with $\mathcal{O}(1)$ gate delays, by replicating the bits of the original number based on their bit position. Assuming the original binary number to be $X = \{x_{n-1}, x_{n-2}, ...x_1, x_0\}$, where n is the number of bits. We convert it to the SP format by producing 2^i copies of bit x_i. The resulting bits are then fed to a shuffle unit, which randomizes the bits of π and yields a SP representation of X. Note that in case this would result in a π vector with $|\pi| > \Pi$, then we appropriately adjust the population vector by decimating or dropping the additional bits. Since we assume the 1's are uniformly distributed after shuffling, dropping the additional bits won't change the numerical value of the population vector.

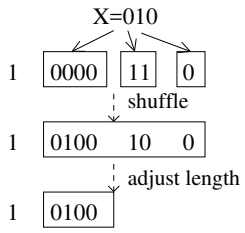


Fig. 6. An Example of Generate Operation, with $X = 0.25$, and $\Pi = 4$

An example of the generate operation is shown in Fig. 6. The numerical value of the binary weighted number $X = 0, 1, 0$ is 0.25. Assume that the length of the population vector Π is 4, $\Sigma = 2$ and $\Sigma_0 = 1$. The initial scaling term is 1, since there is no scaling, initially.

Generating 2^i copies of bit x_i is accomplished by wires. Since the shuffle unit and the length adjust unit are both done with $\mathcal{O}(1)$ gate delays (as will be discussed in the following sections), the generator incurs $\mathcal{O}(1)$ gate delays as well.

4) Shuffle Unit: In order to let the 1's in the population vector π be uniformly randomly distributed, a 2-level shuffle unit is designed in our SP arithmetic system. The bits in π are grouped into W chunks. The *first level* of the shuffle unit generates W permutations of the chunks, and a particular permutation is selected by an LFSR that randomly cycles through a count between 1 and W. Next, within all the chunks, the *second level* generates w permutations of the bits within every chunk, and a particular permutation is selected by an LFSR that randomly cycles through a count between 1 and w. Note that $wW = \Pi$. In order to reduce the number of LFSRs, we select the same bit-level permutation for all the chunks. Additionally, a third LFSR randomly selects a logical shift of the resulting permuted number, performed by a barrel shifter. We choose between left and right shifts randomly. Further, the number of positions V to shift is also chosen randomly.

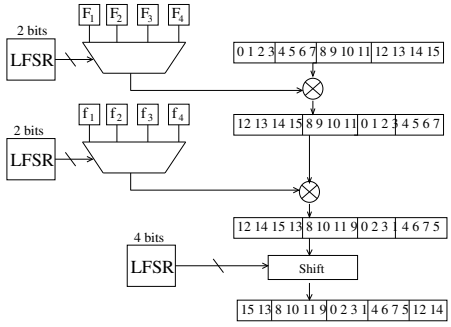


Fig. 7. Shuffle Example ($W = w = 4$)

Fig. 7 shows an example of the shuffle unit design. Since $\Pi = 16$, the position indices as shown on the top position vector range from 0 to 15. F_1, F_2, F_3, F_4 are different permutations for 4 chunks in π (i.e. $W = 4$), while f_1, f_2, f_3, f_4 are the bit-level permutation within the chunks (i.e. $w = 4$). The $\otimes$ symbol means applying the permutation on π, which essentially swaps bits around.

5) Density Checker Unit: In SP multiplication, we need to test if the population density $\frac{|\pi|}{\Pi}$ is greater than T_1, which is a threshold greater than 0.5. We convert the problem to check if $\frac{|\pi|}{\Pi} \times \frac{0.5}{T_1} > 0.5$. Note that $\frac{0.5}{T_1}$ is smaller than 1, and so we use a Π-bit mask with $\frac{0.5}{T_1} \times \Pi$ 1's, and bit-wise AND this mask with π in order to get $\frac{|\pi|}{\Pi} \times \frac{0.5}{T_1}$, which we denote as π_2. Next, we check if π_2 has a density greater than 0.5. Note that we need to do this approximately, with $\mathcal{O}(1)$ gate delays. Since the 1's in π_2 are randomly distributed, we can do this approximately by checking $\pi' = (\pi_2)|(\pi_2 << 1)|(\pi_2 << 2)$. In other words, we perform the logical OR of π_2 with its left-shifted counterparts (by 1 and 2 bit positions respectively). If the result π' is all 1s, we conclude that the density of $|\pi|$ is greater than T_1. The test of whether the number is all 1s is done by using a hash function (HF) of Class 3 [19], in order to ensure a $\mathcal{O}(1)$ gate delay. We hash π', and perform a bit-wise comparison of the result with the pre-computed hash of a population vector with Π 1's.

6) Scaling Unit: According to Lemma III.1, the density of the population vector needs to be adjusted by the scaling unit to achieve an improved accuracy for SP multiplication. Along with a scaling operation, the scaling term σ for each operand needs to be updated accordingly.

Suppose we would like to adjust the density of the population vector π by β, to yield the result $\pi \cdot \beta$. We will discuss how to process such an adjustment for different values of β.

1) When $0 < \beta \leq 1$, we use a mask M_β with Π bits, where there are $\beta\Pi$ 1's in the mask at arbitrary locations. Then we perform a bit-wise AND operation of the mask M_β with π, shuffle the result, and increase σ by $\frac{1}{\beta}$, in order to keep the numerical value of the SP representation to be the same. When $\beta = 1$, the density of the population vector π doesn't change.
2) When $1 < \beta < 2$, we solve the problem of computing $\beta \times |\pi|$ as follows. We first scale down π by $\frac{\beta}{2}$ by using the mask $M_{\frac{\beta}{2}}$ with density of 1's being $\beta/2$, and resulting population vector is called π'.
$$\pi' = \pi \& M_{\frac{\beta}{2}} \tag{1}$$
In Eq. 1, $M_{\frac{\beta}{2}}$ is a mask with Π bits, where $\frac{\beta}{2}\Pi$ bits are 1's. Since, $\beta \times |\pi| = 2 \times (|\pi| \times \frac{\beta}{2}) = 2 \times |\pi'|$, we next double the density of π' by using the following equation:
$$\pi^d = (\pi'|\pi'^s)|(\pi' \& \pi'^s) \tag{2}$$
In Eq. 2, π^d is the population vector after doubling, and π'^s is the population vector after shuffling π'. If the 1's are uniformly randomly distributed in π', $\pi'|\pi'^s$ will result in $2\pi' - \pi'^2$. Therefore, another term of $\pi' \& \pi'^s$ is added in Eq. 2 as a compensation for the numerical loss of π'^2. Due to the error of using bit-wise OR operation to perform addition, π^d is only an approximate version of $2\pi'$. However, it is computed with $\mathcal{O}(1)$ gate delays.
3) When $\beta = 2$, we first shuffle π and the result after shuffling is π'^s. Then compute by following Eq. 2, where π' in Eq. 2 is the same as π.
4) When $\beta > 2$, we repeatedly double $|\pi|$ until the remaining adjustment ratio is less than 2. Then we can use the methods mentioned above to compute $\pi \cdot \beta$.

IV. Experiment Results

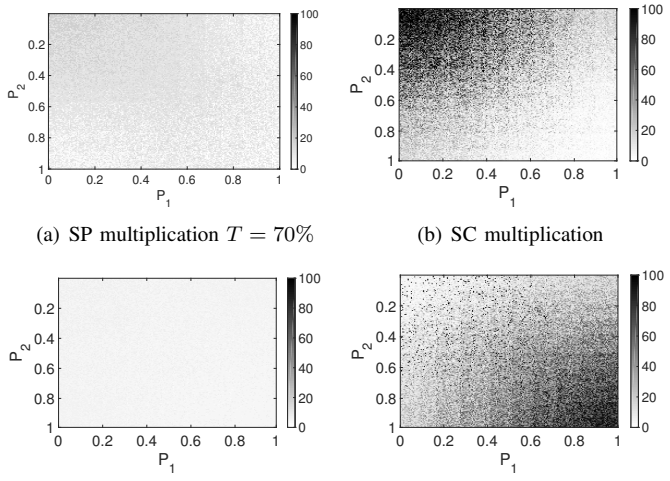
The proposed SP arithmetic scheme is evaluated for single multiplication/addition operations, matrix inner product computation, and MNIST image classification, in terms of accuracy, power, delay and area. It is implemented on a Zybo Zynq-7000 development broad with the following specifications: Xilinx XC7Z010-1CLG400C; number of look-up tables (LUTs): 17,600; number of flip-flops: 35,200.

A. Single Arithmetic Operation

We first show the results of evaluating accuracy. Our goal of SP arithmetic is to improve accuracy over SC. The accuracy is evaluated by comparing average relative errors (over a large number of operations) with respect to the conventional SC

TABLE I
ERROR CONTRIBUTION OF SP-BASED MULTIPLICATION (%)

T	Perfect	Generator	Density Check	Scale	Shuffle	Imperfect
50%	32.76(10.49)	32.91(10.87)	33.42(9.81)	35.02(12.66)	34.96(11.23)	35.12(13.98)
60%	16.39(4.33)	16.99(4.39)	17.73(5.91)	19.94(12.23)	18.30(7.92)	18.95(9.18)
70%	5.76(3.49)	5.91(3.50)	7.82(4.78)	9.94(11.28)	9.84(9.73)	10.26(11.02)



(a) SP multiplication $T = 70\%$ (b) SC multiplication

(c) SP addition (d) SC addition

Fig. 8. Relative errors proportional to darkness.

using IEEE 32-bit floating point representation as a baseline for comparison. The errors are depicted as heat maps in Fig. 8, where error is proportional to darkness. The two axes indicate $P_1 = \frac{|\pi_1|}{\Pi}$ and $P_2 = \frac{|\pi_2|}{\Pi}$, which are the densities of 1's in the population vectors of the two operands. For multiplication, the density threshold which we use to decide whether to scale or not is $T = 70\%$ for both operands. As expected, SC multiplication causes large errors for low densities, while SC addition results in large errors for high densities. By contrast, the errors from the SP arithmetic are much smaller.

Since an operand of multiplication is scaled if the density of 1's in its population vector is lower than T, we analyze the errors for two different situations. Errors e_{lo} are for the case where either operand has density less than T. Errors e_{hi} are for the case where both operands have densities greater than T. We report the separated error results in the format of $e_{lo}(e_{hi})$. In this format, the average errors from SP multiplications are $10.26\%(11.02\%)$ while those from SC multiplication are $71.04\%(3.19\%)$. Please note that we categorize SC multiplication errors into the $e_{lo}(e_{hi})$ format for the ease of comparison. When P_1 and P_2 are both larger than T, SP is supposed to process the multiplication in the same way as SC. However, since the density checker is not prefect, it might give the wrong judgement of whether P_1 or P_2 is smaller than T, which will introduce error. Therefore, e_{hi} actually increases for SP-based multiplication. On the other hand, when P_1 or P_2 is smaller than T, the density checker is less likely to make a wrong decision. Also even the density checker is wrong, scaling up a small fraction will cause less overflow error. Therefore, when when P_1 or P_2 is smaller than T, the error of SP-based multiplication decreases. As expected, introducing scale terms makes the density of the population vector to be controllable, which benefits the accuracy of multiplication. For addition, the average SP error is 5.83% while the error of SC is 23.43%.

We further show the errors from individual components of each SP arithmetic operation. In Table I, columns 3-6 display the errors from generator, density check, scaling unit and shuffle unit, respectively, for SP multiplication. When estimate the error of one component, we compute the other components using software, which is regarded as *perfect*. Column 2 summarizes the errors when all components are *perfect* and the errors are from the SP number representation alone, while the rightmost column is for the cases where all components are realized in circuits. Such decomposed error analysis is performed for different threshold levels. Note that the shuffle operation has the highest error contribution.

TABLE II
ERROR CONTRIBUTION OF SP-BASED ADDITION (%)

Perfect	Generator	Shuffle	Imperfect
0.84	1.21	3.57	5.83

The errors from different components in SP addition are provided in Table II. Please note that SP addition does not need the density checker or scaling unit. One can see that most of the errors are from the shuffle unit. The reason behind is that the shuffling results are not uniformly random enough.

TABLE III
EFFICIENCY COMPARISON FOR MULTIPLICATION ($\Pi = 32$)

	delay (ns)	#LUTs
SP	1.92	20
SC	4.28	16
Floating point	14.53	956
Fixed point	6.12	84

Next, we show the FPGA resource utilization and performance in comparison with SC, Floating point (conventional arithmetic using IEEE 32-bit floating point representation) and fixed point (conventional arithmetic using 32-bit fixed point representation). The results for multiplication are summarized in Table III. Note that for SP and SC, since the supporting units such as generator, shuffle unit, scaling unit, and density checker are shared among multiple operations, the area and LUTs used by these operations are not counted in Table III. For reference, if these units are counted, SP-based addition and multiplication use $7.13\times$ and $3.75\times$ fewer LUTs than the conventional floating point number based arithmetic circuit, respectively. One can see that SP arithmetic is $2.2\times$ faster than SC. On the other hand, SP arithmetic uses almost the same LUTs as conventional floating point arithmetic, and much less when considering the supporting units. Thus, SP reaches a compromise between SC and conventional arithmetic on performance and resource utilization. Meanwhile,

earlier results indicates its great accuracy improvement over SC. A similar trend can be observed for addition, whose results are in Table IV.

TABLE IV
EFFICIENCY COMPARISON FOR ADDITION ($\Pi = 32$)

	delay (ns)	#LUTs
SP	1.45	22
SC	3.24	18
Floating point	12.81	535
Fixed point	4.96	48

B. Application 1: Matrix Inner Product

The accuracy of SP arithmetic is also evaluated for the matrix inner product computation and the results are shown in Table V. Errors from individual operations accumulate in an application that contains many arithmetic operations. The errors increase with the vector size. The SP approach improves the accuracy by $20.72\times$ on average compared to SC. The errors from perfect SP and fixed point only comes from the limited resolution of the number representation. Therefore, they are more accurate. Since perfect SP uses population-based representation, which has a worse resolution than fixed points, which uses binary weighted representation. SP introduced slightly more error in the result.

TABLE V
ERROR OF MATRIX INNER PRODUCTION (%) ($\Pi = 32$)

Vector size	SC	SP	Perfect SP	Fixed point
32	172.50	16.55	1.00	0.21
64	283.52	18.92	1.11	0.38
128	420.15	21.03	1.19	0.46
256	589.39	24.59	1.29	0.49
512	797.42	28.02	1.38	0.52
Avg	452.21	21.82	1.19	0.41

C. Application 2: MNIST Digit Classification

The effect of SP approximation is also evaluated in a neural network application on MNIST digit classification [20]. The neural network has two hidden layers, 784 input nodes and 200 hidden layer nodes. The training set and test set have 60,000 samples and 10,000 samples, respectively. The size of each figure is 28×28. The multiplications and additions in the network are implemented with SP, SC, and conventional floating point and fixed point arithmetic. The classification success rates from these arithmetic methods of different bitwidth are listed in Table VI. On average, SP can reach accuracy of about 81%, which is a significant improvement over the 60% accuracy by SC.

TABLE VI
MNIST CLASSIFICATION SUCCESS RATE (%)

	SP	Perfect SP	SC	Fixed point	Floating point
32-bit	69.06	75.63	49.42	91.78	93.12
64-bit	78.41	84.72	60.07	92.62	93.68
128-bit	86.26	90.11	67.15	93.34	93.96
256-bit	90.86	92.74	70.26	93.99	94.11
Avg	81.15	85.81	61.72	92.93	93.71

V. CONCLUSION

In this paper, a scaled population arithmetic is proposed to improve accuracy compared to classical stochastic computation, by introducing a scaling (exponent) term, along with an adjustable population vector. Also, delays of the scheme are kept low by ensuring each operation uses $\mathcal{O}(1)$ gate delays. The SP arithmetic scheme improves the accuracy of multiplication and addition by $6.5\times$ and $4.0\times$, respectively. Its computation is also much faster than the classic stochastic computing. We also apply the SP arithmetic scheme on a MNIST image classification application and improve the accuracy by 32.79% compared to SC. In the future, more arithmetic operations will be designed, including subtraction, division and logarithm.

REFERENCES

[1] B. Grigorian, N. Farahpour, and G. Reinman, "Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing," in *HPCA*, 2015, pp. 615–626.

[2] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *ISCA*, 2015, pp. 554–566.

[3] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 505–516, 2014.

[4] Y. Wang, H. Li, and X. Li, "Real-time meets approximate computing: An elastic cnn inference accelerator with adaptive trade-off between qos and qor," in *DAC*, 2017, pp. 1–6.

[5] G. Keramidas, C. Kokkala, and I. Stamoulis, "Clumsy value cache: An approximate memoization technique for mobile gpu fragment shaders," in *Workshop on Approximate Computing*, 2015.

[6] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012, pp. 820–825.

[7] J. S. Vetter and S. Mittal, "Opportunities for nonvolatile memory systems in extreme-scale high-performance computing," *Computing in Science & Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[8] M. Samadi and S. Mahlke, "CPU-GPU collaboration for output quality monitoring," in *1st Workshop on Approximate Computing Across the System Stack*, 2014, pp. 1–3.

[9] B. R. Gaines, "Stochastic computing," in *Proceedings of the Joint Computer Conference*, 1967, pp. 149–156.

[10] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, p. 92, 2013.

[11] A. Alaghi, W. Qian, and J. P. Hayes, "The promise and challenge of stochastic computing," *TCAD*, vol. 37, no. 8, pp. 1515–1531, 2017.

[12] B. R. Gaines, "Stochastic computing systems," in *Advances in Information Systems Science*. Springer, 1969, pp. 37–172.

[13] A. Alaghi and J. P. Hayes, "Exploiting correlation in stochastic circuit design," in *ICCD*, 2013, pp. 39–46.

[14] T.-H. Chen and J. P. Hayes, "Design of division circuits for stochastic computing," in *ISVLSI*, 2016, pp. 116–121.

[15] A. Alaghi and J. P. Hayes, "A spectral transform approach to stochastic circuits," in *ICCD*, 2012, pp. 315–321.

[16] S.-J. Min, E.-W. Lee, and S.-I. Chae, "A study on the stochastic computation using the ratio of one pulses and zero pulses," in *ISCAS*, vol. 6, 1994, pp. 471–474.

[17] Z. Wang, S. Mohajer, and K. Bazargan, "Low latency parallel implementation of traditionally-called stochastic circuits using deterministic shuffling networks," in *ASPDAC*, 2018, pp. 337–342.

[18] M. Van Daalen, P. Jeavons, J. Shawe-Taylor, and D. Cohen, "Device for generating binary sequences for stochastic computing," *IEE Electronics Letters*, vol. 29, pp. 80–80, 1993.

[19] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, 1997.

[20] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, p. 18, 2010.