SASCHA - Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration

Wojciech Romaszkan, *Student Member, IEEE,* Tianmu Li, *Student Member, IEEE,* Puneet Gupta, *Fellow, IEEE*

***Abstract*—Stochastic computing (SC) has recently emerged as a promising method for efficient machine learning acceleration. Its high compute density, affinity with dense linear algebra primitives, and approximation properties have an uncanny level of synergy with deep neural network computational requirements. However, there is a conspicuous lack of works trying to integrate SC hardware with sparsity awareness, which has brought significant performance improvements to conventional architectures. In this work, we identify why common sparsity-exploiting techniques are not easily applicable to SC accelerators and propose a new architecture - SASCHA - Sparsity-Aware Stochastic Computing Hardware Architecture for neural network acceleration that addresses those issues. SASCHA encompasses a set of techniques that make utilizing sparsity in inference practical for different types SC computation. At 90% weight sparsity, SASCHA can be up to 6.5X faster and 5.5X more energy-efficient than comparable dense SC accelerators with a similar area without sacrificing the dense network throughput. SASCHA also outperforms sparse fixed-point accelerators by up to 4X in terms of latency. To the best of our knowledge, SASCHA is the first stochastic computing accelerator architecture oriented around sparsity.**

***Index Terms*—Accelerators, Machine Learning, Stochastic Computing**

I. Introduction

New classes of machine learning mobile applications, like virtual assistants, translation, and image recognition, continue to emerge, amplifying the demand for fast, efficient, and secure inference [23], [46], [47]. Increasingly, this demand cannot be satisfied by offline, cloud-based processing due to substantial and unpredictable network latencies as well as privacy concerns [23], [51]. To enable online machine learning, mobile devices increasingly incorporate custom accelerators, broadly known as neural processing units, or NPUs [47]. Those devices are deployed under strict area, power, and energy constraints.

To improve the throughput and energy efficiency, researchers have increasingly looked into model compression methods, like quantization and pruning [6], [7], [49], [50]. Non-conventional computing methods, like in-memory or stochastic computation, have also been gaining popularity [2], [12], [26], [38]. Hardware support for some of those techniques has already made its way into commercially available devices [9].

Stochastic computing (SC) in particular has been shown as a very promising approach to approximate computing acceleration, particularly for dense, compute-heavy models like convolutional neural networks [28], [29], [38], [48]. Recent works have shown significant improvements in both the accuracy of SC computation as well as the area and energy efficiency of the arithmetic units [20], [28], [48]. There is now a plethora of possible SC *flavors*, spanning the accuracy-efficiency Pareto curve, depending on application requirements. However, there is a conspicuous lack of SC architectures trying to take advantage of neural networks' resilience to pruning [49], something that could enable further performance improvements. To address this gap, we propose SASCHA - Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration. SASCHA consists of computational units design, accelerator architecture, and a scheduling method that improves the efficiency of executing sparse neural networks using SC computation without sacrificing high parallelism and data re-use opportunities.

The key contributions of this work are:

- We introduce the multi-group, parallel stream sparse SC SASCHA processing element (PE), agnostic of underlying SC computation style, and perform a thorough evaluation of its extensive design space.
- To the best of our knowledge, we propose the first stochastic computing neural network accelerator architecture that takes advantage of parameter sparsity. SASCHA can achieve up to 6.5X throughput and 5.5X energy efficiency improvement at 90% sparsity level, compared to a dense SC accelerator with a similar area, while maintaining the throughput and suffering only up to 31% energy efficiency in the dense case.
- We propose a weight bit-slicing technique using asymmetric streams unique to SC that can extract weight sparsity even in dense networks, improving SASCHA throughput and energy-efficiency by up to 1.75X on unpruned networks.

The rest of this paper is organized as follows: Section II provides a brief introduction to stochastic computing and neural network pruning. Section III explains why conventional approaches to exploiting sparsity are not easily applicable or beneficial in the case of SC accelerators. Sections IV introduces the sparse SASCHA PE and explores its design space. Section V describes the architecture of the SASCHA accelerator and its asynchronous scheduler. Section VI discusses a bit-slicing method that can extract high effective sparsity in unpruned networks. Section VII shows the benefits of the SASCHA accelerator. Finally, Section VIII summarizes related work.

II. Background

A. Stochastic Computing

Stochastic computing is a number representation using a proportion of 1's in a binary stream to represent fractional numbers [13]. Compared to conventional fixed- or floating-point formats, SC makes it possible to implement certain arithmetic operations, like multiplication and addition, using single logic gates. For example, two values can be multiplied by passing their stochastic streams through an AND gate. However, the compact arithmetic offered by SC comes at the cost

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented in the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2022 and appears as part of the ESWEEK-TCAD special issue.

The authors are with the Electrical Computer Engineering Department, University of California, Los Angeles, Los Angeles, CA 90095, USA (e-mail: wromaszkan@ucla.edu, litianmu1995@ucla.edu, puneetg@ucla.edu

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-27867. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advance Research Projects Agency (DARPA) or the U.S. Government.

of random errors and precision issues, which make it unsuitable for certain applications [2]. Fortunately, neural networks' error-tolerant nature, as well as heavy reliance on linear algebra kernels and multiply-accumulate (MAC) operations, make them perfect candidates for SC acceleration [20], [38], [48]. Because of that, SC has been enjoying a renaissance, with many different flavors of computation proposed, spanning various points on the accuracy-efficiency spectrum - some of them maximizing the density and efficiency, while others maintaining the accuracy close to fixed-point designs [20], [28], [29], [38], [48]. Below we briefly outline the most relevant components of SC.

Stochastic computing offers two alternative number representation formats: unipolar and bipolar. In the former, a value in the range of $[0,1)$ is represented as the proportion of 1's in a binary stream of arbitrary length, as shown in Figure 1 a). The latter, shown in Figure 1 b), represents a number in the range of $(-1,1)$ using the difference between the number of 1's and 0's in the stream. Bipolar representation has twice the effective range of the unipolar, and therefore lower precision when using the same stream length. To maintain the higher accuracy of unipolar streams as well as the ability to represent both positive and negative values, various techniques have been proposed [20], [38], [48]. In this work, we use the split-unipolar implementation, where the negative and positive multiplication results are accumulated separately using replicated adder trees, and partial sums are subtracted in the fixed-point domain [38].

One of the main considerations of SC architectures is the conversion cost. To enable efficient, single-gate stream processing, values stored in integer format must be converted into stochastic streams. Conversion is achieved using the *Stochastic Number Generator* (SNG) circuit. It most commonly consists of a random number generator (RNG), a fixed-point value buffer, and a comparator, as shown in Figure 1 c). The implementation of this circuit can be very costly, depending on the type of RNG used. Previous works have used either linear-feedback shift registers (LFSRs), or low-discrepancy (LD) sequence generators, such as Halton or Sobol [28], [48]. Conversion from stochastic streams back to fixed-point values is achieved by counting the number of 1's in the output stream. Due to high costs, SC architectures often try to amortize conversion by using high spatial reuse [38], [48]. Figure 1 also shows an example of unipolar multiplication using an AND gate multiplier, like the one used in [20], [38]. Additions can be performed using multiplexers [37], binary adders [20], [37], [48] or OR gates [38].

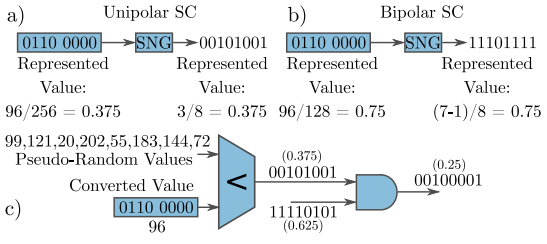

Fig. 1. Examples of unipolar (a) and bipolar (b) SC representation, and a unipolar Stochastic Number Generator Circuit (SNG) with an AND gate SC multiplication (c).

B. Sparse Neural Networks

Pruning superfluous weights in neural networks has been extensively explored as a way of reducing the model size and latency, as well as improving generalization [16], [17]. Potential performance benefits come from the fact that multiplications involving zero-valued weights do not change final results, hence the operation itself, as well as associated memory accesses can, in theory, be skipped. If the hardware can take advantage of it, storage, runtime, and energy consumption can be reduced, at some cost to network accuracy. Previous works have demonstrated that weights can be pruned by up to 90% without increasing model error [16]. Unfortunately, the hardware commonly used to accelerate deep learning execution, particularly GPUs, is frequently optimized toward dense linear algebra kernels and cannot deal efficiently with *unconstrained* sparsity. Because of that, various forms of *constrained*, or structured, pruning have been introduced, including filter-wise sparsity [18], group-wise sparsity [49], and pattern sparsity [31]. While structured sparsity can be efficiently executed on parallel architectures such as GPUs, constraining pruning flexibility in training can have a large impact on network accuracy, counteracting the generalization benefits mentioned above [49]. Because of that, structurally pruned networks generally achieve lower overall sparsity compared with unstructured ones with the same accuracy [49]. Due to the very high potential benefits of pruning, particularly unconstrained, multiple custom sparse accelerator architectures have been proposed in recent years [11], [14], [19], [30], [35], [50].

III. Motivation

Given the recent popularity of stochastic computing and sparsity-aware accelerators, there is a surprising lack of attempts to combine both approaches. This section explains why taking advantage of sparsity in stochastic computing hardware cannot be tackled by the same techniques as conventional, floating- or fixed-point accelerators.

Most common attempts to exploit sparsity in hardware rely on matching non-zero input and weight values that need to be multiplied together to avoid ineffectual computations, i.e. ones where at least one operand is zero [11], [14], [15], [19], [25], [30], [35], [36], [50]. To avoid an issue where no non-zero operands are available, causing stalls and poor utilization, larger staging buffers that can spatially or temporarily *advance* operands are frequently used [15], [25], [30], [36]. While pre-trained weights can often be scheduled offline, simplifying the hardware, exploiting input sparsity must be performed dynamically, incurring non-negligible hardware overheads. For example, [25] increases the area of the compute core by 2.8-5.9x to support sparsity compared to the dense baseline. In other approaches, matching can be achieved by calculating an intersection operation between input and weight values on an output-by-output basis [14], [19], [35], or coupling custom dataflows with sparse storage format choices [34], [43]. Those techniques can also incur significant overheads. For example, in [14], the intersection calculation module is more than 10x larger than the compute. In [35], a large crossbar network is required to route the outputs, exceeding the size of compute units by more than 3X. Similarly, in [43], more than 95% of the processing element area is consumed by the sorting queues.

For devices operating with conventional floating- or fixed-point values, high area, and energy cost of computation can justify such overheads. However, that is not the case in SC-based architectures. As mentioned before, conversion circuitry, including SNGs, RNGs, and buffers, is frequently the dominant area and power contributor in SC accelerators. Analyzing the accelerator area breakdowns in [28], we can see that the SC MAC arrays (multipliers and adders) occupy as little as 6% of the overall area, while the conversion circuitry consumes 51%, with similar energy contributions. While the techniques mentioned above could be applied to SNGs, the high spatial data reuse introduces an additional level of complexity [20], [29], [38]. Commonly, thousands (or more) of multiply-accumulate operations can be scheduled concurrently, with individual operand streams being broadcast across many multiplications in parallel. Therefore, a given operand could only be skipped if all corresponding operands it is meant to be concurrently multiplied with are zeros. Such extensive

reuse would limit ineffectual computation skipping opportunities and dramatically increase the cost of already expensive dynamic intersection calculation, dwarfing low-cost SC computation. Further, conventional sparse accelerators require complex sparsity detection logic to improve the utilization of the limited number of their large and complex processing elements [25]. SC's high parallelism and low cost make underutilization less of a problem compared to architectures with a limited number of large floating- or fixed-point PEs [38]. In short, for SC architectures, *detecting sparsity is more costly than ignoring it.*

Taking advantage of sparsity in stochastic computing architectures by using conventional approaches can therefore yield minimal benefits or could end up being detrimental. To this end, we make a few guiding observations. First, any attempt at exploiting sparsity should not compromise the high level of parallelism enabled by SC [38]. Therefore any approaches requiring fine-granularity, dynamic scheduling, or other sources of hardware overheads are highly undesirable. Particularly, individual dynamic intersections between both sparse weights and activations are not compatible with spatial data reuse employed by SC. To this end, whenever possible, we want the burden of exploiting sparsity to lie on the offline, static side, so as not to introduce unnecessary hardware overheads. Because of that, we focus only on the sparsity of weights, which are static and known a priori, while keeping activations dense. Second, explicitly avoiding the ineffectual SC computations, as opposed to floating- or fixed-point ones has limited benefits and should not be the goal in itself. Instead, we believe sparsity should be used to improve the dominant area and energy contributors in SC-based architectures: memory and SNGs [28], [38].

IV. SASCHA Sparse SC PE

A. Sparse PE Design Objectives

In this section, we outline the design of a sparse SC processing element, implementing a parallel dot product operation. Our PE is agnostic of the underlying style of SC computation. To demonstrate that, we evaluate three different implementations: split-unipolar AND-based multiplication with partial-binary OR-based accumulation used by GEO [28], modified GEO-style PE with full binary accumulation, and the uMUL multiplier with binary accumulation proposed by uGEMM [48]. We will refer to them as GEO-, GEO+- and uGEMM-style PEs, respectively. Those PEs offer progressively higher precision at an increased area/power cost, as discussed in Section VII. While full-binary accumulation refers to adding all individual bits of SC multiplication results, preserving their full fidelity, partial-binary refers to performing the first part of accumulation using streaming, OR-based adders for more compact area, and the rest using binary accumulation [28]. Alternative SC computational components are fundamentally compatible with SASCHA, but they are beyond the scope of this work.

Our goal is to design a sparse SC compute unit, given SC hardware peculiarities outlined in Section III. First, as explained in Section III, our prime target for optimization is the cost of converting the numbers between fixed-point and stochastic representations. Second, we need to make a choice whether we should target weight or input sparsity. As explained in Section III, we want to avoid any dynamic approaches that incur high hardware overheads relative to cheap SC compute. Because of that, we avoid trying to exploit both weight, and activation sparsity due to costly intersection calculations [11], [14], [35]. Even exploiting just activation sparsity would require spatial and temporal operand advancement [15], [30] performed dynamically in hardware. Instead, we focus solely on the sparsity of network weights, which is known a priori for inference accelerators and enables static, offline scheduling. It allows us to exploit sparsity with minimal hardware changes that do not compromise SC density and high spatial reuse.

B. G:C Sparse PE

Similar to [9], we exploit structural sparsity in parameters. We believe this simple approach is well suited to our requirements as a) it can be statically scheduled, b) allows us to reduce the overhead of stochastic number generation, c) does not exploit or make any assumptions about input sparsity, and d) does not require high multiplexing overheads, as we will show shortly. However, we make three important distinctions between the sparse compute units of [9] and our Sparse SC PE. First, we explore other sparsity structures - in general, for every group of G parameters, we allow C non-zero ones. We will henceforth be referring to G as a *group size*, and C as *capacity*. Second, where [9] can only accelerate networks pruned to their exact sparsity structure, we design a scheduler capable of mapping networks with arbitrary levels and structure of sparsity onto the compute fabric built using sparse PEs. Finally, we focus specifically on stochastic computing, which enables different design trade-offs compared to conventional compute.

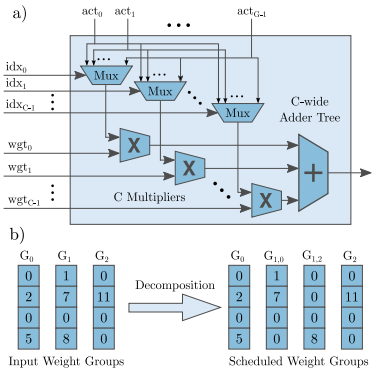

Fig. 2. Sparse PE with group size G and capacity C (a). Decomposing 3 arbitrary parameter groups of size $G=4$, into groups satisfying the capacity requirement of $C\leq 2$ (b).

A block diagram of a generic, i.e., supporting any format of underlying computation, sparse PE with a group size G and capacity C is shown in Figure 2 a). It performs a spatially parallel dot product operation between a dense vector of G activations and a sparse vector of C weights and their corresponding indices. A weight's index indicates its position in the dense vector and is used to select an activation that needs to be multiplied by the weight's value. Compared to an equivalent dense PE, it requires $G-C$ fewer multipliers and adders, at the cost of C additional $G:1$ multiplexers.

Using this PE is only possible when there is a guarantee that every group of G weights contains only up to C non-zero ones. We refer to such groups as *balanced*. When the balancing is enforced on the network parameters, as is the case with [9], with $G=4$ and $C=2$, the computation can be scheduled in the same way as on hardware using dense PEs while reducing storage and ineffectual operations. To schedule a network with an arbitrary level and structure of sparsity, we can decompose any weight vector of size G to between 1 and $\lceil G/C \rceil$ vectors. This is shown schematically in Figure 2 b), where 3 groups of size $G=4$ get decomposed into 4 balanced groups, each satisfying $C\leq 2$. The decomposed groups can then be scheduled on a sparse PE with $G=4$ and $C=2$. However, this computation will require 33% longer runtime compared to a dense processing element. We refer to the ratio of the number of decomposed to original weight groups as the *iteration overhead*. The maximum iteration overhead for a given sparse PE configuration is $\lceil G/C \rceil$. It is one of the main metrics for evaluating the efficiency of different sparse PE configurations. For simplicity, we restrict both G and C to be powers of 2.

A stochastic computing equivalent using GEO-style PE [28] is shown in Figure 3 a). The main concern in the SC case is the

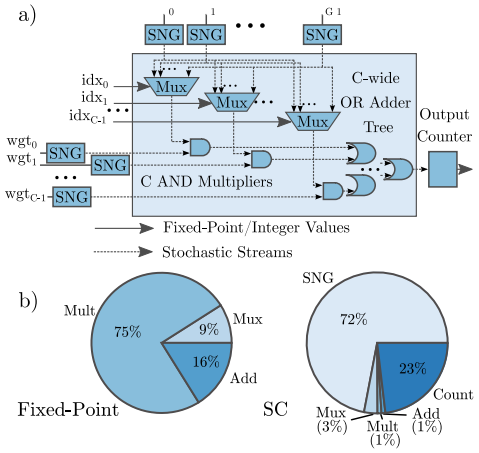

Fig. 3. Sparse GEO-style SC PE with group size G and capacity C (a). Split-Unipolar [38] logic is omitted case for readability. Area breakdown of fixed-point (left) and GEO SC (right) sparse PEs with G=4, and C=2 (b).

overhead of additional conversion circuitry. We have implemented and synthesized a set of sparse SC PEs using a commercial TSMC 28nm library and Cadence Genus synthesis tool to evaluate this. Figure 3 b) shows the breakdown of a fixed-point and GEO SC sparse PE with $G=4$ and $C=2$, excluding input and output buffers. While the fixed-point PE is dominated by the area of multipliers, in the SC one, the arithmetic occupies only about 2% of the area. While the cost of conversion can be amortized through input broadcasting and wider dot-products, it presents us with different optimization priorities compared to a fixed-point sparse PE. We have also compared the area of sparse SC PEs with different G and C, shown in Figure 4, for different styles of SC. For GEO-style PEs with $C<8$, the area of the sparse SC processing element is roughly equivalent to half of the dense one, even for large group sizes. This reduction is because conversion circuitry dominates the overall area, and the sparse SC PE structure eliminates roughly half of the overall SNGs when C is small. uGEMM sparse PE shows much higher area reduction compared to dense, due to the fact that weight buffers and SNGs are bundled together with the multiplier, resulting in a larger size of a dense PE. The close coupling of stream generation and computation is done for decorrelation purposes, resulting in higher multiplication precision [48]. If the sparsity structure can be enforced, this area reduction shows great potential for synergy between SC and sparse neural networks.

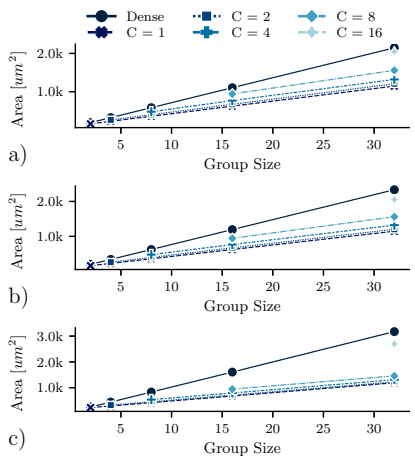

Fig. 4. Area of sparse and dense SC PEs, given different group sizes and capacities, for GEO (a), GEO with full binary accumulation (b), and uGEMM (c) style SC.

C. Multi-Group Sparse SC PE

Given the proposed sparse SC PE design, we need to choose the best G and C for a sparse SC accelerator. There are three main considerations here. First, as mentioned above, is minimizing the iteration overhead. We explore it in detail in Section V. The second is maximizing the hardware efficiency and amortizing SC conversion costs. SC accelerators often employ highly parallel dot product units, ranging from 16 to hundreds of concurrent MAC operations [28], [29], [38], [48]. High parallelism allows them to amortize the cost of converting streams corresponding to partial sums back to fixed-point representation. From this standpoint, using sparse PEs with large group sizes like 16 or 32 is desirable. However, this is where the third consideration comes into play - storage compression efficiency.

While we focused our discussion on sparse computation until now, another benefit of sparsity is reducing required storage - one of the main contributors to the area and energy consumption of neural network accelerators [4], [5], [50], including SC ones, [28], [38]. Sparse accelerators often employ compressed memory formats like compressed sparse row (CSR) to reduce storage and throughput requirements [19], [30], [36], [43]. In this work, we explore a simple compression scheme, where each weight is coupled with an index indicating its position in the group. The group's relative position in a given filter is then handled by the scheduler as described in Section V. While more efficient compression schemes may be available, they are beyond the scope of this work.

The index size is determined by the group size G and is equal to $log_2(G)$. Since indices are required on a per-value basis, they are independent of C. Larger group sizes will therefore incur higher indexing overheads. Figure 5 shows storage compression, the ratio between the memory required for storing all dense weights and storing only the sparse weights and their indices. It is an ideal case, where we assume only the sparse weights are stored, and there are no additional overheads, for example, coming from alignment requirements. Using a group size of 64 requires 1.44X more storage than a group size of 2. At a sparsity level of 90%, this translates to 5.7X and 8.9X compression for $G=64$ and $G=2$, respectively. More importantly, when running a dense network, going from a group size of 2 to 64 reduces compression from 0.89X to 0.57X. We want SASCHA to be flexible and support networks with different sparsity levels with high efficiency, even in the dense case. Because of that, large group sizes are highly undesirable.

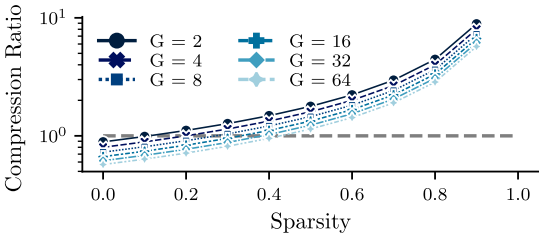

Fig. 5. Ideal ratio of dense to sparse storage cost for different PE group sizes, and sparsity levels. Gray line shows the break-even point between sparse and dense storage.

However, there is a way of implementing wide SC dot products while maintaining better compression ratios enabled by smaller group sizes. Up until now, we have only considered dot products consisting of a single group, referred to as *single-group* sparse SC PEs. Alternatively, we can construct a wide dot product using multiple smaller PEs. For example, a dot product of width $K=32$ can be constructed using $L=4$ PEs with $G=8$ or eight with $G=4$. We refer to those as *multi-group* sparse SC PEs, where the number of groups L is equal to K/G. An example of single- and throughput-equivalent multi-group sparse processing element with L groups is shown

on Figure 6 a) and b) respectively. They are not strictly equivalent because the sparsity structure required for the multi-group PE is more restrictive - the capacity is now uniformly distributed among individual groups instead of the whole dot product width.

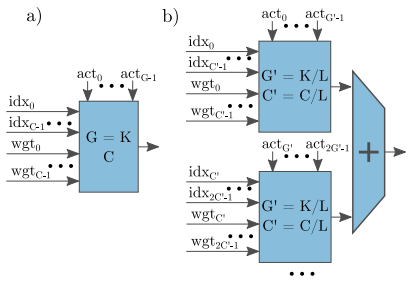

Fig. 6. Single-group sparse PE with a group size G, capacity C and dot product width K (a), and a throughput-equivalent multi-group sparse PE with L groups.

D. SASCHA PE Analytical Model

While the sparse SC PE is expected to be more area and energy-efficient than the dense one when weight groups are balanced, i.e., highly sparse, the opposite will happen when running a dense network. In the worst-case scenario, iteration overhead will cause a $\lceil G/C \rceil$ times longer runtime when the same number of dense and sparse PEs is used. To evaluate potential benefits at different sparsity levels, we develop a simple analytical SASCHA PE iteration overhead model. For simplicity, we assume that sparsity is uniformly distributed among weights. The iteration overhead of a multi-group sparse PE of size K, with $L = K/G$ groups, will be determined by the group with the largest number of non-zero weights. Therefore we want to find out the expected maximum number of non-zero values in a group of size G, across L groups. For a group of size G, the probability of having O non-zero weights given sparsity S is:

$$P_{NZ=O} = \binom{G}{O}(1-S)^O S^{G-O} \tag{1}$$

Where $S \in \langle 0,1 \rangle$ indicates the ratio of zero weights to all weights. Probability that across L groups of size G, one or more have O non-zero values, and non have more than O:

$$P_{LGO} = \sum_{i=1}^{L} \binom{L}{i} P_{NZ=O}^{i} P_{NZ<O}^{L-i} \tag{2}$$

Where $P_{NZ<O}$ is the probability that a group of size G has fewer than O non-zero values:

$$P_{NZ<O} = \sum_{i=1}^{O-1} \binom{G}{i}(1-S)^i S^{G-i} \tag{3}$$

The average maximum number of non-zero values A across L groups of size G is therefore:

$$A = \sum_{i=0}^{G} i P_{LGi} \tag{4}$$

And the expected iteration overhead I, given group capacity C is:

$$I = \frac{A}{C} \tag{5}$$

In our model, we assume that dot products where all weights are zero can be skipped entirely in the sparse PE example, as explained in Section V. This allows the iteration overhead to become less than 1. We used the above model to estimate iteration overheads of a multi-group sparse SC PE of width $K = 16$, at different group sizes, capacities, and sparsity levels. Results, normalized by the area, are shown in Figure 7 a), with a reference line showing the latency break-even point compared to a dense PE with the same K. It shows that configurations with larger group sizes are not as efficient at low sparsity levels but much better on highly sparse networks. For example, while sparse PE with $G = 8$ and $C = 1$ is on average 22% slower than the one with $G = 2$ and $C = 1$ at sparsity below 40%, it is on average 63% faster at higher sparsity levels. Further, increasing the capacity is an efficient way of improving the throughput: PE with a group size of 8 and capacity of 4 is on average 32% faster than the one with the capacity of 1 when normalized to the area. Based on those results, we will opt for larger group sizes, e.g., 4 and 8, compared to smaller ones.

Figure 7 b) compares the iteration overhead obtained through the model with the one obtained using an ideal scheduler described in Section V on the CIFAR-10 TinyConv network, for a PE with $G = 4$. Our model achieves a 0.996 correlation with the scheduler results, which justifies our choice of modeling weight sparsity using a uniform distribution. While our design could be optimized better towards forms of structured pruning, we want SASCHA to be flexible enough to handle any form of unstructured sparsity without putting the burden on machine learning researchers to conform to the underlying hardware. Using capacity > 1 seems like an obvious choice given its area-throughput benefits. This might explain the configuration chosen by [9], since the above analytical model is also applicable to fixed-point PEs. However, we will now discuss an alternative way of improving the throughput of sparse PEs that is unique to SC and enables yet another design axis to explore.

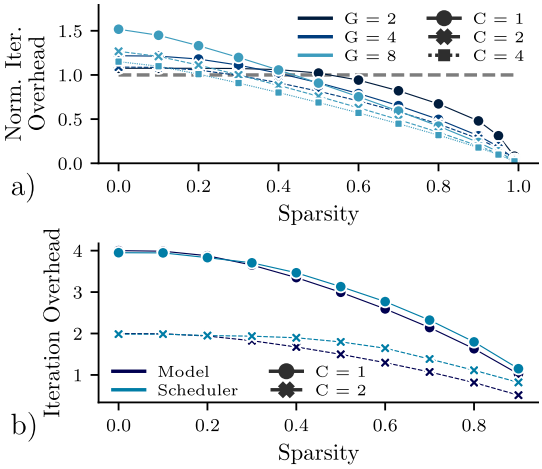

Fig. 7. Multi-group $K = 16$ sparse SC PE iteration overheads normalized to dense PE area (GEO-style), for different group size G and capacity C, at different sparsity levels, estimated using the analytical model (a). Gray line shows the latency break-even point with a dense PE. Iteration overhead difference between the model and an ideal scheduler described in Section V on the CIFAR-10 TinyConv network, for a PE with $G = 4$ (b).

E. Parallel Stream Processing

As Figure 4 shows, sparse SC PEs can be as much as 2.7X smaller than dense ones. The straightforward way of using this area advantage to increase the throughput is by packing more PEs in the same area. Unfortunately, in the case of sparse SC computation, this approach would yield only limited improvement. As shown in Figure 7, depending on the group size and capacity, iteration overheads can be as high as 4 or 8 times. Doubling, or tripling, the number of PEs would not be enough to compensate for it. Another way, as shown in the previous subsection, is to increase the capacity, which shows a good area-throughput trade-off.

However, stochastic computing provides us with another option of increasing the computation throughput. Until now, we assumed

that stochastic streams are processed sequentially - one bit at a time. However, by using multiple SNGs, multipliers, and adders per weight, the computation can be parallelized by a varying degree, cutting down the stream processing time and improving throughput [8], [39]. An example of a sparse SC PE with group size G, capacity $C=1$ and $P=2$ parallel streams is shown in Figure 8. Stream parallelism factor P can be varied to improve the throughput at the cost of additional area. This area-throughput trade-off space is unique to SC and not applicable to conventional fixed- and floating-point architectures, except for bit-serial ones [22], [25].

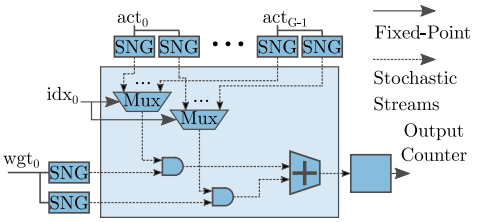

Fig. 8. Sparse SC PE with group size G, capacity $C=1$, and $P=2$ parallel streams. Split-unipolar accumulation fabric is omitted for readability.

If it is possible to apply parallel stream processing to sparse SC PEs to improve their throughput, the same technique could be applied to dense ones. However, the cost of increasing the stream parallelism is much higher for the dense PEs. Figure 9 shows the area of a 32x32 array of $K=32$ PEs, for dense and sparse PEs with different group sizes. Capacity is fixed at 1. Buffers, SNGs, and output counters are included. The area of the dense compute grows at a much faster rate with P than the sparse ones because the dense implementation requires many more SNGs, LFSRs, and compute units than sparse. For example, GEO-style array with $G=2$, $C=1$, and $P=2$ is only 5% larger than the dense one with $P=1$, while providing the same throughput in the dense case. Dense implementation with parallel streams scales particularly poorly for the uGEMM-style implementation where each parallel stream path requires a local decorrelating SNG. Because of that sparse uGEMM-style arrays are significantly smaller, up to 9.5X.

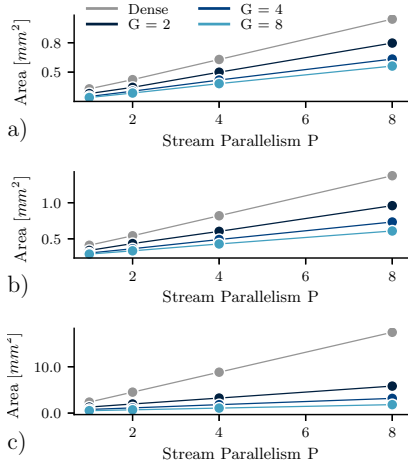

Fig. 9. Total area of a 32x32 array of $K=32$ GEO (a), GEO+ (b) and uGEMM (c) PEs, dense and sparse, with different stream parallelism factors. $C=1$ for all sparse configurations.

From now on, we will refer to the multi-group, parallel-stream sparse SC PE as *SASCHA PE*. SASCHA PE can be uniquely identified by a set of parameters K,G,C,P, where K is the dot product size, G is the group size, C is the capacity, and P is the stream parallelism factor. We will restrict our evaluation to group sizes of 8 and smaller, which guarantees high storage compression ratios. To further restrict the design space, we will use SASCHA PEs with $PC/G=1$.

V. SASCHA Architecture

A. SASCHA Accelerator

SASCHA accelerator block diagram is shown in Figure 10. It uses a highly parallel PE array similar to [48] and [8]. It relies on broadcasting and spatial data re-use, where all PEs in the same row share the same set of weights, and all PEs in the same column share the same set of activations. As explained in [48], this structure is uniquely suited for SC computation given small PE sizes and low wire congestion, as opposed to conventional fixed- and floating-point computation. The major distinction in our architecture is the use of SASCHA PEs, instead of dense processing units, using a sparse weight storage format and additional indexing and circuitry required to support asynchronous scheduling as described below. We refer to the number of rows as M, the number of columns as N, and the dot product width as K. Each of the M rows operates on a set of CK/G weights (WGT), their corresponding indices (IDX_W), and the parent filter index (IDX_F). The latter is needed to support asynchronous scheduling as described in Section V-B. Apart from SASCHA PEs, it contains a weight memory and PCK/G SNGs for parallel stream generation and merger units required for asynchronous scheduling. All N columns share the activation memory, which is organized as a ping-pong buffer [10] to facilitate simultaneous input reads and output writes. Similarly to [28], we use a near-memory vector unit to handle additional partial sum accumulation, batch normalization, scaling, and activation functions. We assume the use of ReLU activation, but the vector unit could be modified to support different ones.

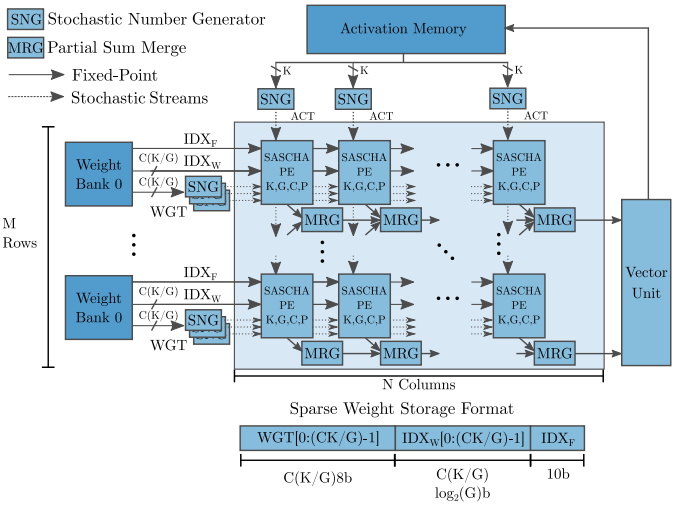

Fig. 10. SASCHA accelerator architecture block diagram. Partial sum output connections were omitted for readability.

The architecture in [48] focuses on accelerating general matrix multiplication (GEMM) operations, which can be used to implement both fully-connected and convolutional layers in neural networks. Since those two types of layers frequently consume $> 90\%$ of inference runtime [21], optimizing the efficiency of GEMM computation is highly desirable. However, while fully-connected layers yield themselves to GEMM representations naturally, convolutional layers need to be transformed. There are two common ways of doing that: image to column (im2col) and kernel to row (kn2row) [45]. In most cases, the latter is desirable, as it does not result in the input replication required by im2col. On the other hand, kn2row can result in compute underutilization on layers with a small number of input channels, Z [45]. For our SASCHA architecture and scheduler, we opt to implement layers that satisfy $Z<K$ using im2col

to maintain high utilization, while layers that satisfy $Z \geq K$ using kn2row. In both scenarios, filters are partitioned into *partial filters* whose sizes match the dot product width K. Partial filters that come from different *parent filters*, but correspond to the same spatial extents, can be scheduled concurrently in multiple rows of the SASCHA array as they can be multiplied with the same sets of inputs, producing partial sums corresponding to the same row and column in the output tensor.

B. SASCHA Asynchronous Scheduler

We now discuss the strategy for scheduling computation in a SASCHA architecture defined by M, N, K, G, C, P parameters. Naively, after performing im2col or kn2row unrolling, we can assign each partial filter to a specific row in the array and co-schedule M partial filters at a time. For the dense architecture, an example schedule of five partial filters, each with $K=4$, using an architecture with $M=4$ rows, is shown in Figure 11 a). We refer to this as *dense synchronous scheduling* since the execution of each group of M partial filters has to be synchronized. However, in the SASCHA case, as shown in Figure 2 c), depending on the level and structure of sparsity, as well as K,G, and C parameters, a given partial filter can be decomposed into a different number of balanced groups. If multiple partial filters are scheduled synchronously, their overall execution time will be constrained by the one with the lowest sparsity, as shown in Figure 11 b), for $K=G=4$, and $C=1$, referred to as *sparse synchronous scheduling*. In this toy example, synchronous scheduling leads to $I=2$ iteration overhead (assuming $P=1$) and 50% compute underutilization.

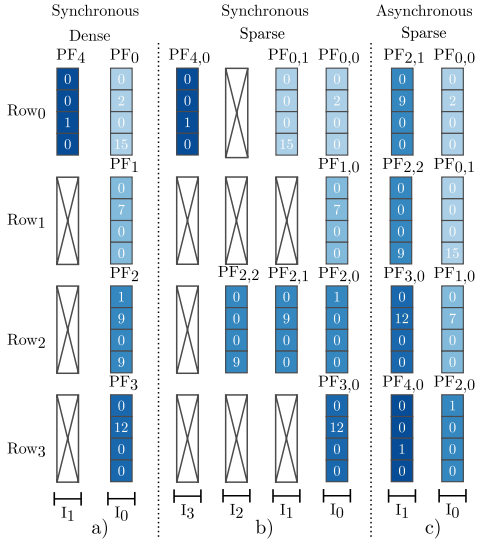

Fig. 11. Three schedules of 5 partial filters, with $K=G=4$ and $C=1$, on an architecture with $M=4$ rows: dense synchronous (a), sparse synchronous (b) and sparse asynchronous (c). Crossed out boxed indicate compute underutilization.

To improve scheduling efficiency, we propose the SASCHA *sparse asynchronous scheduling*. In essence, while the sparse synchronous approach operates on partial filters before decomposition into balanced groups, the SASCHA asynchronous scheduler works with individual balanced groups after decomposition. For all partial filters that correspond to the same spatial subset of original filters, their decomposed balanced groups are combined into a single list. The list elements are then sequentially scheduled onto available rows while keeping track of which partial filter they belonged to initially. The resulting SASCHA asynchronous schedule for the same set of partial filters as before is shown in Figure 11, resulting in the same iteration count as the dense schedule and 100% utilization. Using the asynchronous scheduler comes at the cost of additional storage. Each balanced group now needs to carry information about which parent filter it belongs to so that it can be written to the correct location in memory. However, we estimate this penalty to be modest - the worst-case overhead, given $K=32$ and $G=8$, would be 30%, while for $G=2$, it would be below 10%. The resulting compressed weight storage format is shown in Figure 10. Weight bank word size will depend on G,K,C parameters, but since those are dictated by PE configuration and fixed for a given SASCHA implementation, weight memory width can be explicitly provisioned for it. We assume a 10-bit parent filter index, allowing us to index up to 1024 filters, which is enough for commonly used neural network models.

We have implemented both the sparse synchronous and asynchronous schedulers in software. They take as an input a trained network and SASCHA configuration and output iteration counts. The asynchronous scheduler cannot guarantee perfect utilization if the total number of balanced groups corresponding to a set of partial filters is not divisible by M. To assess the effectiveness of the asynchronous scheduler, we also consider the *ideal* scheduler, which is the asynchronous scheduler for a single row, which can always be perfectly utilized. Combined results for all three schedulers for the convolutional layers of the CIFAR-10 TinyConv network [24], using a $N=32, M=32, K=32$ SASCHA array with different group sizes are shown in Figure 12 a). All configurations have $C=1$, and $CP/G=1$ (iso-throughput PEs). When parallel streams are used to compensate for the loss of MAC throughput, larger group sizes are better at converting sparsity into lower iteration overhead. Using a group size of 8 has, on average, 1.3X and 1.6X lower iteration overhead than when using group sizes of 4 and 2, respectively. With $G=8$, iteration overhead starts decreasing at as low as 10%, sparsity, while $G=4$ and $G=2$ require sparsity of at least 40% and 70% to start showing benefits. In the best case of $G=8$, our asynchronous scheduler has on average 1.4X lower iteration overhead than the sparse synchronous one, up to 2.2X at 90% sparsity. It is also, on average, within 11% of the ideal scheduler.

Figure 12 b) compares the iteration overhead with different group sizes and capacity while maintaining $CP/G=1$ when using the asynchronous scheduler. It shows that using parallel stream processing is a more efficient way of using the additional area than increasing the capacity. SASCHA with $G=8$, $C=1$,$P=8$ is on average 1.2X and 1.46X faster than $C=2$,$P=4$ and $C=4$,$P=2$ configurations, respectively. For group size of 4, $C=1$,$P=4$ is on average 1.15X faster than $C=2$,$P=2$. This conclusion is unique to SC - conventional fixed- and floating-point accelerators do not have access to stream parallelism design trade-offs. Therefore, we will focus on configurations with $C=1$ and $P=G$, as the optimal SASCHA PE choices.

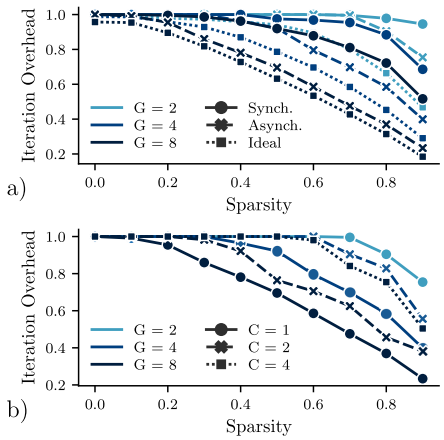

Fig. 12. Iteration overhead using different sparse scheduling methods (a) and different group sizes and capacity using the sparse asynchronous scheduler (b).

C. Memory Organization

While beneficial from the point of view of runtime, using the asynchronous scheduler comes at a cost. Synchronous scheduling allows simple combining of partial sums from different balanced groups corresponding to the same partial filter, as they are assigned sequentially to the same row, as shown in Figure 11 b). It means that individual balanced groups do not generate multiple partial sum memory accesses, which can be very costly. For the asynchronous scheduler, partial filters can now be distributed across multiple rows, making such combining non-trivial. To avoid generating unnecessary memory accesses, we implement merging logic on the datapath used for flushing partial sums out of the PE array, as shown in Figure 10. By having parent filter indices (IDX_F) associated with partial sums, those corresponding to the same parent filter can be accumulated when being flushed out of the array and before being written back to activation memory.

We used our scheduler to model the number of memory accesses for activations, weights, and partial sums, for dense and sparse architectures using different schedulers and dataflows. Results of the convolutional layers of the CIFAR-10 TinyConv network, at 90% sparsity, are shown in Figure 13. Output stationary dataflow is impossible when using the asynchronous scheduler due to balanced groups belonging to the same parent filter being potentially distributed across different PE units. Input stationary dataflow is, therefore, the best choice of the dataflow for the SASCHA asynchronous scheduler, cutting down the number of memory accesses by 18% compared to weight stationary, at high sparsity levels. It is also within 7% and 13% of the best achievable dataflow for dense and synchronous scheduling, respectively.

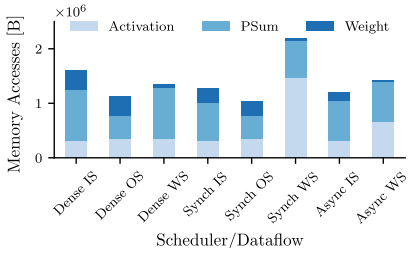

Fig. 13. Number of memory accesses in bytes for the convolutional layers of CIFAR-10 TinyConv network at 90% sparsity, depending on the choice of scheduling and dataflow. All results for $M = 32, N = 32$, and $K = 32$. Sparse results for $G{=}8, C{=}1$, and $P{=}8$.

VI. Bit-Slicing Weights

While SASCHA architecture, discussed in the previous section, shows high latency improvements on sparse networks, it can at best maintain the same throughput when running dense ones. In this section, we show how higher effective sparsity and more efficient hardware can be extracted by exploiting intra-value sparsity of unpruned weights through *bit-slicing*. By bit-slicing, we mean decomposing weights into smaller slices, and processing them individually, then scaling the results depending on the LSB position of a given slice. For example, two-way slicing involves splitting the fixed-point weight value into equally sized MSB and LSB slices, multiplying them individually with each corresponding activation, scaling the MSB result, and adding both results. A similar technique has been proposed in the context of SC in [8], however, it is used only as a means of reducing the computation stream length, and not exploiting additional operand sparsity. For SASCHA, we assume an equal split between the number of most significant and least significant bits. While other split sizes and granularities are possible, their analysis is beyond the scope of this work.

The idea behind improving sparsity with bit-slicing comes from the observation that if we divide a set of values into bit-slices, the resulting sparsity, i.e., the percentage of slices that are completely zero, will be at worst the same, and at best higher than sparsity of non-sliced values. Given that weights in neural networks exhibit zero-centered bell-shaped distributions, we would expect the sparsity of MSB bit-slices to be even higher. To verify this, we evaluated the overall, MSB and LSB sparsity in the convolutional layers of the CIFAR-10 TinyConv network, at different network pruning levels. Results are shown in Figure 14 a). We can see that even for the unpruned networks, MSB slices exhibit very high sparsity - 64%. While the high MSB sparsity could help when processing dense networks, we expect the benefits to be minuscule at high sparsity levels - the MSB and LSB sparsity of the network pruned to 90% is 90% and 90.1% respectively, meaning there is not a lot of additional computation savings available.

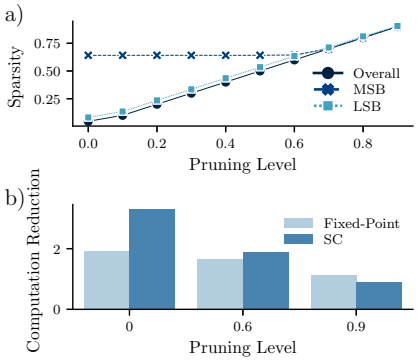

Fig. 14. Overall, MSB and LSB sparsity for (a) and reduction in sliced multiplication area x delay cost relative to non-sliced cost for SC and fixed-point (b), at different pruning levels for CIFAR-10 TinyConv.

We assume that only one of the operands, weight, is sliced, as we mainly care about extracting more sparsity on the weight side. In a naive implementation, where each slice is computed with the native stream length, e.g., 64, this would lead to a minuscule runtime reduction at best, since most of the LSB slices are not sparse. However, we can capitalize on the fact that the MSB slice contribution to computation will be much higher than the LSB one. By computing the MSB part with the original stream length, e.g., 64, and the LSB part with a shorter one, e.g., 8, and then scaling the LSB result and adding it to the MSB one, we can approximate the original result. We refer to this as asymmetric-stream slicing, as opposed to prior works which used symmetric stream lengths [8]. The comparison of non-sliced and sliced unipolar multiplication is shown schematically in Figure 15 a) and b), respectively. As can be seen, when using sliced operands, the size of the SNG and its buffers can be reduced, which improves the area.

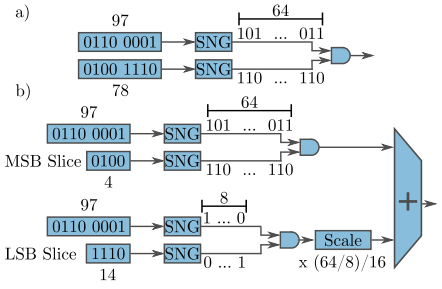

Fig. 15. SC unipolar multiplication a), and sliced multiplication b).

The slicing technique is also applicable to fixed-point computation. However, while in the fixed-point case it would result in the same precision used for both the MSB and LSB parts, SC makes asymmetric precision possible. In terms of area-delay product cost of a multiplier, 4-bit MSB/LSB slicing results in a 2.24X reduction for each of the parts, compared to a regular multiplier. Assuming 64-bit

long MSB streams and 8-bit long LSB streams in the SC case, there is no reduction in the cost for the MSB part, but there is an 8X cost reduction in the LSB part. When considering the sparsity levels of each part in Figure 14 a), we see that at low pruning levels the number of LSB slice multiplications dominate, and we expect the asymmetric SC precision to give us higher benefits than symmetric fixed-point slicing. To evaluate this hypothesis, we multiplied the proportion of non-zero MSB and LSB slices by their area-delay cost reduction factors when slicing, and combined the results to show an overall ideal reduction in multiplication cost. Results, normalized to ideal non-sliced sparsity computation reduction, are shown in Figure 14 b). For low and moderate levels of pruning, SC slicing achieves higher cost reduction than fixed-point one - 2.6X on average, compared to 1.8X, respectively. While not as effective at high sparsity levels, asymmetric precision of slicing allows us to show improvement even at low sparsity levels, as shown in Section VII.

There is a concern about how slicing will affect computation precision. To evaluate that, we analyzed the root mean square error (RMSE) of stochastic unipolar multiplication in both the non-slicing and slicing scenarios using the same stream lengths as shown in Figure 15. The operands are drawn from activation and weight distributions of the CIFAR-10 TinyConv model, where weight distribution is used for the sliced operand. Average RMSE across 1000 trials is shown in Table I, and the sliced multiplication error is within 30% of the non-sliced one. We will discuss network-level accuracy and performance impact of slicing in Section VII.

TABLE I
RMSE OF UNIPOLAR MULTIPLICATION WITH AND WITHOUT BIT-SLICING, W.R.T. FLOATING-POINT PRECISION, FOR DIFFERENT STREAM LENGTHS (1000 TRIALS). LSB STREAM LENGTH IS 8.

	Stream Length	16	32	64	128	256
RMSE	No Slicing	4.49	3.14	2.26	1.531	0.98
	With Slicing	4.97	3.28	2.78	1.867	1.31

To summarize, bit-slicing allows us to expose higher levels of sparsity present in weights to SASCHA compute. By utilizing the asymmetric sparsity of MSB and LSB slices, lower relative precision required for the latter, and SC's unique precision-latency trade-off space, bit-slicing enables SASCHA to show performance improvements even on dense networks, as shown in Section VII. Bit-slicing can be handled natively by SASCHA, at the cost of underutilizing the sparse storage and SNG buffers (provisioned for 8-bit values). However, for completeness' sake, we also evaluate the SASCHA-S variant, which is dedicated to weight-sliced networks, by having reduced weight storage and SNG buffers. From the scheduler's point of view, each sliced part can be treated as a separate layer. The outputs of those layers are then scaled and added element-wise.

VII. Evaluation & Results

A. SASCHA Accuracy

All models are trained using PyTorch. The training setup is similar to the one used in [28], but with added layer-wise magnitude-based pruning. TinyConv, VGG-11 and VGG-16 [42] are trained on the CIFAR-10 dataset, and ResNet-18 and -34 are trained on ImageNet dataset. The fully-connected layers of VGG-16 are reduced to 512 to accommodate the small CIFAR-10 dataset. All models are trained with 64-bit streams. Bit-slicing has little effect on accuracy on VGG-11, and the accuracy with and without bit-slicing differs by less than 0.7%, as shown in Figure 16.

Figure 17 summarizes the results on CIFAR-10. Compared to [28], and [38] which also use OR accumulation, accuracy has been

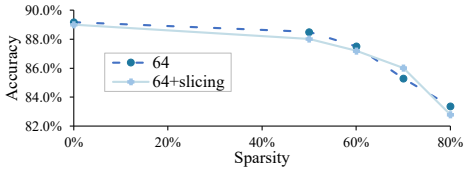

Fig. 16. Accuracy of CIFAR-10 VGG-11 with different sparsity levels. 0% sparsity means no sparsity constraint.

improved by switching the order of ReLU, pooling, and batch normalization (bn). While previous works use pooling-bn-ReLU order to achieve spatial pooling, SASCHA does not have the same constraint and can use the more optimal bn-ReLU-pooling. This change improves accuracy by 2% for both TinyConv and VGG-16 on CIFAR-10. Due to its small size, TinyConv is less resilient to sparsity. At 60% sparsity, accuracy drops by 0.3-0.8%. Accuracy drop using VGG-16 is milder, with no noticeable drop in accuracy when using 60% sparsity and $\approx 4.5\%$ using 90% sparsity. While slicing reduces accuracy when the models are dense, the accuracy gap reduces with increased sparsity. While the gap is 1-1.7% for the dense models, the gap is negligible or even reverses at maximal sparsity usable for each model.

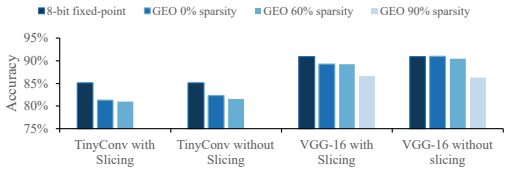

Fig. 17. SASCHA CIFAR-10 Top-1 accuracy with dense and sparse networks.

Figure 18 summarizes the results of Resnet-18 and Resnet-34 on ImageNet. We use a higher accuracy version of stochastic computing, denoted as GEO+. In GEO+, full binary accumulation replaces partial binary accumulation, eliminating OR accumulation. Since only inputs within the same OR accumulation window require different seeds, all multiplications can use the same seed pair. We further improve the multiplication accuracy by choosing the seed pair that produces the lowest error. With this modification, SASCHA achieves comparable accuracy to 8-bit fixed-point throughout different sparsity levels. Because uGEMM achieves accuracy comparable to 8-bit fixed-point without retraining, as reported by [48], we expect it will behave similarly with pruning. Since no efficient stream simulation functions are available for uGEMM, we have not trained it separately.

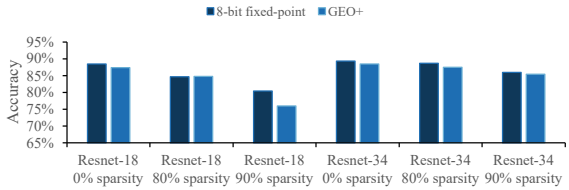

Fig. 18. SASCHA ImageNet Top-5 accuracy with dense and sparse networks.

B. Performance Results

To evaluate SASCHA performance, we implement the entire GEMM array, including individual components such as SASCHA PEs, SNGs, LFSRs, merge blocks, vector unit, and the necessary glue logic using Verilog HDL, and synthesize them using TSMC 28nm library and Cadence Genus synthesis tool, at 400MHz clock frequency. We use the results to estimate the overall area, power, and energy consumption of different SASCHA configurations. We use the scheduler described in Section V to estimate runtime and memory

accesses required for different networks and levels of sparsity. We use the CACTI tool to estimate the cost of memory accesses [32]. To demonstrate how SASCHA is agnostic to the underlying style of SC computation, we evaluate three configurations: SASCHA-GEO, -GEO+, and -uGEMM, using the PEs as described in Section IV.

We compare SASCHA to GEO ULP [28], which uses the same underlying computation as SASCHA-GEO but is optimized towards convolutional layers. We use the same simulator as described in [28] to estimate performance. Also, for each of the SASCHA versions, we compare it with a corresponding dense version, with the same number of PEs, and no stream parallelism, referred to as GEMM-GEO, -GEO+, and -uGEMM. The last one is very similar to the original uGEMM architecture [48] but uses binary instead of streaming accumulation. To keep the results consistent with uGEMM, we only report the logic area without including memories. For a comparison with sparse fixed-point accelerators, we use SCNN [35] and Laconic [40]. For a fair comparison with SASCHA, we omitted the area of on-chip memories, based on the area breakdowns provided by the original works, and scaled the area of compute and buffers to account for the change in precision from 16 to 8 bits. We also scaled the technology node to 28nm using the scaling equations provided in [44]. We then configured the number of each accelerator's PEs to roughly match the area of SASCHA. We refer to those configurations as SCNN-M and Laconic-M, respectively. Their execution is modeled using DNNSim [1]. We omit ResNet-34 results on Laconic-M, as the simulator was not able to schedule the computation successfully. All designs are iso-frequency.

Based on the results discussed in the previous sections, we limited our exploration to configurations with $G=4$ or 8, as they are better at extracting sparsity benefits than $G=2$. We also limit ourselves to configurations with $C=1$ and $G=P$, since they provide a better area-throughput trade-off than ones with $C>1$ and lower parallelism. This choice, enabled by parallel stream processing unique to SC, allowed us to arrive at a different design point that would be optimal for fixed-point PEs, as exemplified by [9]. Based on area estimates, for the SASCHA-GEO and -GEO+ versions we picked $M=32$,$N=16$,$K=32$,$G=4$,$C=1$,$G=4$, referred to as *SASCHA-GEO* and *SASCHA-GEO+ 4/1/4* as they are within 8% of the logic area of GEO ULP. We also include SASCHA-GEO and -GEO+ configurations with the same M,N,K, and C, but with G and P equal to 8, referred to as *SASCHA-GEO* and *SASCHA GEO+ 8/1/8*. Those configurations, while consuming only 23-42% larger area than the 4/1/4 configurations, are much better at extracting sparsity benefits, as we will show shortly. Figure 19 shows the area and power breakdown of SASCHA-GEO 8/1/8, based on individual module synthesis, showing a more balanced, and not SNG-dominated, distribution compared to [28]. SASCHA achieves this through the combination of GEMM-style architecture and sparsity-oriented design. For the uGEMM variant, due to a much larger PE area, we size the array with $M=16$,$N=16$,$K=16$, for both 4/1/4 and 8/1/8 versions, which brings them close to the iso-area with the GEO and GEO+ variants. Finally, we include two SASCHA-GEO and GEO+ configurations with the same parameters as the ones above, but with bit-slicing support, referred to as *SASCHA-GEO-S* and *SASCHA-GEO+-S*. We do not include SASCHA-uGEMM slicing configurations, as the impact of slicing on the uGEMM-style PE accuracy is beyond the scope of this work. All non-slicing configurations use a stream length of 64, while the slicing ones use 64-bit long streams for the MSB computation and 8-bit long streams for LSB. We assume that memory bandwidth is provisioned for maximum expected throughput. When reporting sparse results, we pick maximum sparsity at which SC accuracy is within 4% of fixed-point, GEO for CIFAR-10, and GEO+ for ImageNet, sliced or non-sliced, whichever is higher.

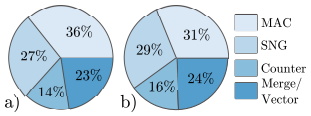

Fig. 19. Area (a) and power (b) breakdown of SASCHA GEO 8/1/8.

Final results are shown in Table II, for the TinyCONV and VGG-11 and -16 networks on the CIFAR-10 dataset, and ResNet-18 and -34 networks on the ImageNet dataset. Results are shown at two different levels of sparsity. We first analyze the performance of dense networks. Compared to the dense GEMM versions with the same array size, non-slicing SASCHA configurations maintain a similar throughput while suffering at most 31% loss in energy efficiency. This is expected - while stream parallelism is used to recover runtime, it lowers PE energy efficiency through lower SNG reuse. Further, in the dense case, SASCHA will require more overall memory accesses due to indexing overheads and asynchronous scheduling. The exception is SASCHA-uGEMM, where going from dense to sparse-parallel PEs can actually reduce area and power, due to the large size of the former, which results in marginally higher energy efficiency in the dense case. In the case of CIFAR-10 VGG networks, where the network has high *natural* weight sparsity without pruning, throughput, and energy efficiency are improved by up to 3.6X and 4.4X, respectively. Bit slicing SASCHA-S configurations perform much better on unpruned networks, as expected. By exploiting high inherent MSB slice sparsity, they can improve the throughput by up to 1.33X over GEMM Dense and improve energy efficiency by up to 1.2X, except in the CIFAR-10 VGG case, where improvements are higher. GEO achieves higher throughput and energy efficiency on the dense TinyConv, which comes from the fact that its architecture is highly optimized towards convolutional layers. Compared to SCNN-M, SASCHA-GEO and GEO+ can improve the throughput by 4X-8.7X, owing to the higher efficiency of SC compute over fixed-point. SASCHA-uGEMM configurations, despite having 4x fewer PEs than the other configurations, still outperform SCNN-M by 2.2X. Compared to Laconic-M, GEO and GEO+ configurations have up to 19X, while the uGEMM ones have up to 7.8X speedup on dense networks.

When running moderately sparse (60%) TinyConv, SASCHA accelerators improve the throughput by up to 1.92X compared to the dense variants and up to 1.94X compared to the unpruned network on the respective SASCHA configurations. At 90% sparsity, SASCHA configurations can be up to 6.5X faster and 5.5X more energy-efficient than the dense versions. Compared to their respective versions running dense networks, they improve runtime by up to 8.8X and energy efficiency by up to 10.1X. SASCHA-GEO and GEO+ maintain a 1.5X to 4X throughput advantage over SCNN-M on sparse convolutional networks, despite SCNN taking advantage of both weight and activation sparsity, while SASCHA only utilizes the former. SASCHA-uGEMM 8/1/8 outperforms SCNN-M by up to 2.2X on convolutional networks. While sliced configurations are not as efficient at high sparsity, they still achieve up to 4.7X and 3.4X throughput improvement over GEMM Dense and SCNN-M, respectively, on sparse convolutional networks. Laconic-M extracts most of its benefits on a *bit-sparsity* level even without pruning and does not show large improvements when small weights are removed.

In Table III we show the achieved weight compression ratio for all four evaluated SASCHA configurations and three evaluated networks. For weight compression, the underlying PE architecture does not matter. As expected, at high sparsity, bit-slicing configurations have lower effective compression due to indexing overhead affecting both

TABLE II
AREA [mm^2], POWER [MW], THROUGHPUT [FR/S] AND ENERGY-EFFICIENCY [FR/J] FOR DIFFERENT ACCELERATORS, MODELS AND DATASETS, AND SPARSITY.

			CIFAR-10 TinyConv				CIFAR-10 VGG-11				CIFAR-10 VGG-16				ImageNet ResNet-18				ImageNet ResNet-34			
			Sparsity 0%		60%		Sparsity 0%		70%		Sparsity 0%		90%		Sparsity 0%		80%		Sparsity 0%		90%	
Architecture	Area [mm2]	Power [mW]	Fr/s [k]	Fr/J [k]	Fr/s [k]	Fr/J [k]	Fr/s [k]	Fr/J [k]	Fr/s [k]	Fr/J [k]	Fr/s [k]	Fr/J [k]	Fr/s [k]	Fr/J [k]	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J
SCNN-M	0.3	-	1.7	-	3.0	-	0.11	-	0.42	-	0.11	-	0.42	-	8	-	79	-	4	-	46	-
Laconic-M	0.2	-	3.4	-	3.6	-	0.07	-	0.07	-	0.12	-	0.12	-	46	-	49	-	-	-	-	-
GEO ULP	0.24	50	10.6	240	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GEMM-GEO	0.20	78	6.6	72	-	-	0.5	5.5	-	-	0.26	2.9	-	-	50	499	-	-	27	299	-	-
SASCHA GEO 4/1/4	0.21	75	6.7	73.5	8.8	93.5	0.8	8.2	1.1	10.5	0.65	7.0	0.95	9.8	50	525	105	969	27	301	76	754
SASCHA GEO 8/1/8	0.30	93	6.7	60.2	12.2	102.7	0.9	7.2	1.5	11.1	0.81	7.0	1.39	11.2	50	435	151	1078	27	246	121	921
SASCHA-GEO-S 4/1/4	0.19	62	6.8	76.2	7.5	83.7	1.3	10.8	1.5	11.9	0.68	7.3	0.84	8.4	46	481	97	821	29	357	68	736
SASCHA-GEO-S 8/1/8	0.26	80	8.8	73.6	10.3	86.1	1.6	10.1	2.1	11.6	0.91	7.3	1.21	8.9	47	435	140	863	36	331	108	894
GEMM-GEO+	0.24	81	6.6	70	-	-	0.5	5.3	-	-	0.26	2.8	-	-	50	485	-	-	27	289	-	-
SASCHA GEO+ 4/1/4	0.23	78	6.7	71.0	8.8	90.4	0.8	8.0	1.1	10.2	0.65	6.8	0.95	9.5	50	508	105	941	27	290	76	731
SASCHA GEO+ 8/1/8	0.32	99	6.7	56.1	12.2	96.1	0.9	6.7	1.5	10.5	0.81	6.5	1.39	10.5	50	406	151	1019	27	229	121	867
SASCHA-GEO+-S 4/1/4	0.21	65	6.8	75.4	7.5	82.8	1.3	10.7	1.5	11.8	0.68	7.2	0.84	8.3	46	476	97	814	29	353	68	755
SASCHA-GEO+-S 8/1/8	0.27	83	8.8	72.5	10.3	84.8	1.6	10.0	2.1	11.4	0.91	7.2	1.21	8.8	47	375	140	854	36	325	108	881
GEMM-uGEMM	0.34	112	2.1	17.5	-	-	0.1	1.0	-	-	0.06	0.55	-	-	13	106	-	-	7	57	-	-
SASCHA uGEMM 4/1/4	0.26	82	2.0	23.0	2.9	32.5	0.2	2.3	0.3	3.4	0.19	2.1	0.36	3.9	13	148	33	354	7	77	26	272
SASCHA uGEMM 8/1/8	0.27	86	2.0	21.7	4.0	40.8	0.2	2.2	0.4	4.1	0.23	2.4	0.57	5.5	13	142	48	475	7	72	42	399

the LSB and MSB slices. The exception is the VGG-11 network, where a combination of high natural sparsity and relatively low pruned sparsity allows the slicing configurations to come out ahead. SASCHA 8/1/8 has a higher compression ratio at 90% sparsity compared to 4/1/4, despite higher indexing overhead. This is due to more efficient weight storage - for the same dot product width, it will store half of the weights in each memory word compared to SASCHA 4/1/4. For the latter, at high sparsity, many of those words will be underutilized.

TABLE III
WEIGHT COMPRESSION RATIO FOR DIFFERENT SASCHA CONFIGURATIONS, NETWORKS, AND SPARSITY LEVELS.

Model Sparsity	TinyConv 60%	VGG-11 70%	VGG-16 90%	ResNet-18 80%	ResNet-34 90%
SASCHA 4/1/4	1.09	2.00	4.22	1.43	2.11
SASCHA 8/1/8	1.22	2.27	5.34	1.75	2.85
SASCHA-S 4/1/4	0.90	2.34	4.07	1.22	1.68
SASCHA-S 8/1/8	0.93	2.52	5.19	1.38	2.10

VIII. RELATED WORK

Sparse Accelerators. Exploiting sparsity to improve performance in hardware has been extensively studied for floating- and fixed-point accelerators. Some of the prior works only try to exploit the sparsity of one operand type, like Cnvlutin (activations) [3], or Cambricon-X (weights) [50]. Others, like Cambricon-S [52], SCNN [35], Bit-Tactical [25], or TensorDash [30], can exploit both activation and weight sparsity, often through a combination of static and dynamic scheduling. While most accelerators opt for some form of operand advancing through a staging window [15], [25], [30], others like SCNN [35] or MatRaptor [43] rely on multiplying all non-zero operands and mapping the results to appropriate partial sums afterwards. Due to the high cost of detecting and supporting sparse execution, the majority of sparse accelerators focus on higher precision arithmetic like 32-bit and 16-bit floating-point or 16-bit fixed-point [15], [30], [35], [43], [50]. Such datatypes and accelerators are more suited towards training neural networks. In contrast, SASCHA focuses on approximate edge inference, where quantized, 8-bit, and lower fixed-point precision has become a standard [7]. Despite only focusing on weight sparsity, it can outperform fixed-point accelerators that also exploit sparse activations, thanks to highly efficient stochastic computation.

Stochastic Computing Accelerators. While stochastic computing has been enjoying a recent renaissance due to its synergies with deep learning algorithms; there is a surprising lack of configurable, system-level designs available. Few examples include ACOUSTIC [38], which is an accelerator targeting convolutional neural networks specifically, GEO [28], which improves on ACOUSTIC's accuracy and performance, uGEMM [48] and StoRM [8] , which are flexible general matrix multiply engines, and SCOPE [27], and in-memory DRAM accelerator. We compare SASCHA with GEO and uGEMM, which target similar, low-precision edge inference. StoRM can be considered a specific case of uGEMM with specialized PEs. While it explores operand slicing, it processes them in a spatially unrolled manner, requiring symmetric stream lengths and not being able to utilize additional sparsity exposed by it, unlike SASCHA. SCOPE is a data center accelerator with area requirements orders of magnitude higher than SASCHA. Other works, like HEIF [29], or SC-DCNN [37] have proposed generating custom hardware for specific neural network models. Those approaches often lead to the impractically high area and are of limited utility in the rapidly changing neural network landscape. BISC-MVM [41], and SkippyNN [20], propose more accurate stochastic multiplier designs but are limited to fixed-point addition and do not present system-level elaboration. Finally, some recent works propose methods of doing stochastic computing in a deterministic manner, without introducing any error [33]. However, they often require long stream lengths for processing and are not competitive in terms of latency and energy.

IX. CONCLUSION

In this work, we present SASCHA - a sparsity-aware neural network accelerator architecture using stochastic computing. SASCHA exploits sparsity in a way that synergizes with the main advantages of SC. It encompasses a sparse multiply-accumulate block design, GEMM accelerator architecture, and asynchronous scheduling method. Further, we propose a bit-slicing method unique to SC that can exploit sub-operand sparsity even in dense networks. At 90% weight sparsity, SASCHA can be up to 6.5X faster and 5.5X more energy-efficient than comparable dense SC accelerators with a similar area, and up to 8.7X faster than sparse fixed-point accelerators, without sacrificing performance on dense networks. Our future work will explore the interplay between sparsity and bitstream length in the context of SC.

REFERENCES

[1] "DNNSim," 2000. [Online]. Available: https://github.com/isakedo/DNNsim

[2] 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.

[3] J. Albericio, P. Judd, T. Hetherington *et al.*, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 1–13, 2016.

[4] 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.

[5] Y. Chen, T. Luo, S. Liu *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.

[6] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks," *arXiv preprint arXiv:1710.09282*, 2017. [Online]. Available: http://arxiv.org/abs/1710.09282

[7] T. W. Chin, P. I. Chuang, V. Chandra, and D. Marculescu, "One Weight Bitwidth to Rule Them All," *European Conference on Computer Vision Workshops*, 2020.

[8] 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.

[9] J. Choquette, E. Lee, R. Krashinsky *et al.*, "The A100 Datacenter GPU and Ampere Architecture," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 48–50.

[10] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[11] C. Deng, S. Yang, S. Liao, and B. Qian, XuehaiYuan, "GoSPA : An Energy-efficient High-performance Globally Optimized SParse Convolutional Neural Network Accelerator," in *International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1110–1123.

[12] C. Eckert, X. Wang, J. Wang *et al.*, "Neural Cache : Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 383–396.

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

[14] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "SparTen: A Sparse Tensor Accelerator for Convolutional NeuralNetworks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[15] Z. Gong, H. Ji, C. W. Fletcher *et al.*, "Save: Sparsity-aware vector engine for accelerating DNN training and inference on CPUs," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 796–810.

[16] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," in *Advances in neural information processing systems*, 2015, pp. 1135–1143. [Online]. Available: http://arxiv.org/abs/1506.02626

[17] B. Hassibi, D. G. Stork, and G. J. Wolff, "Optimal brain surgeon and general network pruning," pp. 293–299, 1993.

[18] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, 2017, pp. 1398–1406.

[19] K. Hegde, H. Asghari-Moghaddam, M. Pellauer *et al.*, "ExTensor: An accelerator for sparse tensor algebra," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2019, pp. 319–333.

[20] R. Hojabr, K. Givaki, S. M. R. Tayaranian *et al.*, "SkippyNN : An Embedded Stochastic-Computing Accelerator for Convolutional Neural Networks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. ACM, 2019, pp. 1–6.

[21] Y. Jia, *Learning Semantic Image Representations at a Large Scale*. University of California, Berkeley, 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.html

[22] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-Serial Deep Neural Network Computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[23] Y. G. Kim and C. J. Wu, "Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2020-Octob, pp. 1082–1096, 2020.

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

[25] A. D. Lascorz, P. Judd, D. M. Stuart *et al.*, "Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 749–763, 2019.

[26] S. Li, D. Niu, K. T. Malladi *et al.*, "DRISA : A DRAM-based Reconfigurable In-Situ Accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 14, 2017, pp. 288–301. [Online]. Available: https://doi.org/10.1145/3123939.3123977%0Ahttps://energyestimation.mit.edu/

[27] S. Li, A. Oliver Glova, X. Hu *et al.*, "SCOPE: A Stochastic Computing Engine for DRAM-based In-situ Accelerator," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 696–709.

[28] 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.

[29] Z. Li, J. Li, A. Ren *et al.*, "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.

[30] M. Mahmoud, I. Edo, A. H. Zadeh *et al.*, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2020, pp. 781–795.

[31] A. Mishra, J. Albericio, J. Pool *et al.*, "Accelerating sparse deep neural networks," *arXiv preprint arXiv:2104.08378*, 2021.

[32] 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. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.1426&rep=rep1&type=pdf

[33] M. H. Najafi, D. Jenson, D. J. Lilja, and M. D. Riedel, "Performing Stochastic Computation Deterministically," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2925–2938, 2019.

[34] S. Pal, J. Beaumont, D. H. Park *et al.*, "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator," *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2018-Febru, pp. 724–736, 2018.

[35] A. Parashar, M. Rhu, A. Mukkara *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[36] J.-s. Park, J.-w. Jang, H. Lee *et al.*, "A 6K-MAC Feature-Map-Sparsity-Aware Neural Processing Unit in 5nm Flagship Mobile SoC," in *2021 IEEE International Solid-State Circuits Conference-(ISSCC)*, 2021, pp. 152–153.

[37] A. Ren, J. Li, Z. Li *et al.*, "SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 405–418, 2017. [Online]. Available: http://arxiv.org/abs/1611.05939

[38] W. Romaszkan, T. Li, T. Melton *et al.*, "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.

[39] V. Sehwag, N. Prasad, and I. Chakrabarti, "A Parallel Stochastic Number Generator with Bit Permutation Networks," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 2, pp. 231–235, 2018.

[40] S. Sharify, A. D. Lascorz, M. Mahmoud *et al.*, "Laconic deep learning inference acceleration," *Proceedings - International Symposium on Computer Architecture*, pp. 304–317, 2019.

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

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

[43] N. Srivastava, H. Jin, J. Liu *et al.*, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 766–780.

[44] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," *Integration*, vol. 58, no. January, pp. 74–81, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.vlsi.2017.02.002

[45] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel Multi Channel convolution using General Matrix Multiplication," *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, pp. 19–24, 2017.

[46] S. Wang, A. Pathania, and T. Mitra, "Neural Network Inference on Mobile SoCs," *IEEE Design & Test*, vol. 37, no. 5, pp. 50–57, 2020.

[47] C. J. Wu, D. Brooks, K. Chen *et al.*, "Machine learning at facebook: Understanding inference at the edge," *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019*, pp. 331–344, 2019.

[48] D. Wu, J. Li, R. Yin *et al.*, "uGEMM : Unary Computing Architecture for GEMM Applications," *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 377–390, 2020.

[49] J. Yu, A. Lukefahr, D. Palframan *et al.*, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3080215

[50] S. Zhang, Z. Du, L. Zhang *et al.*, "Cambricon-X : An Accelerator for Sparse Neural Networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[51] G. Zhong, A. Dubey, T. Cheng, and T. Mitra, "Synergy: A HW/SW framework for high throughput CNNs on embedded heterogeneous SoC," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 2, pp. 1–23, 2019.

[52] X. Zhou, Z. Du, Q. Guo *et al.*, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*. IEEE, 2018, pp. 15–28.