# REX-SC: Range-Extended Stochastic Computing Accumulation for Neural Network Acceleration

Tianmu Li, Wojciech Romaszkan, Sudhakar Pamarti, Puneet Gupta

***Abstract*—Deep learning has grown in capability and size in recent years, prompting research on alternative computing methods to cope with the increased compute cost. Stochastic computing (SC) promises higher compute efficiency with its compact compute units, but accuracy issues have prevented wide adoption, and accuracy-improving techniques have sacrificed runtime or training performance. In this work, we propose REX-SC - Range-Extended Stochastic Computing Accumulation to deal with the accuracy issues of stochastic computing. By modifying the functionality of OR-based SC accumulation, we increase SC computation accuracy without sacrificing the performance benefits. Our approach achieves a 2X reduction in stream length for the same accuracy compared to SC with OR-based accumulation and a up to 3.6X improvement in energy compared to SC with binary addition. With proper modeling, our approach improves training performance for SC-based neural networks and makes training SC models practical for large datasets like ImageNet.**

## I. Introduction

Machine learning has become ubiquitous in fields such as image recognition, voice recognition and machine translation [1], [2], [3]. Deep learning using neural networks has been dominant in machine learning because of its ease of training and generalization capabilities [4]. Neural networks have grown in complexity in the past few years [4], [1], [2], with storage and compute requirements of deep learning outpacing hardware improvement [5], [6]. This increased cost makes it hard to run deep neural networks on resource-constrained devices. Stochastic computing (SC) aims to improve deep learning efficiency [7], [8]. The compact computation unit of SC promises reduced computation area, improved data reuse, and reduced memory access costs [3]. Since SC requires only single gates for multiplication and addition, it enables a large number of multiply-accumulate units on the same hardware, which greatly improves operator reuse and alleviates memory bottlenecks of deep learning accelerators [3].

However, stochastic computing introduces random errors. The use of random number generators (RNGs) lead to inaccuracies during stream generation. Single-gate multiplication and addition operators are not guaranteed to be accurate, and very long bitstreams are required to reach acceptable accuracy [3], [9]. As a result, naively applying stochastic computing to neural networks reduces either accuracy or performance compared to traditional fixed-point and floating-point computation.

SC error mainly comes from stream generation, multiplication, and accumulation. Stream generation error can be minimized with maximal-length linear feedback shift registers (LFSR) and low-discrepancy (LD) sequences [10], [11]. Multiplication error can be reduced by utilizing a counter and a LD sequence for the two multiplicands [11], [12]. Despite improvements to generation and multiplication error, most of the previous works use fixed-point accumulation to preserve accuracy, and improvements to SC addition introduced compromises. [3] preserves SC addition by using OR gates for accumulation. However, even with cumbersome, dedicated training, OR gates drop model accuracy due to limited output precision. [10] improves addition precision by using a mix of OR gates and binary adders. Mixing the two accumulation methods complicates the training process.

In this work, we improve SC accumulation accuracy through REX-SC: Range-Extended Stochastic Computing accumulation, while preserving most of the performance benefits of stochastic computing. We build upon energy- and area-efficient OR accumulation and improve its accuracy by increasing the number of output bits relative to the input bits. Our contributions are as follows:

- We introduce extended range SC (REX-SC) addition methods with higher output precision while preserving the streaming nature of SC.
- We explore the extensive design space of REX-SC accumulation, and develop methods of finding optimal transfer functions.
- We show that REX-SC improves accuracy by 3-8% compared to previous methods utilizing SC accumulation [3] and reduces energy consumption by up to 3.6X compared to SC with binary accumulation.
- We propose training optimizations to improve the training speed of SC: stream computation simulation, activation calibration and error injection (a+e).
- We show that our optimizations improve training speed of SC models using REX-SC accumulation by $>$22 times compared to previous methods that achieve similar accuracy levels [10].

## II. Motivation

In this section, we motivate the need for new methods of SC accumulation. By analyzing the shortcomings of existing techniques through the lenses of *precision* and *efficiency* we expose a glaring design space gap, waiting to be explored. Most of the previous works on SC focused on stream generation and multiplication [10], [12], [11], [13]. Our work aims to mend this oversight, first by showing the importance of optimizing SC addition and then by developing such optimization techniques.

TABLE I: Summary of various SC addition methods and works using them.

| Type | Streaming | | | Fixed-Point | |
|---|---|---|---|---|---|
| Realization | MUX | OR | **REX** | Parallel Counter | Accumulator |
| Precision | Scaled | Approx. | **Approx.** | Exact | Exact |
| Area | Compact | Compact | **Compact** | Large | Large |
| Bipolar | Yes | No | **No** | No | Yes |
| Output Range | Same as Input | | **Higher than Input** | Configurable | |
| Example Usage | [9], [14] | [3], [10] | **This Work** | [9], [14], [13] | [11], [12] |

### A. SC Accumulation Primer

Stochastic computing addition techniques can be divided into two categories: streaming and fixed-point. The former preserves the number representation of the inputs, through the use of single-gate, streaming adder implementations, either a multiplexer (MUX) or an OR gate [8]. A MUX-based SC adder uses a random select signal with equal probability for each input, downsampling them to implement scaled addition [8]. In comparison, an OR-based adder does not introduce a scaling factor, but it realizes an approximate addition function: $Sum = A + B - AB$ [3]. The fixed-point addition techniques implicitly convert the streams into the fixed-point domain using parallel counters, accumulators, or a combination of both. Approximate or exact parallel counters add individual bits of multiple stochastic streams. Tab. I shows an overview of different SC accumulation methods.

### B. Precision

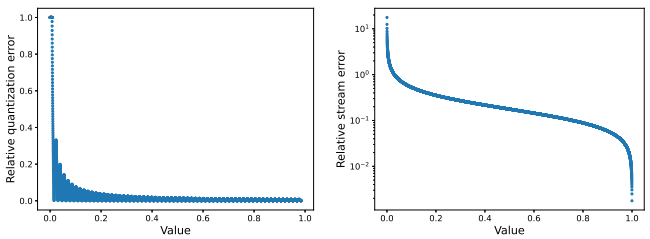
SC is an approximate computing method that can introduce random errors during both conversion and computation. Therefore, one of the chief concerns when designing SC systems is their accuracy. While extensive efforts have been directed at reducing the error of both stream generation [10], [11], and multiplication [11], [13], they have done little to explore and improve the accuracy of addition operators. The use of multiplexers (MUX) has been largely abandoned for addition because of their scaling factors equal the sizes of accumulation [9]. SC additions using MUX-based adders typically come in two variants. The first is where multiplications and additions are separated. Multiplications are performed using a traditional SC multiplier in the form of an AND or XNOR gate, and addition is performed using MUX with a random signal that uniformly selects between the different products. We denote this formulation of MUX-based adders as the multiply-add version of MUX, or M-MUX in short. For M-MUX, output of a dot product can be formulated as M-MUX$(a, b) = 1/n \sum a_i b_i$, where $a$, $b$ are the input vectors, and $n$ is the size of the dot product. The M-MUX adder thus scales the multiplication result by $1/n$ before adding them together. To understand how MUX downscaling affects the error of accumulation, we consider two components that contribute to the output error. The first is the quantization error. Stochastic computing typically utilizes uniform quantization to take advantage of the probabilistic compute units. Uniform quantization has consistent absolute quantization error regardless of magnitude. As such, the relative error from uniform quantization is larger for smaller values, which can be observed in Fig. 1a. The second component is the random error from stochastic computing. If streams are generated using a true random number generator (TRNG),[1] the expected output error can be derived from a binomial distribution and is equal to $E_{abs} = \sqrt{\frac{v(1-v)}{n}}$, where $v$ is the expected output value and $n$ is the stream length. The expected error is a concave function of $v$ with a small error for values close to 0 and 1, but the relative error $E_{rel} = \sqrt{\frac{1-v}{vn}}$ is a decreasing function of $v$ and is large close to zero, as is shown in Fig. 1b. Both components increase relative error for values close to 0, which M-MUX accumulation enforces for large $n$ values. Since M-MUX accumulation results need to be scaled up by $n$ to recover the true accumulation results, the large relative error translates directly to large absolute error. For neural networks with dot products going up to the thousands, MUX accumulation leads to a significant loss in accuracy, as the scaling factor is the same size as the dot product size, Fig. 2 compares the accuracy of MUX- and OR-based accumulation for 1000-wide accumulation. Inputs to the accumulation are sampled such that the sum of all inputs has a variance of 0.5 and mean of 1. Since neural networks are typically initialized to maintain unit variance throughout the model, it is a valid assumption to have 0.5 variance for half (positive values) of the inputs. Due to the scaling effect, M-MUX requires very long streams to achieve reasonable accuracy.

The other form of MUX addition fuses the multiplication component into the MUX adder and utilizes the select signal to achieve a weighted sum between inputs, which we denote as the weighted-add version of MUX, or W-MUX in short. Output of a dot product for this implementation can be formulated as

$$\text{W-MUX}(a, b) = \frac{\sum a_i b_i}{\sum |b_i|} \tag{1}$$

where $b$ is the weight vector. The select signal for the weighted sum can be further simplified to a counter as proposed in the CeMux implementation in [15]. Compared to the multiply-add version which scales all multiplication results by $1/n$, the weighted-add version alleviates scaling by normalizing the weight values to $\frac{b_i}{\sum |b_i|}$, as can be seen in Eq. 1. However, this also means that the weighted-add version does not fully resolve the scaling issue, as the weight values have an average magnitude of $1/n$ after normalization. With weight and output

[1]TRNG is used due to the simplicity of modeling, but other RNG sources and SC computation results show similar trends




(a) (b)

Fig. 1: Relative error resulting from (a) quantization and (b) randomness of stochastic computing.

quantization considered and the output scaled back to the original range, Eq. 1 becomes

$$\text{W-MUX}(a,b) = (\sum |b_i|)\text{Quant}(\sum a_i \text{Quant}(\frac{b_i}{\sum |b_i|})) \quad (2)$$

"Quant" is the quantization function that rounds the weights and addition results to $log2(b)$ bits, where $b$ is the bit stream length. This modeling represents a best-case scenario for W-MUX and ignores any additional error from SC computation (such as the ones shown in Fig. 1b). MUX_Ideal in Fig. 2 depicts this quantization effect assuming all weights are one. All weights being one is only one case and weight values are typically different in actual applications, so we considered its performance when training a neural network. We trained a small 4-layer CNN [16] on the CIFAR-10 dataset using the idealized model of the W-MUX adder as shown in Eq. 2. Due to the $1/n$ average weight value, *the stream length needs to be at least the size of the dot product for reliable representation of the weight values*. Since the largest dot product in the model is 1024, W-MUX fails to converge reliably and always drops to 10% accuracy either from the beginning (stream length $\leq 128$) or after a few epochs (stream length $\leq 512$). [2]Deeper neural networks have even larger dot products. The Resnet models [2] used in Sec. IV, for instance, have layers with 3x3x512-size filters, which translates to 4608-sized dot products. Large dot products in turn require longer streams to represent the weight values. As we will show in Sec. IV, stochastic computing has trouble competing against fixed-point computation with stream lengths $>> 64$. Since prior works on MUX addition can be categorized either into M-MUX or W-MUX and both suffer from the scaling issue, it is difficult to achieve good accuracy with reasonable stream length using MUX-based addition.

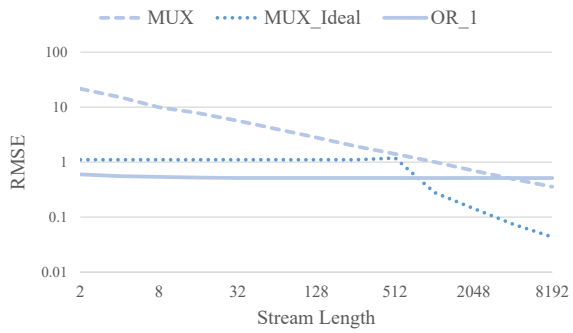



Fig. 2: Comparison of MUX- and OR-based adders' error with respect to accurate addition.

OR-based adders, while not troubled by scaling factors, come with their own set of issues. First, as mentioned before, they do not implement exact addition. For two inputs $a$ and $b$, an OR gate performs $a + b - ab$ compared to $a + b$ in an exact addition, which leads to output saturation for high-magnitude inputs. While the saturation of OR-based adders can be alleviated with small input values, small input values increase relative error, as is discussed in the previous discussions on MUX. Second, the outputs of OR accumulation have the same precision as the inputs due to the bit-wise computation, which reduces accuracy compared to accumulation without truncation. Recall that adding two N-bit fixed-point numbers requires N+1 bits to avoid overflow. Prior works have shown that algorithms can be trained for approximate addition and saturation for the first and second issue, for example, by using custom neural network training [3], [10]. Unfortunately, they cannot correct for the loss in intermediate output precision. Other previous works have tried to combine the OR accumulation and fixed-point accumulation to achieve a better trade-off between the two [10]. While it does improve accuracy, combining the two dramatically slows down the training process, as discussed in Section IV-C.

In contrast, using various forms of fixed-point accumulation achieves accurate or near-accurate summation between inputs, making them the most common choice of addition implementation in recent SC works [11], [12], [13]. However, as we will show in the next subsection, they are expensive to implement since full adders required in APCs are larger, slower, and less energy efficient than OR gates, or multiplexers [9].

### C. Efficiency

Despite the precision issues outlined above, SC remains a promising candidate for accelerating various types of computation. The reason for the continued interest in it is the potential for unparalleled compute density [3], [13]. Small, single-gate SC multipliers and adders can be orders of magnitude smaller than their fixed-point equivalents. However, this comes at the cost of increased computation latency, as SC operators require tens or hundreds of cycles depending on the stream length being used [3], [9], [13]. To harness the most benefits out of SC, the area reduction offered by multipliers and adders needs to be used to enable higher spatial reuse, which in turn can amortize costs of conversion and memory accesses. When playing the spatial reuse game, the additional area and delay imposed by binary accumulation can wipe off a large part of the gains, as we will show shortly.

To demonstrate the difference between different styles of accumulation, we synthesized 256-wide dot-products with different processing element. Wide dot product is justifiable for modern neural networks that require multiply-accumulate computations with hundreds or thousands of inputs [1], [2]. We compare fixed-point (*FXP*) multiply-accumulate computation and SC computation using accumulators (*Accumulator*), parallel counters (*Parallel Counter*), and OR gates (*OR*). We assume 6-bit input precision, which translates to 64 bits for the SC accumulators. Parallel counter and OR adder trees use split-unipolar computation [3], which assumes one of the operands is bipolar while the other one can only be positive. The 64 bits are split into two 32-bit parts for the bipolar operand representing the positive and negative values separately. For positive values, the negative part is all zero, and vice versa. This is also a commonly encountered situation in neural networks employing the ReLU activation [1], [2]. For SC multiplication, we assume that it is performed using simple AND gates. The accumulator configuration assumes 256 individual

[2]Longer stream lengths may still face convergence issues with W-MUX but is not supported in the SC simulation framework used.

accumulators. We further compare non-saturating (*NS*) and saturating (*S*) adder-trees for all configurations except the OR addition. The non-saturating configurations have adder width provisioned such that no truncation ever occurs. Saturating configurations limit the width of all adders to 6-bits. We will further elaborate on the importance of saturation in Section III. Results, synthesized using a commercial 28nm technology and Cadence Genus synthesis tool, are shown in Fig. 3.

Compared to the fixed-point baseline, SC offers significant area improvements. Non-saturating parallel counter implementation has 22.5X area improvement over the corresponding fixed-point design, while for a dot product with OR accumulation that advantage is 58.4X. The accumulator-based dot product can have a 2X area advantage, despite less efficient addition, because of the huge advantage provided by AND-based multipliers. This explains why previous works have focused on optimizing SC multiplication since it is where most of the benefits of SC come from. However, the impact of addition cannot be ignored due to the SC's latency penalties. The non-saturating accumulator, parallel counter, and OR dot products have a 3.2X, 1.6X, and 3.8X critical path advantage over fixed-point. When accounting for stream length and combined with area in the form of area-delay product, OR implementation has a 7X advantage, while the parallel-counter one has only 1.2X. The accumulator one has 5X worse throughput, showing that its use is not competitive without additional techniques reducing the stream length [11], [12]. The ADP difference between OR and parallel-counter dot-products leads us to two important conclusions. First, *SC with binary accumulation has a fundamental limitation in possible performance improvement over fixed-point*. The small ADP margin can easily be whittled down when other system-level considerations come into play. Second, *there is a substantial performance gap between SC with binary and OR-based accumulation*. This *performance gap*, combined with the *precision gap* described in the previous section, *opens up an extensive design space of efficient and precise SC accumulation methods* that has not been given sufficient attention. In the next section, we will describe how this efficiency-precision gap can be bridged.

Finally, we want to acknowledge that the above analysis will depend on the dot-product size, precision and stream length chosen, and different configurations might be more or less advantageous to certain implementations.

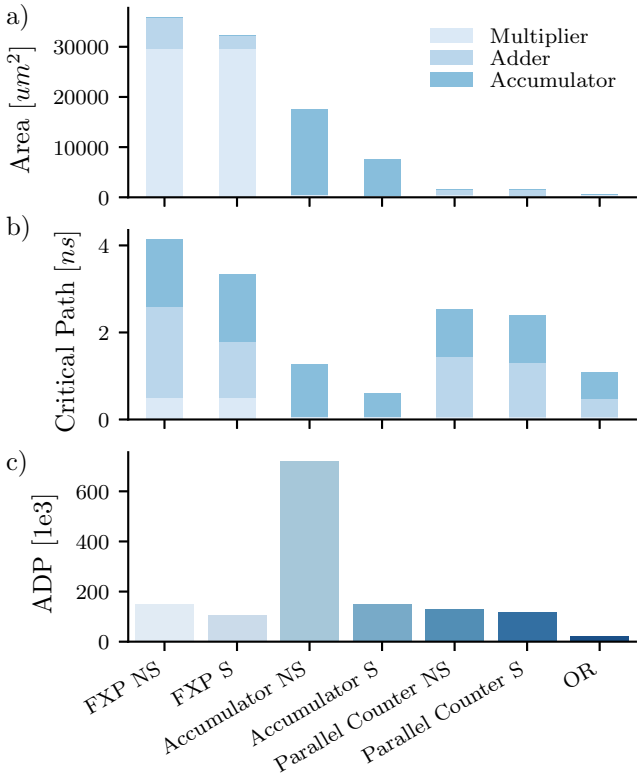



Fig. 3: Area (a), critical path (b) and area-delay product (ADP) (c) for different dot product implementations. For SC implementations, ADP is calculated after multiplying the stream length, which is assumed to be 32 for each split-unipolar part.

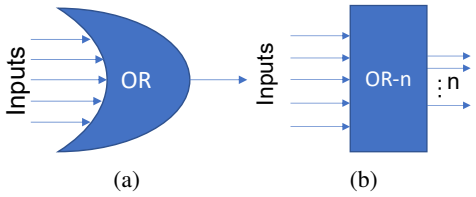



Fig. 4: Overview comparison between (a) OR and (b) OR_n.

## III. Range-Extended OR Accumulation

### A. Increasing the accuracy of OR accumulation

Given the importance of maintaining SC accumulation performance II-C, we want to find a way to improve accumulation accuracy without sacrificing the benefits it offers. Between the three possible baseline implementations, MUX-based adders suffer from scaling on the inputs. Improving MUX accuracy requires increasing input stream length, which also affects the performance of multiplication and stream generation. Accumulator and counter-based adders achieve accurate addition and have performance issues instead of accuracy issues. OR-based adders suffer from the lack of output range. Adjusting the output range keeps the stream generation and multiplication

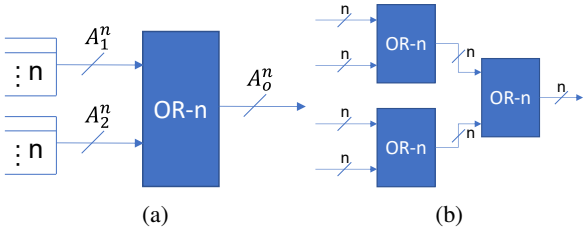



Fig. 5: (a) Single OR_n gate and (b) cascaded OR_n gates.

components constant, and is thus the most plausible candidate for improvement. The accuracy deficit of OR accumulation comes from the bitwise computation nature. The outputs of an OR accumulation have the same stream length as the inputs. Fixed-point accumulation adds all input bits together. It is equivalent to concatenating the input bit streams together, resulting in much longer effective stream length at the output. Therefore, to increase the output precision of an OR gate, we want to increase the number of output bits corresponding to each input bit to $n$. We denote the resulting range-extended OR gate as OR_n, and a regular OR gate can be seen as a special case where $n = 1$. A comparison of an OR_n gate and a regular OR gate is shown in Fig. 4. In theory, any logic operation that takes 1 bit from each input and outputs n bits is a valid OR_n gate, leading to $2^{512}$ unique truth tables exist for OR_2 with 256 inputs (256 input bits, 2 output bits). We will discuss how we determine the OR_2 gate, and then extend the derivation to larger OR_n.

Given the large number of possible OR_2 configurations, we limit our search space through the following steps:

1) Limit the search to two-input OR_n gates. Fig.5a illustrates the idea of a 2-input OR_n gate. $A^n$ represents n bits, and this notation will be used in later discussions to represent $n$ input or output bits. A two-input OR_n gate takes two $A^n$ inputs and creates an $A^n$ output. Larger OR_n gates can be built by cascading multiple levels of two-input OR_n gates, as shown in Fig. 5b). Focusing on two-input gates ensures that the cost of large accumulations scale linearly with the size of accumulation, and reduces the search space for OR_n to $2^{n2^{2n}}$ (2n input bits, n output bits), or $2^{32}$ for OR_2.
2) Focus on the transfer function between the sum of input bits and the sum of output bits. In other words, instead of trying to find the function $A_o^n = f(A_1^n, A_2^n)$, we try to find the sum transfer function $\sum(A_o^n) = f_s(\sum(A_1^n) + \sum(A_2^n))$. The exact form of the function will be determined later after considering other constraints. This concept is illustrated in Tab. II, which depicts one possible implementation of OR_2. The sum transfer function loses positional information of the individual input bits and instead forces the computation to be associative. In other words, the order of the input bits does not affect the sum of the output bits, and the output bits can be adjusted as long as the sum does not change. As discussed in Sec. III-C, the associative constraint allows for more efficient training of models using OR_n accumulation. The sum transfer function constraint reduces the search space to $(n+1)^{2n+1}$ (n+1 output sum values from n output bits, 2n+1 input sum values from 2n input bits), or $3^5$ for OR_2.
3) The sum transfer function should be non-decreasing, and should use all possible sum values of the n output bits. The former ensures that OR_n behaves similarly to normal accumulation, which simplifies training. The latter ensures that the output bits are fully utilized. These two constraints limit the search space to $\binom{2n}{n}$ (n increments in 2n entries), or 6 for OR_2.

Tab. III shows the 6 candidate transfer functions for the OR_2 gates that satisfy all the constraints.

To find the best among the 6 candidate transfer functions, we examine their behaviors when expanded to 3 OR_2 inputs shown in Tab. IV. Of the 6 candidates, half of them loses the associative property, and the output depends on the position of the input. Take candidate 2 as an example. $OR\_2\_2(OR\_2\_2(\{0,1\},\{0,1\}),\{0,0\}) = \{0,0\}$ whereas $OR\_2\_2(OR\_2\_2(\{0,0\},\{0,0\}),\{1,1\}) = \{0,1\}$, even though the input bits add up to 2 for both cases. For the rest of the three candidates, candidate 1 only activates when almost all input bits are 1, which is unlikely for neural networks with very wide accumulations. Candidate 4 faces a similar issue where the second bit only activates when all input bits are 1. We expect OR_2 using candidate 1 or 4 to perform poorly, and this is confirmed in Tab. V. For the same 4-layer CNN (TinyConv) used in [16], candidate 1 does not converge, candidate 4 barely offers any benefit over OR_1, and only candidate 6 is a noticeable improvement over OR_1. As a result, we choose candidate 6 as the sum transfer function of OR_2.

The transfer function of OR_2 can be written as

$$\sum(A_o^2) = \begin{cases} \sum(A_1^2) + \sum(A_2^2), & \sum(A_1^2) + \sum(A_2^2) < 2 \\ 2, & \text{otherwise} \end{cases} \tag{3}$$

where $A_o^2$ is the output tuple and $A_1^2$ and $A_2^2$ are the input tuples. Larger OR_n can be derived similarly by replacing 2 in Eq. 3 with $n$, as shown in Eq. 4. Alternatively, OR_n can also be viewed as a saturating adder that saturates at $n$. A key feature that sets OR_n apart from other designs with a saturating adder/accumulator is that it saturates after every addition and operates only on unipolar inputs. The former ensures that an OR_n gate does not introduce additional bits, as an OR_2 gate will always generate 2 bits and an OR_3 gate will always generate 2/3 bits, depending on the implementation. This linear scaling with the size of

| Input Bits | | | | Input Sum | Output Bits | | Output Sum |
|---|---|---|---|---|---|---|---|
| $a$ | $b$ | $c$ | $d$ | | $e$ | $f$ | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 2 | 1 | 1 | 2 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 2 | 1 | 1 | 2 |
| 0 | 1 | 1 | 0 | 2 | 1 | 1 | 2 |
| 0 | 1 | 1 | 1 | 3 | 1 | 1 | 2 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 2 | 1 | 1 | 2 |
| 1 | 0 | 1 | 0 | 2 | 1 | 1 | 2 |
| 1 | 0 | 1 | 1 | 3 | 1 | 1 | 2 |
| 1 | 1 | 0 | 0 | 2 | 1 | 1 | 2 |
| 1 | 1 | 0 | 1 | 3 | 1 | 1 | 2 |
| 1 | 1 | 1 | 1 | 4 | 1 | 1 | 2 |

TABLE II: Comparison between a full truth table and a sum transfer function. This is one possible truth table of a two-input OR_2 gate that corresponds to Candidate 6 in Tab. III. This particular truth table sets the two output bits to $e = a + c + bd$ and $f = b + d + ac$ respectively, assuming $a$, $b$, $c$, and $d$ are the four input bits, and $e$ and $f$ are the two output bits.

accumulation is advantageous. The latter feature ensures that saturation is well-defined. Hardware implementations saturate the outputs after each OR_n accumulation, minimizing the number of wires out of an OR_n gate and allowing the same OR_n gate to be reused for larger accumulations. In software modeling of OR_n, saturation can be delayed to the end to better utilize dot-product instructions present in recent CPUs and GPUs while still being functionally identical to the hardware version. A bipolar saturating adder, on the other hand, lacks this convenience. For example, if we consider a saturating addition that saturates at {5,-5} between 3, 4, and -5, saturating after adding 3 and 4 will produce a different result compared to saturating after adding all three values together. The accumulation result from a bipolar saturating adder will depend on the order of computation and will be more difficult to model, particularly if saturation occurs frequently.

$$\sum(A_o^n) = \begin{cases} \sum(A_1^n) + \sum(A_2^n), & \sum(A_1^n) + \sum(A_2^n) < n \\ n, & \text{otherwise} \end{cases} \tag{4}$$

Despite focusing on the ease of implementation when deciding between candidates, the final OR_n offers lower error compared to OR when normalized to the same total output stream length. This means using $k$ input bits for OR_n and $kn$ input bits for OR, resulting in $kn$ output bits for both cases. Assuming different bits in a stream have independent and idential distribution (IID), the output value follows a binomial distribution, with root-mean-squared (RMS) error being $\sqrt{\frac{v(1-v)}{kn}}$, where $v$ is the expected error and $kn$ is the stream length. This is a concave function over $v$ [3]. Due to the concavity of the error function, having different probability values in different parts of the stream reduces the final average error. For OR_n, each group of $k$ output bits represents a different value. As a result, we expect even OR_2 to outperform OR in accuracy when using half the input stream length.

[3]This error function assumes using a true random number generator (TRNG) for generation. Using a more accurate generator reduces overall error, but the error is still a concave function with respect to the expected output value

| Candidates | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Input Sum | Output Sum | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | 1 | 1 | 2 | 1 | 2 | 2 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 |

TABLE III: Candidate transfer functions between input and output sums for 2-input OR_2.

| Candidates | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Input Sum | Output Sum | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0,1 | 0,1 | 1 | 1 | 2 |
| 3 | 0 | 0,1 | 1 | 1 | 1,2 | 2 |
| 4 | 0 | 1 | 1,2 | 1 | 1,2 | 2 |
| 5 | 1 | 1 | 2 | 1 | 2 | 2 |
| 6 | 2 | 2 | 2 | 2 | 2 | 2 |

TABLE IV: Candidate transfer functions between input and output sums for 3-input OR_2.

### B. *Efficient OR_n implementation*

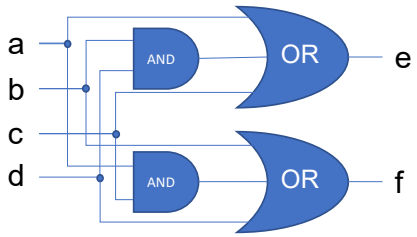



Fig. 6: OR_2 circuit implementation. Similar to Tab. II, a, b, c, and d are the four input bits, and e and f are the two output bits.

While Section III-A defines the optimal transfer function between input and output sums for OR_n, the transfer function does not uniquely define the truth table. Take OR_2, for instance. Given two $A^2$ inputs 00 and 01, the transfer function determines that the $A^2$ outputs sum up to 1 but does not decide whether the outputs should be 01 or 10. For OR_2, there are four output positions that can be either 01 or 10, resulting in 16 possible ~~choices~~ truth tables similar to the Tab. II. Fig.6 shows a possible circuit implementation of OR_2 with the truth table in Tab. II. We synthesized 256-wide adder trees for all 16 ~~choices~~ truth tables, and the area-delay product comparison is shown in Fig. 7. Configuration 4 is the best performing option, and we used that for all performance evaluations of OR_2. It is also 3.3X better in terms of ADP compared to non-saturating parallel-counter based addition.

OR_3 uses a saturating adder instead of the parallel implementation of OR_2. From Eq. 4, OR_n behaves like a saturating adder that saturates at $n$. Instead of adding the three parallel output bits after accumulation, every three input

| Candidate | OR_1 | 1 | 4 | 6 |
|---|---|---|---|---|
| Accuracy | 74.81% | 10% | 75.31% | 78.92% |

TABLE V: TinyConv Neural network accuracy of Candidate 1, 4, and 6 for OR_2. Models are trained using 32-bit streams on CIFAR-10.

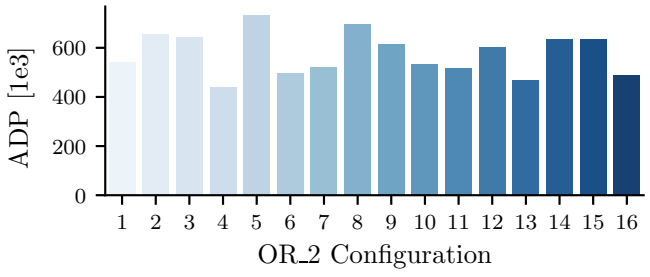


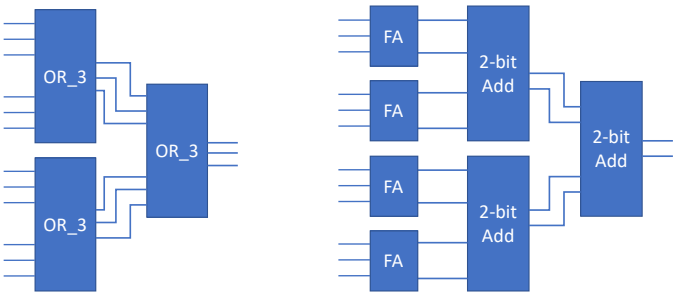

Fig. 7: ADP comparison between different OR_2 truth table implementations.

bits are added together using a full adder, and 2-bit saturating adders perform the rest of the accumulation. The two implementations are illustrated in Fig. 8. Tab. VI and Tab. VII show the truth tables for the parallel and saturating adder implementations respectively. We synthesized both options, and the saturating adder version has ADP 2X lower than the parallel implementation, so we use the saturating adder version for performance evaluations of OR_3. Its ADP improvement over parallel-counter addition is 1.6X. Because of that, we do not explore $n > 3$, since we do not expect it to have better performance than fixed-point accumulation.

### C. *Efficient training for OR_n accumulation*

While we focus on improving inference performance with SC and OR_n, we need to ensure that models are trained so that they work well for OR_n SC during inference. Training is an important aspect of deep learning, and modifications to the compute algorithm during inference need to be easily trainable on commodity hardware. Our training setup is based on the idea of straight-through estimators (STE) [17]. Fig. 13a shows the basic STE setup for training inaccurate neurons. The forward pass accurately models the inaccurate computation (in our case SC stream computation), while the backward pass ignores the inaccuracies. For REX-SC, we use an efficient simulation framework to simulate SC-based computation in the forward pass. This ensures that the training results are representative of what we will get on SC hardware. Every 32 bits in an SC bitstream is packed to a 32-bit integer to maximize performance. The simulation framework is written in CUDA C++[18] to maximize performance on Nvidia GPUs and utilizes AVX2 intrinsics[19] with OpenMP [20] to maximize performance on x86 CPUs. WMMA functions[21] are used for Nvidia GPUs with Compute Capability $\geq 8.0$.

Backpropagation uses floating-point computation during training, and the non-linearity of OR_n accumulation requires modeling in the backward pass. To understand the need for additional modeling during the backward pass, we need to consider the saturation behavior of OR_n. As OR_n saturates at n, output values close to n should result in lower gradient values on the corresponding inputs and weights. The vanilla STE setup ignores the saturation behavior during the backward pass and thus degrades the accuracy of the final trained




(a) (b)

Fig. 8: (a) Parallel and (b) saturating adder implementation of OR_3.

| Input Bits | | | | | | Input Sum | Output Bits | | | Output Sum |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 2 | 0 | 1 | 1 | 2 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 2 | 1 | 0 | 1 | 2 |
| 0 | 0 | 0 | 1 | 1 | 0 | 2 | 1 | 1 | 0 | 2 |
| 0 | 0 | 0 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 3 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 2 | 1 | 0 | 1 | 2 |
| 0 | 0 | 1 | 0 | 1 | 0 | 2 | 0 | 1 | 1 | 2 |
| 0 | 0 | 1 | 0 | 1 | 1 | 3 | 1 | 1 | 1 | 3 |
| 0 | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 0 | 1 | 2 |
| 0 | 0 | 1 | 1 | 0 | 1 | 3 | 1 | 1 | 1 | 3 |
| 0 | 0 | 1 | 1 | 1 | 0 | 3 | 1 | 1 | 1 | 3 |
| 0 | 0 | 1 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 2 | 0 | 1 | 1 | 2 |
| 0 | 1 | 0 | 0 | 1 | 0 | 2 | 0 | 1 | 1 | 2 |
| 0 | 1 | 0 | 0 | 1 | 1 | 3 | 1 | 1 | 1 | 3 |
| 0 | 1 | 0 | 1 | 0 | 0 | 2 | 1 | 1 | 0 | 2 |
| 0 | 1 | 0 | 1 | 0 | 1 | 3 | 1 | 1 | 1 | 3 |
| 0 | 1 | 0 | 1 | 1 | 0 | 3 | 1 | 1 | 1 | 3 |
| 0 | 1 | 0 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 3 |
| 0 | 1 | 1 | 0 | 0 | 0 | 2 | 0 | 1 | 1 | 2 |
| 0 | 1 | 1 | 0 | 0 | 1 | 3 | 1 | 1 | 1 | 3 |
| 0 | 1 | 1 | 0 | 1 | 0 | 3 | 1 | 1 | 1 | 3 |
| 0 | 1 | 1 | 0 | 1 | 1 | 4 | 1 | 1 | 1 | 3 |
| 0 | 1 | 1 | 1 | 0 | 0 | 3 | 1 | 1 | 1 | 3 |
| 0 | 1 | 1 | 1 | 0 | 1 | 4 | 1 | 1 | 1 | 3 |
| 0 | 1 | 1 | 1 | 1 | 0 | 4 | 1 | 1 | 1 | 3 |
| 0 | 1 | 1 | 1 | 1 | 1 | 5 | 1 | 1 | 1 | 3 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 2 | 1 | 0 | 1 | 2 |
| 1 | 0 | 0 | 0 | 1 | 0 | 2 | 1 | 1 | 0 | 2 |
| 1 | 0 | 0 | 0 | 1 | 1 | 3 | 1 | 1 | 1 | 3 |
| 1 | 0 | 0 | 1 | 0 | 0 | 2 | 1 | 1 | 0 | 2 |
| 1 | 0 | 0 | 1 | 0 | 1 | 3 | 1 | 1 | 1 | 3 |
| 1 | 0 | 0 | 1 | 1 | 0 | 3 | 1 | 1 | 1 | 3 |
| 1 | 0 | 0 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 3 |
| 1 | 0 | 1 | 0 | 0 | 0 | 2 | 1 | 0 | 1 | 2 |
| 1 | 0 | 1 | 0 | 0 | 1 | 3 | 1 | 1 | 1 | 3 |
| 1 | 0 | 1 | 0 | 1 | 0 | 3 | 1 | 1 | 1 | 3 |
| 1 | 0 | 1 | 0 | 1 | 1 | 4 | 1 | 1 | 1 | 3 |
| 1 | 0 | 1 | 1 | 0 | 0 | 3 | 1 | 1 | 1 | 3 |
| 1 | 0 | 1 | 1 | 0 | 1 | 4 | 1 | 1 | 1 | 3 |
| 1 | 0 | 1 | 1 | 1 | 0 | 4 | 1 | 1 | 1 | 3 |
| 1 | 0 | 1 | 1 | 1 | 1 | 5 | 1 | 1 | 1 | 3 |
| 1 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | 2 |
| 1 | 1 | 0 | 0 | 0 | 1 | 3 | 1 | 1 | 1 | 3 |
| 1 | 1 | 0 | 0 | 1 | 0 | 3 | 1 | 1 | 1 | 3 |
| 1 | 1 | 0 | 0 | 1 | 1 | 4 | 1 | 1 | 1 | 3 |
| 1 | 1 | 0 | 1 | 0 | 0 | 3 | 1 | 1 | 1 | 3 |
| 1 | 1 | 0 | 1 | 0 | 1 | 4 | 1 | 1 | 1 | 3 |
| 1 | 1 | 0 | 1 | 1 | 0 | 4 | 1 | 1 | 1 | 3 |
| 1 | 1 | 0 | 1 | 1 | 1 | 5 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 0 | 0 | 0 | 3 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 0 | 0 | 1 | 4 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 0 | 1 | 0 | 4 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 0 | 1 | 1 | 5 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 1 | 0 | 0 | 4 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 1 | 0 | 1 | 5 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 1 | 1 | 0 | 5 | 1 | 1 | 1 | 3 |
| 1 | 1 | 1 | 1 | 1 | 1 | 6 | 1 | 1 | 1 | 3 |

TABLE VI: Truth table for the parallel implementation of OR_3.

| Input Bits | | | | Output Bits | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

TABLE VII: Truth table for the saturating adder implementation of OR_3

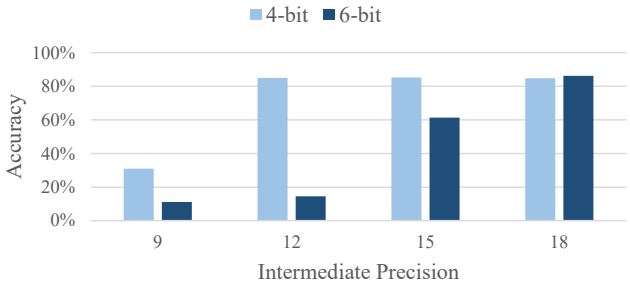



Fig. 9: TinyConv accuracy of with fixed-point computation using different intermediate precision.

model. This is an issue even for fixed-point quantization with limited accumulation precision, as shown in Fig. 9. When there is not enough intermediate precision and the accumulation saturates, accuracy drops significantly when the saturation is not modeled. [3] shows that it is possible to model OR accumulation as normal accumulation + activation function. This allows the usage of optimized convolution and linear kernels in popular deep learning frameworks. OR_n modeling uses a similar concept to modeling OR_1, so we will go through the derivations for OR_1, and then extend the idea to OR_n. For an OR_1 gate with inputs $a_k$, the expected output value is equal to

$$OR\_1 = 1 - prob(0\ input\ is\ 1) = 1 - \prod(1 - a_k) \quad (5)$$

By assuming all inputs are the same, Eq. 5 simplifies to $1 - (1 - s/n)^n$, where s is the normal sum of the inputs. When n is large, it further simplifies to

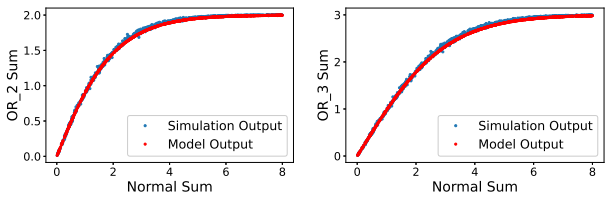
$$OR\_1 \approx 1 - e^{-s} \quad (6)$$

Similar concepts can be applied to OR_2. The OR_2 output can then be approximated as

$$OR\_2(a_k) \approx 2 - 2e^{-s} - se^{-s} \quad (7)$$

The same idea can be extended to higher-degree OR_n, which is shown in Eq. 8.

$$OR\_n(a_k) \approx n - \sum_{i=0}^{n-1} \frac{(n-i)s^i}{i!} e^{-s} \quad (8)$$




(a) (b)

Fig. 10: Modeling performance of (a) OR_2 and (b) OR_3.

The backpropagation through a neural network layer using OR_n accumulation is modeled as a normal convolution/linear backward pass (layer_bwd) with an added pointwise function, as is shown in Eq. 9 and 10, where $g_o$ is the output gradient

$$OR\_n\_layer\_bwd(g_o) = layer\_bwd(OR\_n\_bwd(g_o)) \quad (9)$$

$$OR\_n\_bwd(g_o) = g_o * (1 + \sum_{i=0}^{n-1} \frac{s^i}{i!} e^{-s}) \quad (10)$$

Fig. 10 shows the modeling performance of OR_n approximations. 100 inputs are used in the accumulation, each having a stream length of 1024. Our proposed approximation correctly captures the trend of OR_n accumulation. Since the approximation is only a function of normal accumulation sum S, it can also be modeled as a single activation function and is efficient to implement during training. Such models are only possible when the OR_n function satisfies the associative property mentioned in Sec. III-A. Consider partial binary accumulation [10] (PB) which also tries to extend the range of SC accumulation. In PB, large accumulations are broken up into multiple groups. Each group uses OR gates for accumulation, and results from different groups are added together using binary adders. Using two kinds of accumulators make PB non-associative, which is demonstrated in Fig. 11. Compared to OR_2 that maintains its accumulation behavior regardless of order of computation, PB has different behaviors depending on how the accumulation is broken up. The output response is very different when the two groups are roughly equal (pb_r) and when the two groups are very different (pb_s). While pb_r behaves similar to OR_2, pb_s barely offers any benefit over OR_1. Given the randomness of neural networks, it is impossible to predict how balanced the groups are, and thus impossible to use a single activation function. This will have profound impact on training performance, as we will show in Sec.IV-C.

While the stream simulation improvements for the forward pass reduce forward propagation time, it is still expensive compared to normal forward pass using single/half-precision floating-point numbers. To further reduce the training overhead for REX-SC, we propose to replace stream simulation with activation calibration and error injection (a+e for short) for most of the training epochs. Since the activation approximation is sufficient for the backward pass, one may ask why it is not sufficient for the forward pass. The main issue is that the activation functions cannot completely capture the characteristics of OR_n accumulation. Fig. 12 shows the mean and standard deviation of the difference between actual stream

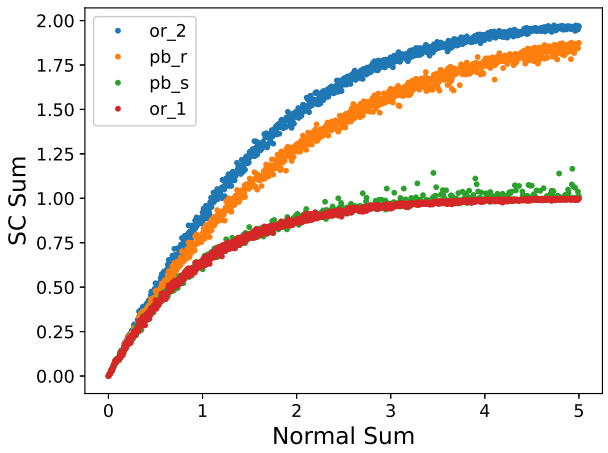
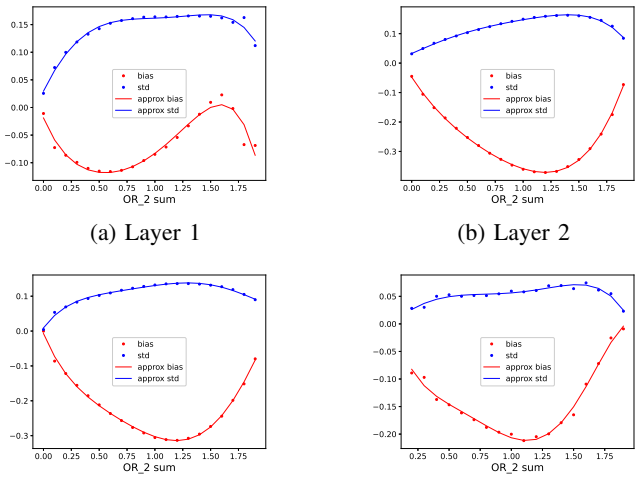



Fig. 11: Modeling performance comparison of OR_2 and 2-way partial binary accumulation. pb_r and pb_s use the same inputs, and only the order of computation is changed, such that for pb_r partial sums are roughly equal, and for pb_s they are very different.




(a) Layer 1 (b) Layer 2

(c) Layer 3 (d) Layer 4

Fig. 12: Difference between outputs from stream computation and from normal computation+activation.

output and activation output for the same 4-layer CNN used in Tab. V. The average error (denoted as bias in the figure) is non-zero, implying that the activation function is not perfect. Since the average error is different for each layer, it is impossible to use a single activation function for all layers. The standard deviation of the error (denoted as std in the figure) means it is impossible to capture all the properties of OR_n accumulation with only the activation function. Fortunately, both mean and variance of the error are smooth functions of the activation value. We can thus approximate the two parts of the error with polynomial functions using standard linear least squares regression [22], as is shown in the curves. The fitted mean curve is then added to the post-activation value as a calibration step for the activation, and the variance curve is used to inject random error to each activation with variance equal to the value predicted by the curve. Fig. 13 compares the training setup of stream and a+e training step. While a+e step seems more complicated, the additional operators are all point-wise operations and can be fused to reduce overhead. The step to update the error curves for a+e is performed only a few times per epoch of training to amortize its relatively large cost.

With the optimized stream computation simulation, backward propagation using activation function, and a+e step, the overall training setup is shown in Alg. 1. Models are first trained using the a+e training step, and then fine-tuned with stream training for a few epochs.

**Algorithm 1** Training setup for OR_n.

**Require:** Epochs with a+e training $n_e$, epochs with stream training $n_s$, number of training steps between a+e update $k_e$, total number of training mini batches $k_0$, dataset $d$.

```
n ← 0
while n < n_e do                                   ▷ a+e training phase
    k ← 0
    while k < k_0 do
        x ← d[k]
        if k mod k_e = k_e − 1 then ▷ update a+e parameters
            for all layers do
                x_l ← layer input
                y_{a+e} ← layer_forward_a+e(x)
                y_stream ← layer_forward_stream(x)
                Update layer mean and variance curves
            end for
        else
            x ← model input
            y = model_forward_a+e(x)
        end if
        model_backward_a+e(y, model target)
        k ← k + 1
    end while
    n ← n + 1
end while
while n < n_e + n_s do                             ▷ stream training phase
    k ← 0
    while k < k_0 do
        x ← d[k]
        y ← model_forward_stream(x)
        model_backward_a+e(y, model target)
        k ← k + 1
    end while
    n ← n + 1
end while
```

## IV. Results

### A. Evaluation

To measure the accuracy benefits of extended-range SC, we evaluate accuracy on image classification datasets CIFAR-10 and ImageNet [4]. For CIFAR-10, we use a 4-layer convolutional neural network (TinyConv) [16] and VGG-16 [1]. For ImageNet, we use Resnet-18 and Resnet-34 [2]. Models

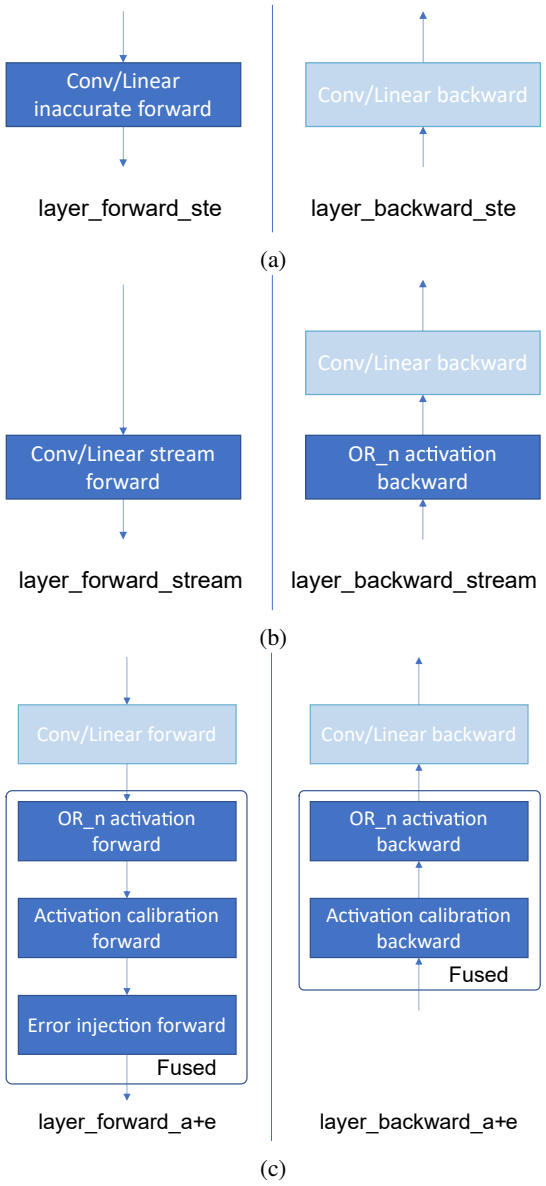



Fig. 13: Comparison between (a) vanilla STE [17], (b) stream with OR_n activation and (c) a+e training step.

are trained using PyTorch 1.12.0. For a+e, the error curves are adjusted 5 times per epoch. Lower frequency is possible but does not yield additional performance benefits as the calibration time is sufficiently amortized. CIFAR-10 models are trained for 100 epochs, where 80 epoch use a+e and 20 use stream simulation. ImageNet models are trained for 70 epochs, where 65 epochs use a+e and 5 use stream simulation. Models are trained using maximal-length LFSRs as the RNG and use the same RNG sharing scheme proposed in [10]. LFSR seeds are shared between different filters and between different input batches. The sharing scheme is demonstrated in Tab. VIII for a layer with 4x4 input and 3x3 weight and a stream length of 8 (3-bit LFSR). Seeds are also shared within the same input and weight filter due to the limited number of unique seeds for a maximal-length LFSR ($2^n - 1$ for an n-bit LFSR) and are correctly modeled. Streams are converted to fixed-point numbers using a counter between layers. The training code is available at https://github.com/nanocad-lab/rex-sc.

TABLE VIII: LFSR seed sharing scheme demonstration.

| Position | Seed values |
|---|---|
| Input 1 | 0,1,2,3,4,5,6,0,1,2,3,4,5,6,0,1 |
| Input 2 | 0,1,2,3,4,5,6,0,1,2,3,4,5,6,0,1 |
| Filter 1 | 6,0,1,2,3,4,5,6,0 |
| Filter 2 | 6,0,1,2,3,4,5,6,0 |

### *B. Accuracy improvements*

Accuracy comparisons between OR_n and other alternatives are shown in Fig. 14a for TinyConv [16] on CIFAR10. OR_1 uses OR gates for accumulation similar to the setup used in [3]. SC-Bin uses parallel counters for accumulation, similar to the setup in [11]. It can also be seen as an upper bound of accuracy for uSADD and uNSADD proposed in [13], as the latter two also trie to reduce multiplication error while sacrificing some accumulation accuracy in the pursuit of streaming compute. MUX-based additions suffer from scaling issues on the weight values, which prevents convergence even with the improvements from CeMux[15]. PB uses partial binary accumulation setup used in [10]. Accuracy of OR_2 is comparable to PB at both stream lengths, while OR_3 outperforms both. SC-Bin has a 2-4% point accuracy advantage over OR_2 and 1-3% point advantage over OR_3. Compared to FXP6 baseline using 6-bit fixed-point multiply-accumulate, OR_2 has a 4-6% point accuracy deficit, while OR_3 narrows the gap to 3-5% points. Fig. 14a also shows the results of the same concept applied to AND multiplication, denoted as "AND_2" and "AND_4". AND_n takes groups of n bits from each of the two multiplicands and multiplies all bits from the first group with all bits from the second group. Traditional AND multiplication can thus be seen as AND_1. AND_2 and AND_4 increase multiplication stream length by 2X and 4X respectively and increase overall area and energy by the roughly same amount. While AND_n can also improve accuracy, it is not energy- or area-efficient. AND_4 roughly matches OR_1 when using half the stream length, but consumes $\approx 2X$ the area and energy.

Accuracy results of VGG-16 on CIFAR-10 and Resnet-18/34 on ImageNet are shown in Fig. 14b) and 14c)/14d). Both OR_2 and OR_3 achieve similar accuracy to 6-bit fixed-point on VGG-16. While OR_2 and OR_3 improves Top-5 accuracy by 3-5% points compared to OR_1 on ImageNet, they are 2-5% points lower than FXP6. Accuracy of OR_N is likely limited by the training hyperparameters. For instance, doubling the number of epochs from 35 to 70 improves accuracies by 1.5-2% points for OR_n, and longer training times should improve accuracies even further. Compared SC-Bin that has unlimited accumulation precision, OR_2 and 3 reduces the accuracy gap while requiring at most 2 bits of accumulation precision. Compared to fixed-point computation and SC-Bin, OR_N enables scaling to higher performance levels. While SC-Bin using 16-bit streams has accuracy between 32-bit and 64-bit OR_3 on TinyConv, it is not a useful setup for the other models. Both fixed-point and SC-Bin have convergence problems in VGG and Resnets when dropping precision further,

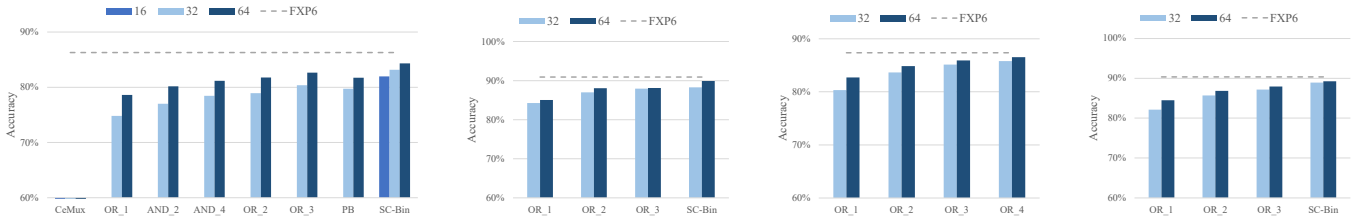



Fig. 14: Accuracy comparison for (a) TinyConv and (b) VGG-16 on CIFAR-10, (c) Resnet-18 and (d) Resnet-34 on ImageNet. 16-bit SC-Bin only has an accuracy result for TinyConv as it does not converge for other larger models.

TABLE IX: Array size, compute area and power, clock period, inference latency, energy, and energy improvement compared to 6-bit fixed-point, for different models and datasets with different types of SC accumulation.

| | | | | | | CIFAR-10 TinyConv | | | CIFAR-10 VGG | | | ImageNet ResNet-18 | | | ImageNet ResNet-34 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Architecture | N | Area [mm2] | Period [ns] | Power [mW] | Stream Length | Latency [us] | Energy [uJ] | E. Impr. vs FXP6 | Latency [us] | Energy [uJ] | E. Impr. vs FXP6 | Latency [us] | Energy [uJ] | E. Impr. vs FXP6 | Latency [us] | Energy [uJ] | E. Impr vs FXP6 |
| FXP6 | 9 | 2.97 | 4.5 | 692 | | 4.5 | 7.9 | 1.0 | 76.5 | 132.2 | 1.0 | 446.8 | 772.8 | 1.0 | 817.5 | 1413.2 | 1.0 |
| SC Bin | 41 | 3.12 | 1.6 | 3282 | 32 | 2.6 | 12.3 | 0.6 | 44.6 | 207.1 | 0.6 | 246.0 | 1145.4 | 0.7 | 449.8 | 1666.0 | 0.8 |
| | | | | | 64 | 5.3 | 21.0 | 0.4 | 89.2 | 353.5 | 0.4 | 492.1 | 1953.0 | 0.4 | 899.6 | 3142.4 | 0.4 |
| SC uSADD | 12 | 2.93 | 1.1 | 970 | 32 | 20.6 | 31.5 | 0.2 | 347.8 | 530.3 | 0.2 | 2032.3 | 3101.2 | 0.2 | 3718.2 | 4243.8 | 0.3 |
| | | | | | 64 | 41.3 | 51.6 | 0.2 | 695.7 | 867.7 | 0.2 | 4064.6 | 5072.5 | 0.2 | 7436.5 | 7850.5 | 0.2 |
| SC uNSADD | 12 | 2.95 | 1.2 | 1026 | 32 | 21.8 | 33.9 | 0.2 | 368.2 | 570.7 | 0.2 | 2151.3 | 3337.1 | 0.2 | 3936.0 | 4675.4 | 0.3 |
| | | | | | 64 | 43.7 | 56.3 | 0.1 | 736.4 | 948.5 | 0.1 | 4302.6 | 5544.4 | 0.1 | 7872.0 | 8713.7 | 0.2 |
| SC OR_1 | 64 | 3.21 | 2.4 | 488 | 32 | 1.5 | 1.0 | 8.0 | 27.4 | 17.8 | 7.4 | 155.9 | 101.4 | 7.6 | 285.5 | 267.2 | 5.3 |
| | | | | | 64 | 3.0 | 1.7 | 4.6 | 54.8 | 31.1 | 4.2 | 311.7 | 177.4 | 4.4 | 571.0 | 406.4 | 3.5 |
| SC OR_2 | 51 | 3.2 | 2.2 | 596 | 32 | 2.7 | 2.1 | 3.7 | 43.2 | 34.8 | 3.8 | 227.8 | 183.8 | 4.2 | 417.8 | 404.7 | 3.5 |
| | | | | | 64 | 5.3 | 3.7 | 2.1 | 86.4 | 60.5 | 2.2 | 455.6 | 319.6 | 2.4 | 835.5 | 653.6 | 2.2 |
| SC OR_3 | 50 | 3.12 | 2.2 | 650 | 32 | 2.6 | 2.3 | 3.4 | 42.4 | 37.3 | 3.5 | 231.6 | 204.5 | 3.8 | 424.1 | 437.0 | 3.2 |
| | | | | | 64 | 5.2 | 4.0 | 2.0 | 84.7 | 64.9 | 2.0 | 463.1 | 355.1 | 2.2 | 848.1 | 712.8 | 2.0 |

as weight values tend to underflow the smallest representable value with 5-bit fixed-range quantization (16-bit stream for SC).

TABLE X: Iteration time comparisons between OR_n and partial binary accumulation. Time is measured in s/256 images on an RTX 3090. The numbers before "PB" are the sizes of OR accumulation, with smaller numbers leading to more binary accumulations.

| | TinyConv (CIFAR-10) | | Resnet-18 (ImageNet) | | |
|---|---|---|---|---|---|
| Stream Length | 5x8 PB | OR_n | 5x8 PB | OR_n | a+e |
| 32 | 0.092 | 0.041 | 8.08 | 1.09 | 0.26 |
| 64 | 0.108 | 0.052 | 9.24 | 1.64 | 0.26 |
| 128 | 0.135 | 0.081 | 11.61 | 2.72 | 0.26 |

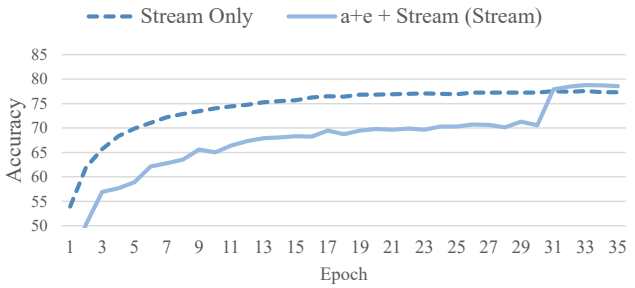



Fig. 15: Convergence behavior of stream only and a+e. The drop in accuracy at epoch 31 for a+e is when switching to using streams for the forward pass.

### C. Training speed improvements

While partial binary accumulation and OR_2/3 have similar accuracy benefits over OR_1 as shown in Section IV-B, partial binary accumulation complicates the training process of neural networks. In contrast to OR_2 and OR_3, it is not feasible to approximate partial binary accumulation with a single activation function, since partial binary accumulation is not associative by design.

Iteration speed comparisons between OR_n and partial binary accumulation are shown in Tab. X. Different n values for OR_n only changes the activation function used during training and does not significantly affect training time. For smaller models like TinyConv where the memory requirement is not an issue, PB is between 1.5X and 2.2X slower. Larger models like Resnet-18 suffer more due to the memory size limitations. Higher levels of binary accumulation exacerbate this effect, making PB up to 7.4X slower than OR_n to train. a+e further reduces iteration time by at least 4X. Since the runtime of a+e not dependant on stream length, it is especially useful when training for longer streams.

Fig. 15 compares the convergence behaviors of training with stream only and with a+e replacing most of the epochs. Training is limited to 35 epochs for both cases, as stream training with the full 70 epochs takes too long. a+e has similar convergence behavior as stream only, and achieves similar accuracy in the end. Overall training time is reduced by at least 3X with a+e compared to stream only and at least 22X compared to training with PB as shown in Tab. XI.

### D. Performance

To evaluate the performance benefits of using OR_n implementations, we use the previously synthesized adder and

TABLE XI: End-to-end training time comparison in hours between OR_n and partial binary accumulation. Results for OR_n and PB are estimated as they take prohibitively long to train.

| Stream Length | 5x8 PB | OR_n | a+e |
|---|---|---|---|
| 32 | 710.9 | 95.9 | 31.6 |
| 64 | 812.4 | 144.2 | 35.0 |

dot product results together with buffers and SNG results using a commercial 28nm technology and Cadence Genus synthesis tools. SNGs are based on maximum-length LFSRs, shared across different processing elements, and comparators. We use 6-bit fixed-point (FXP6) as a baseline, and compare its performance with SC implementations. Besides, SC with binary accumulation (SC Bin), OR_1, OR_2, and OR_3, we include uSADD and uNSADD (scaled and non-scaled adders) from [13]. Both uSADD and uNSADD use uMUL multipliers, which offer high accuracy without retraining, but at a significant hardware cost. To evaluate system-level performance, we assume dot-product processing elements are organized as a N-by-N GEMM array, similar to recently proposed SC accelerators [23], [13], [24]. We assume the individual PE to have a width of 256. We use a $N = 64$ array using OR-based SC dot products as a baseline, which occupies an area of $3.2mm^2$, and we size up all other configurations to be as close to this area budget as possible. We then try to normalize the throughput w.r.t. FXP6 for all configurations, assuming 64-long streams, by increasing or lowering clock frequency, and corresponding power to take advantage of voltage-frequency scaling. SC with binary accumulation and OR_1 accumulation cannot be fully throughput normalized to FXP6 due to impossibly high and low resulting voltage requirements, respectively. As a result of throughput normalization, OR_n configurations can use lower clock frequency, and consume lower power than other SC configurations with similar area. We use an analytical model that takes the array size and layer parameters, such as input and filter size, number of filters etc., and calculates both the number of compute iterations as well as memory accesses required to process a given layer, assuming output stationary dataflow. We focus on convolutional layers, as they dominate runtime and energy for all explored models. Our model assumes the memory bandwidth is provisioned such that computation is never stalled and that computation has maximum valid utilization. We use the clock period, stream length, and iteration count estimate the latency of each model's inference. We estimate inference energy using the latency, synthesized design power, memory access count, and energy obtained using the CACTI 6.5 tool [25].

Performance results including improvement over 6-bit fixed-point are shown in Tab. IX. First, we show that SC with parallel counter accumulation has 1.6-2.7X worse energy consumption than fixed-point. This is because the larger area and latency of parallel counter addition cannot compensate for the increased latency. The proposed OR_2 and OR_3 designs successfully bridge the performance gap between OR_1 and binary accumulation. OR_2 achieves up to 4.2X energy and 2X latency improvement over fixed-point at 32-long streams, and up to 2.4X at 64-long streams, with similar latency. OR_3 fares marginally worse, with up to 3.8X energy and 1.9X latency improvement at 32-long streams, and 2.2X improvement at 64-long streams. Both designs outperform SC with binary accumulation, even at twice the stream length. OR_2 and OR_3 with 64 long streams have up to 3.6X and 3.2X lower energy, respectively, than the parallel counter implementation with 32 long streams. They also outperform uSADD and uNSADD configurations at the same stream lengths, although it comes mainly from the high cost of accurate uMUL multipliers. With an AND-based multiplication, uSADD and uNSADD would be very similar to regular binary accumulation, as they are based on parallel counters.

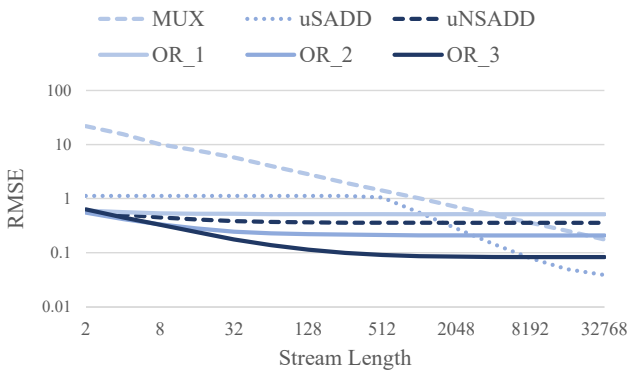



Fig. 16: Comparison of multiplexer- and OR_n-based adders' root mean squared error (RMSE) with respect to normal addition. We use sums of 1000 positive inputs with outputs having an average of 1 and a variance of 0.5.

### *E. OR_n for non-trained applications*

Most of the results focus on deep learning performance of OR_n with training involved. This is valid for neural networks that rely on training, and where the non-linearity of OR_n accumulation can be accounted for. In applications where training is not allowed, OR_1 already offers lower error compared to Multiplexers for relatively short streams as shown in Fig. 2. OR_2 and OR_3 extends this lead to even larger values, as shown in Fig. 16. MUX requires stream length $\geq 4096$ to have lower error than OR_1 before factoring training. For OR_2 and OR_3, that threshold becomes 32768 and 131072 respectively.

We also compared the error of OR_n accumulation with unary computing accumulators proposed in [13]. Despite not having the non-linearity of OR_n, the scaled addition version (uSADD) only beats OR_2 and OR_3 with stream length $\geq 4096$ and 8192 respectively. On the other hand, the non-scaled version (uNSADD) quickly saturates and is comparable to OR_1. Both versions are more expensive than SC Bin discussed previously. At stream lengths used for accuracy and performance evaluations (32 and 64), OR_2 and OR_3 offers 1.57x to 8.10x lower average error.

## V. CONCLUSION

In this work, we presented REX-SC - Range-Extended Stochastic Computing for neural network acceleration. REX-SC improves the accuracy of SC accumulation by increasing the number of output bits after accumulation. It reduces the accuracy gap between SC using OR accumulation and fixed-point computation. While SC with binary accumulation has almost no performance advantage over fixed-point, REX-SC remains competitive in performance with fixed-point neural networks while preserving the benefits of stochastic computing, including variable precision [3] and early termination[13]. REX-SC also improves the training performance of SC-based neural networks with optimizations to both the forward and backward propagation components. Our future work will focus on further improving accuracy of SC while maintaining its performance benefits.

## REFERENCES


[1] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014, arXiv: 1409.1556 ISBN: 0950-5849. [Online]. Available: http://arxiv.org/abs/1409.1556

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778, arXiv: 1512.03385 ISSN: 15737721.

[3] W. Romaszkan, T. Li, T. Melton, S. Pamarti, and P. Gupta, "ACOUSTIC : Accelerating Convolutional Neural Networks through Or-Unipolar Skipped Stochastic Computing," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 768–773.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Communications of the ACM*, 2012, pp. 1106–1114, arXiv: 1102.0183 ISSN: 10495258.

[5] M. Horowitz, "Computing's Energy Problem (And What We Can Do About It)," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, vol. 57, 2014, pp. 10–14, iSSN: 01936530.

[6] Y. H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016, arXiv: 1512.04295 ISBN: 978-1-4673-9466-6.

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

[8] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Transactions on Embedded computing systems (TECS)*, vol. 12, no. 2s, pp. 1–19, 2013.

[9] Z. Li, J. Li, A. Ren, R. Cai, C. Ding, X. Qian, J. Draper, B. Yuan, J. Tang, Q. Qiu, and Others, "HEIF: Highly Efficient Stochastic Computing based Inference Framework for Deep Neural Networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 8, pp. 1543–1556, 2018.

[10] T. Li, W. Romaszkan, S. Pamarti, and P. Gupta, "GEO : Generation and Execution Optimized Stochastic Computing Accelerator for Neural Networks," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1–6.

[11] H. Sim and J. Lee, "A New Stochastic Computing Multiplier with Application to Deep Convolutional Neural Networks," in *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC '17*, 2017, pp. 1–6, iSSN: 0738100X. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3061639.3062290

[12] R. Hojabr, K. Givaki, S. M. R. Tayaranian, P. Esfahanian, A. Khonsari, D. Rahmati, and M. H. Najafi, "SkippyNN : An Embedded Stochastic-Computing Accelerator for Convolutional Neural Networks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. ACM, 2019, pp. 1–6.

[13] D. Wu, J. Li, R. Yin, H. Hsiao, Y. Kim, and J. S. Miguel, "uGEMM : Unary Computing Architecture for GEMM Applications," *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 377–390, 2020, iSBN: 9781728146614.

[14] A. Ren, J. Li, Z. Li, C. Ding, X. Qian, Q. Qiu, B. Yuan, and Y. Wang, "SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 405–418, 2017, arXiv: 1611.05939 ISBN: 9781450344654. [Online]. Available: http://arxiv.org/abs/1611.05939

[15] T. J. Baker and J. P. Hayes, "Cemux: Maximizing the accuracy of stochastic mux adders and an application to filter design," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 3, jan 2022. [Online]. Available: https://doi.org/10.1145/3491213

[16] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," pp. 1–10, 2018, arXiv: 1801.06601. [Online]. Available: http://arxiv.org/abs/1801.06601

[17] Y. Bengio, "Estimating or propagating gradients through stochastic neurons," *CoRR*, vol. abs/1305.2982, 2013. [Online]. Available: http://arxiv.org/abs/1305.2982

[18] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 11.4," 2021. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[19] Intel, "Intel® intrinsics guide," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX,AVX2

[20] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[21] NVIDIA, "Programming guide :: Cuda toolkit documentation," 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma

[22] G. S. Watson, "Linear Least Squares Regression," *The Annals of Mathematical Statistics*, vol. 38, no. 6, pp. 1679 – 1699, 1967. [Online]. Available: https://doi.org/10.1214/aoms/1177698603

[23] V. K. Chippa, S. Venkataramani, K. Roy, and A. Raghunathan, "StoRM: A Stochastic Recognition and Mining processor," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, 2014, pp. 39–44, iSSN: 15334678.

[24] W. Romaszkan, T. Li, and P. Gupta, "SASCHA—Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4169–4180, Nov. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9852757/

[25] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0 : A Tool to Model Large Caches," *HP laboratories*, vol. 27, no. HPL-2009-85, p. 28, 2009.